Correct Tools
for Formal Methods in Software Engineering

Kumulative Habilitationsschrift

vorgelegt zur Erlangung
der
Lehrbefugnis für Informatik
an der
Fakultät für Angewandte Wissenschaften
der
Albert-Ludwigs-Universität Freiburg

von

Burkhart Wolff

Freiburg, April 2005

Abstract

The development of *tools* for program analysis, verification and refinement is a prerequisite for the proliferation of formal methods in industry and research. While most tools were directly implemented in a programming language, the ultimate goal of this work is to represent widely known formal methods in a so-called *logical framework* **by their semantics** using a particular representation technique — called *shallow embedding* — motivated by more efficient deduction. Based on this representation, symbolic computations in tool implementations can be based on formally proven correct *derived rules*. As such, this correctness-oriented approach has been known for a while and has been criticized for a number of shortcomings:

1. the application range of embeddings in logical frameworks is limited to very small and artificially designed languages,
2. their application is impossible when the formal specification method is still under development,
3. embedding the semantics conservatively and deriving some rules on this basis does not imply that there is a comprehensive support of a method that is technically powerful enough for applications,
4. the integration in a more global software engineering process and its pragmatics is too difficult, and
5. the usability of embeddings is doubtful even if one is targeting at the (fairly small market of) proof environments.

In contrast to this criticism, we claim that our approach is feasible. We substantiate this by developing:

1. suitable embeddings for widely used formal methods, including process-oriented, data-oriented and object-oriented specification methods (CSP, Z, UML/OCL),
2. abstractions and aspect-oriented structuring techniques allowing for the quick development of semantic variants enabling the study consequences of changes in formal methods under development (like UML/OCL),
3. particular techniques for generating library theories, for supporting particular deduction styles in proofs, for specialized deduction support for concrete development methodologies,
4. different scenarios of the integration of the developed tools in conventional tool chains in software engineering, and
5. front-ends for light-weight integration into tool chains (like HOL-Z 2.0) or prototypic encapsulation of logical embeddings into generic graphical user-interfaces for a more comprehensive encapsulation.

Finally, we validate one of these tool chains (HOL-Z 2.0) by a substantial case-study in the field of computer security.

Contents

Introduction 1
Burkhart Wolff

I Selected Papers: Embeddings

A Corrected Failure-Divergence Model for CSP in Isabelle/HOL 27
Haykal Tej and Burkhart Wolff

HOL-Z 2.0: A Proof Environment for Z-Specifications 47
Achim D. Brucker and Frank Rittinger and Burkhart Wolff

UML/OCL: Semantics, Calculi, and Applications in Refinement and Test . 65
Achim D. Brucker and Burkhart Wolff

Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL 123
Nicole Rauch and Burkhart Wolff

II Selected Papers: Special Deduction for Method Support

Correct and User-Friendly Implementation of Transformation Systems 143
Kolyang, Thomas Santen and Burkhart Wolff

TAS - A Generic Window Inference System. 163
Christoph Lüth and Burkhart Wolff

Using Theory Morphisms for Implementing Formal Methods Tools 181
Achim D. Brucker and Burkhart Wolff

Symbolic Test Case Generation For Primitive Recursive Functions 205
Achim D. Brucker and Burkhart Wolff

III Selected Papers: Encapsulation and Tool Integration

HOL-Z in the UniForM-Workbench - a Case Study in Tool Integration 225
Christoph Lüth and Einar Karlsen and Kolyang and Stefan Westmeier and Burkhart Wolff

Functional Design and Implementation of Graphical User Interfaces for Theorem Provers 245
Christoph Lüth and Burkhart Wolff

IV Selected Papers: Validation by Case Studies

A Verification-Approach for Applied System Security 269
Achim D. Brucker and Burkhart Wolff

Author Index 285

Selected Paper Bibliography

1. Tej, H., Wolff, B.: A Corrected Failure-Divergence Model for CSP in Isabelle/HOL. In Fitzgerald, J., Jones, C., Lucas, P., eds.: Formal Methods Europe 97. LNCS 1313, pp. 318–337, 1997.
2. Brucker, A.D., Rittinger, F., Wolff, B.: HOL-Z 2.0: A proof environment for Z-specifications. Journal of Universal Computer Science **9** (2), pp. 52-172, Elsevier Science Publishers, 2003.
3. Brucker, A., Wolff, B.: UML/OCL — Semantics, Calculi, and Applications in Refinement and Test. (Conditionally accepted by Acta Informatica, Manuskript 0204).
4. Rauch, N., Wolff, B.: Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL. In: Electronic Notes in Theoretical Computer Science. Volume 80., Elsevier Science Publishers, 2003.
5. Kolyang, Santen, T., Wolff, B.: Correct and User-friendly Implementation of Transformation Systems. In Gaudel, M.C., Woodcock, J., eds.: Formal Methods Europe. LNCS 1051, pp. 629–648, 1996.
6. Lüth, C., Wolff, B.: TAS — A Generic Window Inference System. In Harrison, J., Aagaard, M., eds.: International Conference of Theorem Proving in Higher Order Logics (TPHOLs). LNCS 1869, pp. 405–422, 2000.
7. Brucker, A.D., Wolff, B.: Using Theory Morphisms for Implementing Formal Methods Tools. In Geuvers, H., Wiedijk, F., eds.: Types 2002, Proceedings of the workshop Types for Proof and Programs. LNCS 2646, pp. 59–77, 2003.
8. Bucker,A., Wolff, B.: Symbolic Test Case Generation for Primitive Recursive Functions. In Grabowski, J. Nielsen, B. eds.: International Workshop on Formal Approaches to Testing of Software (FATES'04). LNCS 3395, pp. 16–32, 2004.
9. Lüth, C., Karlsen, E.W., Kolyang, Westmeier, S., Wolff, B.: HOL-Z in the UniForM-Workbench – a case study in tool integration for z. In Bowen, J., ed.: 11. International Conference of Z Users ZUM'98. LNCS 1493, pp. 116–134, 1998.
10. Lüth, C., Wolff, B.: Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. Journal of Functional Programming **9**(2), pp. 167– 189, 1999.
11. Brucker, A.D., Wolff, B.: A Verification Approach for Applied System Security. International Journal on Software Tools for Technology Transfer (STTT), DOI 10.1007/s10009-004-0176-3, 16 pages, to appear 2005.

1 Introduction

The development of *tools* for program analysis, verification and refinement is a prerequisite for the proliferation of formal methods in industry and research. Among the plethora of widely-used specification languages underlying formal methods, we name only a few examples here such as Z [65, 41], VDM [38], UML/OCL [54, 70, 71], CSP [9] and Hoare-Logics [72, 50], which are based on more foundational formalisms like First-order Logic, Axiomatic Set Theory (ZF) or Higher-order Logics (HOL) [24, 8, 32].

With respect to the formal *analysis* of specification languages, their representation **by their semantics** in a powerful meta-logic, a so-called *logical framework* such as NuPRL [1], Coq [3] or Isabelle [6, 51], is a widely accepted technique that has been applied in many studies. With respect to the *implementation of tools* for formal methods, however, these representations have been rarely used. Rather, formal method tools are usually *directly implemented* in a programming language (e.g. [7, 4, 5, 61, 29, 43, 36, 47]), which makes their correctness, their correct extensibility and their correct combination with other tools a major concern. Moreover, we see a lot of splintering in the field: due to all these special-purpose languages and special-purpose theorem provers and model checkers, there is the danger that numerous ad-hoc implementations of logical engines remain in a premature state hampering the progress of the field as a whole.

Representing the semantic theory of a specification formalism or a programming language inside a logical framework is called a *logical embedding*. Using an embedding has two main advantages: First, existing theorem prover kernels can be reused. Second, provided that a logical embedding is a *conservative* extension of the logical framework, it is possible to formulate the symbolic computations inside the tool on the basis of *derived rules* over this embedding. Thus, a tool implementation can guarantee its logical consistency (relative to the consistency of the logical framework) as well as its correct implementation of all symbolic computations. However, in order to be practically useful *and* correctness-oriented, the conservative embedding approach must provide:

1. **suitable semantic embeddings** for commonly used formal specification languages. These embeddings should be sufficiently intuitive for experts in order to be checkable against informal specifications, their logical consistency should be mechanically checkable, and deduction should not be hampered by the overhead due to the representation,
2. **mechanical support** of the **method** for the represented formal specification languages. This may include support for refinements of specifications, support for code verification[1], or support for testing methods of code, and
3. **appropriate tool integration** of theorem provers in the tool support of (pragmatic) software development processes. This includes front-ends for software engineers during modeling activities, support tuned for the underlying methods during refinement, code verification, or test, for example.

[1] not investigated here

In this work, we chose the conservative *shallow* embedding technique as instance for the place-holder "suitable" in the list above — in contrast to my Phd-work [74], where deep embeddings were chosen as foundation for a simple logical framework in itself. This choice is motivated by the requirement of mechanical check-ability of consistency and more efficient deduction. Shallow embeddings are known to add two more issues to the list of challenges above:

1. the **application range** of embeddings is believed to be limited to very small and artificially designed languages, using a type system which is very similar to the one of the logical framework, and
2. the **flexibility** of the approach seems to exclude an application when the formal method is still under development, in particular for languages "designed by committee".

It is the ultimate goal of this work to show that these challenges can be met. For this purpose, we present a number of paradigmatic embeddings, modular representation techniques, automated deduction techniques to support specific methods and integration technologies. In more detail, our contributions in these five fields are:

1. suitable embeddings for widely used formal methods, including process-oriented, data-oriented and object-oriented specification methods (CSP, Z, UML/OCL),
2. abstractions and aspect-oriented structuring techniques allowing for the quick development of semantic variants enabling the study consequences of changes in formal methods under development (like UML/OCL),
3. particular techniques for generating library theories, for supporting particular deduction styles in proofs, for specialized deduction support for concrete development methodologies,
4. different scenarios of the integration of the developed tools in conventional tool chains in software engineering, and
5. front-ends for light-weight integration into tool chains (like HOL-Z 2.0) or prototypic encapsulation of logical embeddings into generic graphical user-interfaces for a more comprehensive encapsulation.

The plan of this introduction — mirroring the plan of this work as a whole — is as follows: First, in the following Sec. 2, we explain the *foundations* of this work in more detail. In the subsequent sections, we will describe our contributions in the field of *embeddings* (Sec. 3), *method-support* (Sec. 4) and *integration/encapsulation* (Sec. 5) and put them into perspective. In the final Section 6 we will validate one of the resulting tool chains (including the theorem proving environment) with a complex case study arising from the field of applied computer security. Note that some of the presented papers are discussed in more than one of these sections since they cover several of the mentioned categories of issues.

2 Foundations

As the basis for the subsequent presentation and discussion of own work, we will introduce to some core notions like "logical framework" and our particular choice "Isabelle/HOL", "conservative embeddings" and "shallow embeddings".

2.1 Logical Frameworks and Isabelle

According to the *Logical Frameworks Home Page* maintained by Frank Pfenning ([57], see also for a bibliography and a collection of short surveys), a *logical framework* is a formal meta-language for deductive systems. The primary tasks supported in logical frameworks to varying degrees are

1. specification of deductive systems,
2. search for derivations within deductive systems,
3. tactic programming of deductive algorithms,
4. proving meta-theorems about deductive systems.

As typical systems, logical frameworks like ELAN, Elf, Forum, Isabelle, and lambda Prolog were mentioned. Typically, a logical framework is based on a typed *lambda*-calculus. However, the precise borderline to general purpose higher-order logic provers like HOL, LEGO, NuPRL is not clear-cut; for Pfenning the choice of terminology merely indicates the relative emphasis placed on these tasks. Logical frameworks have been applied to many examples from logic and the theory of programming languages.

2.2 Concepts and Use of Isabelle/HOL

The logical framework Isabelle [51] is a generic theorem prover of the LCF prover family; as such, it offers the possibility to build programs performing symbolic computations over formulae in a logically sound way on top of the logical core engine. Throughout this work, we will use Isabelle/HOL, the instance for Church's higher-order logic [24, 8] extended by parametric polymorphism with order sorted types ([49]). Isabelle/HOL supports conservative extension schemes (see Sec. 2.3), and, derived from these principles, support for data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

A number of theories built by conservative definitions provide the theory for arithmetics for natural and integer numbers, typed set theory based on the type *set(α)* and a list theory based on the type *list(α)*. Moreover, there are products, maps, and even a specification on real numbers and non-standard analysis. The HOL-library provides several thousand derived theorems — yielding the potential for reuse in a specialized tool for a particular formal method.

2.3 Conservative Theory Extensions

In this section, we explain the key concept of "conservative theory extension" which — appropriately supported by Isabelle — yields the technology to build consistent semantic models of the specification languages and finally tools supporting deduction over them.

Slightly simplifying the situation in Isabelle, we define a *theory* as a pair (Σ, Ax) with a signature Σ assigning constant symbols to their types and Ax a set of axioms. A *theory extension* is a theory (Σ', Ax') where $\Sigma \subseteq \Sigma'$ and $Ax \subseteq Ax'$. The idea of *conservative* theory extensions (see [32]) is to provide syntactic characterizations of theory extensions that guarantee that if a theory was consistent, then its extension will also be.

The simplest form of a conservative theory extension is a the *constant definition* $(\Sigma', Ax') = (\Sigma \cup c \mapsto \tau, Ax \cup c \equiv E)$ where c is "fresh" i.e. (not declared in Σ and $c \equiv E$ is a "definitional axiom", i.e. E is a closed expression not containing c. Thus, the conservativity of a constant definition can be justified since it works essentially as abbreviation.

Another form of a conservative theory extension is a the *type definition* where a new type is defined isomorphic to a set of previously defined individuals — generalizing the idea of "abbreviation" to types.

It turns out that the theory libraries and the semantic models of the specification languages treated in this work can be built up entirely conservatively; this also holds for recursive function definitions and recursive data types which are ruled out at first sight by the above extension schemes.

For more details see the introduction of [20].

2.4 Embedding Techniques — An Overview

A theory representing syntax and semantics of a programming or specification language is called an *embedding*. While the underlying techniques are known since the invention of typed λ-calculi, it was not before the late seventies that the overall importance of *higher-order abstract syntax*(HOAS) (a term coined by [30]) for the representation of binding in logical rules and program transformations [31] and for implementations [30] was recognized.

For example, a universal quantifier may be represented in HOAS by a constant $All :: (\alpha \to bool) \to bool$, where the term $All(\lambda x.\, P(x))$ is paraphrased by the usual notation $\forall x.\, P(x)$. In contrast to the usual textbook definition for predicate logic providing a free data-type for terms and predicates, explicit substitution and well-typedness functions over them, and explicit side-conditions in logical rules over quantifiers preventing variable-clashes and variable capture, this representation of the universal quantifier has two advantages:

1. the substitution required by logical rules like $\forall x.\, P(x) \Longrightarrow P(t)$ can be directly implemented by the β-reduction underlying the λ-calculus, and
2. the typing discipline of the typed λ-calculus can be used to represent the typing of the represented language. For example, a multi-sorted first-order

logic (having syntactic categories for arithmetic terms, list terms, etc.) is immediately possible by admitting expressions of type nat and $\alpha list$.

In short, HOAS has the advantage of "internalization" of substitution and typing into the meta-language, which can therefore be handled significantly more general and efficiently.

When using semantic definitions on the basis of HOAS, the technique is extended into a "shallow embedding" (this terminology has been coined in [15]) of an object-language, a technique which is opposed to a "deep embedding" which treats syntax by free datatypes and semantics via semantic interpretation functions as common in logical textbooks.

A shallow embedding definition of the universal quantifier is, for example, $All\, P \equiv (P = \lambda x.true)$ (the propositional function of the "body" of the quantifier must be equal to the function that yields $true$ for any argument), a deep representation follows usual textbooks:

$$Sem[\![\forall x.P(x)]\!]\gamma \equiv true \;\; \texttt{if}\, Sem[\![P(x)]\!]\gamma[x := d]\, \texttt{for all}\, d \\ false\, \texttt{otherwise}$$

where we assume a meta-language with well-defined concepts such as `if` ... `otherwise` and ... `for all` ..., for example ZF-set theory.

As can be seen, shallow embeddings can have a remarkably different flavor in their semantic presentation, in particular when striving for conservativity as in the example above. However, since the usual inference rules were derived from these definitions (like the rule $\forall x.\, P(x) \Longrightarrow P(t)$ from above), they are finally proven as semantically equivalent.

If a shallow embedding is in itself a conservative theory extension (which implies that the language definition is consistent if the meta-language is), we also speak of a *conservative embedding*.

For more details see the introduction of [20].

3 Embeddings of Specification Languages

Since the principle of shallow embeddings in itself is fairly well-known, the question arises, what the fundamental problems and technical challenges exist for representing "real world" specification languages. The main difficulty stems from the fact, that the most successful shallow-embeddings (as, for example, HOL encoded in typed λ-calculus) are *designed* to fit to the underlying meta-language. In contrast, "Real world"- languages are typically conceived solely on a kind of mathematical notation based on naive set theory and no specific experience in theorem proving in mind; in some cases, the definition process of a "real" specification language takes place in a "development by committee" process prone to all sorts of arcane compromises. While a deep embedding is always possible whenever a formal semantics exists, a shallow embedding may conflict with certain constraints the shallow technique imposes, be it on the representability of the binding structure, the type discipline and the semantics of the language.

Thus, even fairly worked out semi-formal semantic descriptions of a specification language may be quite distant to a formal (i.e. machine-checkable) representation, and even a formal semantic representation may be quite distant to a set of derived rules that allow for the formal support of a particular *method* of this language. Providing paradigmatical embeddings and new techniques helpful for bridging this gap is the main goal of this work. In order to describe the problem domain in more detail, we use the following classification: Specification languages may be:

1. **process oriented**, i.e. their focus of description is the *behavior*, usually described in terms of possible sequences (*traces*) of states or communication events,
2. **data oriented**, i.e. their focus of description is the structure of data a system processes, its states, and individual steps the system performs when making a transition from one state to another, and
3. **object-oriented**, which is essentially a data-oriented specification approach where object-oriented data structuring techniques such as inheritance and sub-typing are emphasized.

Examples for process-oriented specification languages are temporal or modal logics (such as LTL, CTL, μ-calculus [25]) or process-algebras (such as CCS[48] or CSP [9]). Examples for data-oriented specification languages are VDM [38] or Z [65], examples for the object-oriented language class are ObjectZ [64] or UML/OCL [54, 70, 71].

In the following subsections, we will discuss the specific technical challenges with respect to binding structure, typing and semantics of typical representatives of these language classes. These subsections serve as overview and summary of the papers [69], [18], [17] and [58] of the first part of this work.

3.1 A Shallow Embedding of CSP in HOL: HOL-CSP

The shallow representation of the syntax of CSP in HOL from the (deep) textbook presentation follows a schema which we call *canonical translation*; for the sake of the completeness of the presentation, we will repeat this construction here. For the following example, we refrain from an explanation of the presented operators (see [9] and [69] for the details) and just focus on the syntactic aspects of the translation.

In the CSP textbook [9], the syntax is given in the form of an extended Backus-Naur-Form (EBNF) and reads as follows:

$$\begin{aligned} P ::= \; & X \mid \mathit{Skip} \mid \mathit{Stop} \mid P \Box P \mid P \sqcap P \\ & \mid a \rightarrow P \mid P; P \mid ?x : A.P \\ & \mid P \setminus A \mid P[\![A]\!]P \mid P||P \mid P|||P \\ & \mid \mu X.F \end{aligned}$$

where some additional explanations are given such as "P is the non-terminal for all process expressions" or "X is the non-terminal for variables, a for events,

and A for event sets". The canonical translation of an EBNF to a shallow-style signature works as follows:

1. expanding the EBNF to a BNF,
2. erase the production for "X" and all other explicit references to variables in the grammar; the latter results in "$? : A.\ P$" and "$\mu.\ F$", (variables were represented by variables of the meta-language HOL),
3. map the non-terminals *events*, *event sets* and P to the types α, $\alpha\ set$ and $\alpha\ process$, where *set* is a type constructor from the HOL-library and *process* is newly introduced type constructor,
4. lift all occurrences of non-terminals in the production rules, where variables were implicitly bound, to functions from the type of these implicitly bound variables; thus, we get: $? : \_.\_ :: [\alpha\ set, \alpha] \to \alpha\ process$ [2] and $\mu.\ \_ :: [\alpha\ process \to \alpha\ process] \to \alpha\ process$ for the two language constructs where HOAS is used. The notation $?x : A.\ P$ is treated as syntactic paraphrasing for $? : \_.\ \_\ A\ (\lambda x.\ P)$ and $\mu x.\ P$ for $\mu.(\lambda x.\ P)$.

The remaining resulting "shallow" signature for CSP looks as follows:

$$
\begin{array}{rl}
\mathit{Skip} & :: \alpha\ process \\
\mathit{Stop} & :: \alpha\ process \\
\_\Box\_ & :: [\alpha\ process, \alpha\ process] \to \alpha\ process \\
\_ \sqcap \_ & :: [\alpha\ process, \alpha\ process] \to \alpha\ process \\
\_ \to \_ & :: [\alpha, \alpha\ process] \to \alpha\ process \\
\_; \_ & :: [\alpha\ process, \alpha\ process] \to \alpha\ process \\
\_ \setminus \_ & :: [\alpha\ process, \alpha\ set] \to \alpha\ process \\
\_[\![\_]\!]\_ & :: [\alpha\ process, \alpha\ set, \alpha\ process] \to \alpha\ process \\
\_||\_ & :: [\alpha\ process, \alpha\ process] \to \alpha\ process \\
\_|||\_ & :: [\alpha\ process, \alpha\ process] \to \alpha\ process
\end{array}
$$

As a consequence of this translation, informal side-constraints such as "all sub-expressions in a process refer to the same set of basic events α" or the binding of variables including their type discipline are made explicit. The gain of this internalization is that we can now represent substitution simply by function application as in rules like $\mu x.P = P(\mu x.P)$ since P is no longer a process, but a function that takes a process as argument and yields a process. With respect to types, there is not much to do: the underlying set of all events Σ (represented by the type variable α above) is considered to be completely unstructured. A similar language like CSP_M used in FDR providing features for data types, pattern matching and functional programming has been developed in HOL-CSP.

While a shallow representation of its syntax and substitution is canonical for CSP, at first sight, the semantics of HOL and the process algebra CSP could not be more different. CSP talks about non-deterministic computations depending on their context, whereas HOL, sometimes referred as "functional language with quantifiers", is based on a deterministic and functional "computation model".

[2] the notation $[\tau_1, \ldots, \tau_n] \to \tau_{n+1}$ is an abbreviation for $\tau_1 \to \ldots \to \tau_n \to \tau_{n+1}$

Recursion in CSP may be undefined ($\mu X.X$ is a legal expression in CSP), whereas in HOL, only some form of well-founded recursion is admitted. However, earlier work [23, 60] had already shown that it is possible to represent Scott's approach to denotational semantics [66, 72] in HOL in a conservative shallow embedding — in other words, HOL is expressive enough to formalize Scott's *Logic of Computable Functions* (LCF) which stood at the beginning of theorem proving environments and logical frameworks. The idea is to represent key concepts like complete partial orders (*CPO*'s), and continuity inside a basic semantic theory. This allows for representing partial functions inside CPO's and explaining unbounded recursion via fix-point combinators. It turns out that tricky syntactic side-conditions such as the *admissibility of predicates* for fix-point induction can be represented semantically in HOL, leaving the reasoning over the admissibility to a side-calculus based on derived rules. Meanwhile, a fully fledged conservative shallow embedding of LCF in HOL — called HOLCF [52] — based on the work of [60] is part of the Isabelle distribution.

Our work [69] extends this line by formally showing that processes in the sense of the semantic definitions in [9] represent CPO's and can be captured by a type definition forming α *process*. This construction revealed an error for one of the core operators in previous CSP literature which could be communicated to Bill Roscoe before his book went to print. Moreover, the tricky details concerning the continuity of the operators could be formally shown, together with a derived side-calculus that allows for the deduction of the admissibility of process-refinements (in the sense of [9]). This has the consequence that the fix-point-induction principle is applicable for refinement proofs over processes. However, no tactic support for automated refinement proofs (based on fix-point induction) has been developed so far, which would be highly desirable for larger case studies.

3.2 A Shallow Embedding of Z in HOL: HOL-Z

Representing the syntax of Z terms following the "canonical translation" technique presented in detail in the previous section is a standard exercise. However, the binding structure of some expressions, namely the expressions of the so-called schema-calculus, is highly non-standard with respect to binding: according to the type of certain expressions, implicit bindings to components of underlying schemas were produced. In Z, for example, it is possible write:

$$\forall A.B \qquad or \qquad A \wedge B$$

where A and B are schema expressions, which means semantically sets of records. If A has the record components a and b, and B the components a, b, c, then $\forall A.B$ is again a schema with the component c, i.e. the components a and b are bound as a consequence of implicit conventions based on types. Moreover, schemas may play different roles in an expression: They may serve as sets, or as predicates as above. According to their role, explicit coercions must be generated.

Besides the necessity to develop the library of Z — the so-called "Mathematical Toolkit" — conservatively in HOL, the reconstruction of this implicit

bindings is the main technical achievement in HOL-Z [18]. The problem is solved by an independent type-checker that produces additional syntactic information and combinators that capture the essence of bindings and translation. The effects on these representational "tricks" on deduction, and the development of support for specific methods such as system development by formal refinement or testing is discussed in Sec.4.

3.3 Shallow Embedding of UML/OCL

A shallow embedding of the key features of object-oriented modeling in HOL is the technically most challenging enterprise in this work - but also the one with the greatest potential of use in software engineering. A shallow embedding in HOL must capture the essence of sub-typing and inheritance in a type system that does not provide a notion of subtypes, and provide means for extensibility and reuse in order to capture the pragmatics of object orientation. Moreover, a formalization in HOL must capture that object-oriented specification as in UML/OCL is deeply intertwined with the notion of state: A substantial part of the language is concerned with the map of object-references to object-instances (abstractions of pieces of memory) and logical constraints on this map expressed in form of class invariants or pre- and postconditions of methods. A consequence of our specific choice UML/OCL for an object-oriented specification language is the necessity to cope with role of undefinedness in computations and in the logic.

The representation of syntax of UML/OCL follows the canonical translation technique described in Sec. 3.1, but with a slight modification: Since all arguments of operations may refer to an environment γ (a data structure required by the UML/OCL standard containing the states before and after the execution of methods), an OCL operation as, for example, the addition $\_ + \_$ on $INTEGER$ in the sense of the standard must be lifted over these environments, i.e. the corresponding operation $\_ +' \_$ in HOLOCL must have the type $(\gamma \to INTEGER) \to (\gamma \to INTEGER) \to \gamma \to INTEGER$. When introducing the type synonym $Integer_\gamma$ for $\gamma \to INTEGER$ this type can be rephrased as $Integer_\gamma \to Integer_\gamma \to Integer_\gamma$. Thus, a type in OCL can be mapped to a type eypression (instead of a type constructor) in HOL when representing OCL operations; we call a translation of this type a *semi-canonical translation over* $\gamma_1,\ldots,\gamma_n$. It turns out that the complete set of UML/OCL operators, be it from the core logic or be it from the quite rich library, can be syntactically constructed as semi-canonical translations. This includes operators with non-trivial binding-structure such as the *let*-construct, the so-called *iterators* (corresponding to *map*-like and *fold*-like operators known from functional languages).

Beyond the typical problems concerning undefinedness and states, the main achievement of HOL-OCL as a shallow embedding consists in a particular *semantic* understanding of the key-notions such as static and dynamic types in object-orientation, and combining this with *practical* need of extensibility of the object-type system and modular theories over it. In the end, the theory also offers a new perspective on methods and late-binding method invocation.

3.4 Encoding System-oriented Libraries

Capturing the semantics of specification and programming languages in itself is only part of the problem. Beyond libraries considered as part of the language (such as the "Mathematical Toolkit" of Z or the description of the "Collections" in UML/OCL), a significant part of the work that makes formal methods practically relevant consists in the modelling of the underlying system environment — be it arithmetic data types as implemented by the underlying processor or be it interfaces of the underlying operating system.

In this work, one example for each of these situations is presented, by providing new theories covering these aspects.

In [58], we adress the problem of arithmetic data types as specified by the Java Language Specification [33], or, in other words, the machine arithmetics of the Java Virtual Machine. With respect to floating point numbers, similar work had been undertaken earlier (as in the seminal work by John Harrison [35]). The Integral Numbers in the sense of the Java Language Specification are similar to the ANSII-C specification, and similar to most processor specifications, but more specific with respect to the division and remainder operations. Machine integral numbers behave in many respects similar to their "mathematical" counterparts: they enjoy ring properties (even if exceptions are taken into account when the law $a - a = 0$ is constrained to non-exceptions) and many other simplification laws. However, in some details, they behave remarquably different (the successor of the maximal number is the minimal number) which results in complex cancellation rules ($a \leq b \Longrightarrow a + c \leq b + c$ only holds for non-trivial side-conditions) and, moreover, sometimes in bad surprises which are very likely to be overlooked by programmers (for example, $abs(x)$ is not necessarily positive). While floating point numbers are meanwhile a fairly well-investigated subject, machine integral numbers as used in the Java Language Specification have attracted the interest of researchers only recently: they form the basis for the verification of arithmetic libraries and libraries of cryptographic functions, which are gaining more importance.

In [22], we address the modelling problem of a widely used operating system library, namely UNIX/POSIX, with respect to the access control facilities for files in a UNIX filesystem. This model is used to verify an access control policy by a sytem implementation model. More details on this model and the verification method is given in Sec. 6.

3.5 Summary

In order to summarize the presented logical embeddings, we define several notions for their classifification. As already mentioned, we call a syntax translation:

- *canonical* iff the non-terminals (or types) of the object language can be mapped to type constructors in the HOL representation,
- *semi-canonical* iff iff the non-terminals (or types) of the object language can be mapped to type expressions (possibly containing free type variables),

– *pre-compiled* iff the non-terminals (or types) of the object language were mapped to type expressions according to their context in the term-language.

As an example for a pre-compiled translation, consider the HOL-Z embedding, where a schema expression A may be mapped to a function $\tau_1, \ldots, \tau_n \to \mathit{bool}$, where both the τ_i and the n depend on previous declarations.

We turn now to a classification of type systems resulting from an embedding function $\texttt{embedding} : L \to HOL$ mapping expressions of an object-language L to expressions in HOL. We assume the existence of predicates $\texttt{well}-\texttt{typed}_{\mathrm{L}}$ (assuring that an expression of language L is well-typed w.r.t. type discipline) and $\texttt{well}-\texttt{typed}_{HOL}$ (assuring that a HOL-expression is well-typed with respect to the simple-type type system including parametric polymorphism called λ^{α}.

Then we can characterize a type translation underlying an embedding as

– *approximate* iff for any e, $\texttt{well}-\texttt{typed}_{HOL}(\texttt{embedding}\, e)$ is implied by $\texttt{well}-\texttt{typed}_{L}(e)$,
– *tight* iff the type translation is approximative and we have additionally for any e, $\texttt{well}-\texttt{typed}_{L}(e)$ is implied by $\texttt{well}-\texttt{typed}_{HOL}(\texttt{embedding}\, e)$.

Thus, if an embedding is tight, a possible implementation for a type-checker consists in applying the `embedding`-function and trying to type-check its result in HOL; an error on the converted term indicates the existence of a type error on the original term. Note, however, that the practical usability of such a conceptual implementation is usually very limited, in particular for semi-canonical or precompiled languages, since the error messages gained by this process are hard to decipher for users not familiar with the semantic details of the embedding.

On this basis, we can summarize the discussed embeddings in the following table:

	Process-Oriented	Data-Oriented	Object-Oriented
Example	CSP	Z	UML-OCL
Syntax	Canonical Translation	pre-compiled HOAS	Semi-Canonical
Typing	Tight	Approximate	Tight
Semantics	Denotational Semantics, Complex Side-Calculi	Fairly trivial	Theory Morphisms, Complex Side-Calculi, Three-Valued Logics

The reason for the non-tightness of the Z embedding lies in a mere technicality: in the Z type-system, n-ary tuples or records exists and are distinct to their (isomorphic) represenation as nested pairs: (a, b, c) has not the same type as $(a, (b, c))$. Since the latter is the representation of the former in HOL-Z, an expression like $(a, b, c) = (a, (b, c))$ can be typed after applying the embedding function although it can not be typed with the Z type discipline.

Some explanation for the remarks in the line summarizing semantic features: in a shallow embedding, it is necessary to express syntactic constraints (e.g. for admissibility as in the case of fixpoint-induction, or continuity in the case of

process-refinement, or state-passing in the case of rewriting in OCL) *semantically*, i.e. by second or third-order predicates expressing conditions that were otherwise characterized by inductively defined subsets of the syntax. These semantic predicates result in side-conditions which can be established by derived rules having a similar form and purpose than the inductive rules of a sublanguage definition (for example, the admissibiliy rules characterize formulas built over logical constants, $\_ \wedge \_$, $\_ \vee \_$ and universal quantification; thus, existential quantification or negation are ruled out except that they occur in the construction of constant terms). However, their semantic construction leads sometimes to unexpected generalizations and often improves the insight into their logical nature than their (traditional) syntactic counterparts.

4 Support for Specific Methods

The previous section was predominantly concerned with the foundation of tool construction in our approach, i.e. the semantic representation of specification or programming languages. In this section, we are concerned with the processing of specifications or programs based on rules derived from this foundation. We distinguish three categories:

- *specific deduction*: this covers calculi geared towards interactive or automated deduction in specifications. Examples for the former are particular, refinement oriented proof presentation techniques, examples for the latter are tableaux calculi for a logic such as OCL.
- *theory development support*: this comprises systematic approaches for theory construction or support for theory development. Examples are our technique of theory construction based on a theory morphism as well as refinement-oriented proof support.
- *methodological support*: this comprises support for specific methods such as system development by formal refinement and test-case generation.

4.1 Specific Deduction

Derived Transformational Calculi. A straight-forward approach of program development is the "program development by transformation"-approach following systems like PROSPECTRA or KIDS [63] The main idea is to represent well-known algorithmic schemes such as "divide and conquer" or "global search" (c.f. [27]) as built-in rules of the form:

$$A \Longrightarrow Spec \sqsubseteq Prog$$

where A is the applicability condition of the transformation rule, $Spec$ the specification pattern and $Prog$ the resulting program pattern and $\sqsubseteq$ the refinement relation that states the logical relation between $Prog$ and $Spec$ (equality in the simplest case). These rules were directly supported by a formal program development system that supports the stepwise transformation from an abstract

specification to a more concrete one that can be automatically converted into executable code.

The contribution in our paper [40] [3] consists in the demonstration that these program transformation rules can be derived from induction principles inside *Higher-Order Logic (HOL)*[8] or the *Logic of Computable Functions (LCF)* going back to the work by Scott and Milner. Moreover, it could be shown that these derived rules can be used to *implement* these program development systems in a logically sound way by encapsulating LCF-style theorem prover engines into a *tool* by providing control via method-specific tactical programs (called "tactic sugar") and a user-interface that hides the technical details of the engine away from the user. This paper is methodologically and technically the starting point of this work.

Derived Tableaux-Calculi. Shallow embeddings may introduce new combinators that cope with certain aspects of an embedded language or logics that have a substantially different flavor from the used meta-logics HOL. This results in particular challenges for proof-support of an object-logic, since expanding the constructs into their definition in the meta-logic and proving the formulas there is often computationally infeasible or results in formulas that are difficult to interpret by a user and therefore unsuited in an interactive setting.

In [18], we have developed a tableaux calculus for a sub-language of Z with a noteworthy non-standard treatment of binding called the "schema calculus" (see Sec. 3.2). The binding structure is made explicit in HOL-Z by special combinators which must be generated by a pre-compiler; for example, the shallow representation of the schema conjunction can be presented as:

$$SB"z_1" \longrightarrow z_1 \ldots "z_p" \longrightarrow z_p.\, A(x_1 \ldots x_n) \wedge B(y_1 \ldots y_m)$$

Instead of unfolding combinators like SB and converting a Z-formula into a potentially large representation in HOL, we developed a special tableaux calculus that maintains the structure of a Z-formula by assigning to each schema language construct special tactics mimicking introduction and elimination rules. Thus, a mechanical proof can follow the Z-structure of a formula and is not forced to reason over the (less abstract) HOL-formula. Formulas resulting from such introduction or elimination tactics can still be pretty-printed and interpreted as Z formulas by the user which is vital for interactive proof.

In [17], we have to cope with a three-valued logics of OCL required by the OCL Standard [55]. As a consequence, automatic proof support by our proof environment Isabelle can not be reused on OCL formulas. Consequently, we derived an own tableaux calculus (roughly following previous work by Kohlhase and Hähnle, but based on derived rules and specific side-calculi treating expressions consisting of strict operations or other semantic side conditions such as "state passing").

[3] David: We should discuss your objections here muendlich.

4.2 Theory Development Support

The mentioned *theory morphisms* refer to the work [20], which shows that the semantic presentation via a set of combinators leads not only to a very modular organization of the semantic theory of the object language (which may be under development and is therefore a moving target for analysis and tool development), but also to techniques to *generate* a significant part of the semantic library, i.e. the necessary body of rules derived from the semantic theory of a language in order to be useful for applications.

4.3 Methodological Support

Derived Window-Inference Calculi. Transformational program development is a fairly coarse-grained approach, and transformational program development calculi in the original sense were deliberately incomplete. This lead to a number of theoretical and technical shortcomings. In practice, many syntactic "massage steps" are necessary in order to bring the original specification into a form that makes a rule "applicable". Applicability conditions has been traditionally considered as formulas to be proven by "external" tools which raises the question of correct tool integration. In order to gain more flexibility, in many transformation systems "invent and verify"-rules following the (obviously sound) scheme:

$$Spec \sqsubseteq Prog \Longrightarrow Spec \sqsubseteq Prog$$

which instantiates the applicability condition A with $Spec \sqsubseteq Prog$ and therefore shifts the burden of the correctness-proof to the applicability condition and thus entirely to external reasoning.

In order to increase the flexibility of the approach, a more general framework for transformational program development (introduced by [62], investigated by [34] and combined with Back's refinement calculus by [10]) was used as foundation for our work on *generic* transformation systems [46], which represents a re-implementation and generalization of [40]. The basic idea of the new generalized framework is the concept of a *window rule* of the form:

$$adm(C) \Longrightarrow (A \Longrightarrow Spec \sqsubseteq_{(j)} Prog) \Longrightarrow C(Spec) \sqsubseteq_{(i)} C(Prog)$$

where C is called the "focus", $Spec$ the "window" containing the specification to be refined to a program $Prog$, and a family of partial orderings $\sqsubseteq_{(i)}$, called refinement orderings. Such refinement orderings may be all sorts of program refinement notions, or just logical proof refinement, i.e. implication, which makes it possible that not only sub-windows, but also applicability conditions A may be treated with a possibly complete calculus (as existing for HOL). A focus is constrained to an applicability condition $admC$, which can enforce, for example, that in logically negative contexts proof refinement is realized by implication in the opposite direction. Note that the proof of the special applicability condition $adm(C)$ is automated and hidden from the user.

Our contribution in [46] is the implementation of generic automated tactical support of window inference reasoning in Isabelle (including monotonicity reasoning and management of the refinement orderings), its integration into a newly developed "point-and-click" generic graphical user interface geared to support window inference calculi and a resulting *hierarchically structured* (refinement) proof presentation, which we believe is more suited for software engineers and novice formal method users than, say, natural deduction. The generic implementation has been instantiated with HOL and proof refinement, CSP and process refinement and Back's Refinement Calculus for data-refinement in imperative imperative programs.

Support for Post-hoc Refinement Methods. Instead of transforming a specification into its refined version, as in the transformational approach, it is of course also possible to *state* that some function or procedure is refined by another. From such a statement, proof-obligations can be generated that allow for a post-hoc verification of the postulated refinement relation. It is then possible to verify these proof obligations.

We applied standard refinement notions such as "forward simulation" in both the HOL-Z [18] as well as the HOL-OCL [17] environment. The method implemented in HOL-Z had been applied in substantial case studies (see Sec.6). In the case of object oriented specification as in HOL-OCL, our proof system is even general enough to refine methods by each other which refer to completely independent data universes, i.e. they may be defined in different class diagrams.

Test-Case Generation. In [21] we present a completely different line of exploiting formal specifications besides code verification and refinement: Generating (unit-)tests out of a specification given by pre- and postconditions. The presented setting is geared towards functional programs.

While the former two validation techniques are motivated by the question "are we building the program right?", the latter is focused on the question "are we specifying the right program?". In particular, if a formal model of the environment of a software system (e.g., based on, amongst other things, the operating system, middle-ware or external libraries) must be reverse-engineered, testing — in the sense of "experimenting" — is a necessity (see [19, 22]).

Our method has two stages: first, the original formula is partitioned into test cases by transformation into a Horn-clause normal form (HCNF). Second, the test cases are analyzed for instances with constant terms satisfying the premises of the clauses. Particular emphasis is put on making test hypotheses underlying a test explicit and on using test hierarchies to avoid intractability.

5 Embedding Encapsulation and Tool Integration

One major obstacle for the use of formal methods in an industrial setting is the difficulty of integrating it in the conventional software engineering process.

Developers of IT-technology usually have their own suite of tools capturing all phases of software engineering ranging from requirement analysis, design phase, coding, validation and deployment. During the development, all documents produced and exchanged between these phases underly a constant flux of changes. The management of these changes and the achievement of suitable degrees of consistency for the documents of the overall process is a major problem (which is often enough unsolved in practice).

Integrating formal specifications in requirement, design and code documents offers both new potentials as well of new technological challenges. We consider as new potentials that the effects of changes can be traced mechanically and that the correctness of code with respect to the original specifications can be validated. The most notable challenges is the increasing amount of information to be processed, the increasing number of tools and technologies involved as well as the increase of the overall bureaucracy.

Two major tool integration schemes can be identified in software engineering:

1. the tool-chain,
2. the repository/plug-in architecture

Characteristic for a tool-chain is the fairly coarse granularity of data that it processes; usually, input files and local files were passed to a tool, which produces intermediate files which are then fed into the next tool of the chain and so forth. The chain proceeds in a linear manner, with the consequence that errors occurring in the later stages of the chain have to be interpreted in terms of the input files of the overall chain and corrected there. The dependency of the documents to be exploited by a build management, which can yield work-flow control and also some coarse-grained process optimization. A version control system may add the ability of reconstruct each single stage of the development. Tool control may also be integrated in the editing/viewing environment, as well as specialized views of error reports resulting from running members of the tool chains.

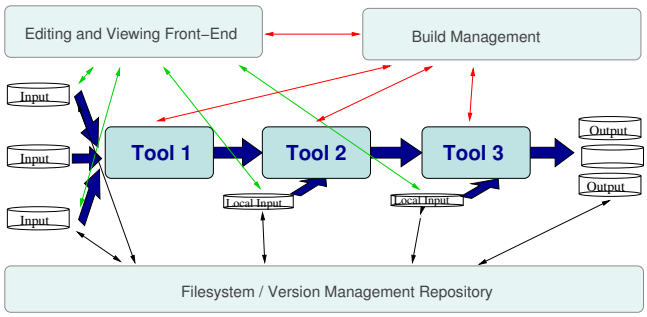


Figure 1. Toolchain Architecture Schema.

The main advantages of this integration scenario is its wide use in UNIX environments, its flexibility and the low integration efforts necessary. Its main drawback is the coarse grainedness of data (which tends to increase the turn-round times of a user) and the assumed underlying linear information flow (which implies not only difficulties for error-handling as mentioned, but also hampers the systematic reuse of intermediate results of tool chain members).

In order to overcome the limitations of tool-chains, plug-in architectures have been provided, which can be seen as variant of component frameworks (Cf. see [68], CORBA[56] or EJB [67]). Prominent examples of plug-in architectures are Rational/Rose [37] and its open-source equivalents Argo/UML [26] and Eclipse [28](for program development). Plug-in architectures provide a uniform data-exchange format and a bus, over which not only files can be exchanged between the tools, but also more fine-grained objects, containing increments of documents, events or intermediate results of integrated tools.

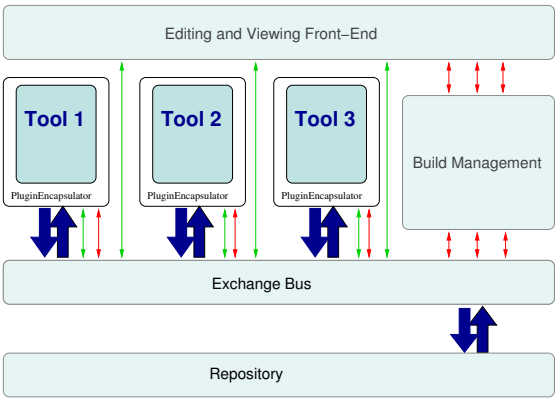


Figure 2. Plugin-Architecture Schema.

After first naive approaches to integrate formal methods into industrial software engineering processes, which resulted in insular specifications and code verifications that had little to do with the delivered product, the tool integration aspect has been addressed by several research projects more seriously, namely UniForM [16] and PROSPER [2], but also various attempts to integrate specific formal methods as plug-ins into plug-in-architectures such as [14, 26, 73].

The work presented here grew out of the UniForM project, but was completed long after the project was finished. Essentially, we followed both approaches: the HOL-Z 2.0 environment is a classical tool-chain, which could be applied successfully for applications, while GenWin and the more comprehensive UniForM-Workbench are (prototypes for) plug-in-architectures.

5.1 The HOL-Z Tool-chain.

The HOL-Z [18] environment follows the tool-chain-architecture paradigm. Using the shallow embedding of Z in HOL described in Sec. 3.2 as the core of the overall architecture, the front-end deficiencies of the shallow embeddings (i.e. syntax and type errors are difficult to interpret on the level of its semantic representation in HOL) are addressed by an own type-checker for a TeX-based input format integrated into a widely used editor (Emacs). This paves the way for a *central document* design which contains all input data, be it informal descriptions, unchecked specifications, machine-checked formal specifications, proofs, code, etc. With respect to Fig.1 Several extractors are used to generate the subcontent of the central document which is passed to the elements of the tool-chain like type-checker and theorem prover. The integration of the type-checker into the editor is close enough to allow for immediate error-messages referring to the type-setting level.

As a result, the gap has been closed between a logical embedding which is proven correct and a *tool* suited for applications of non-trivial size, as is shown in Sec. 6.

5.2 The GenWin-Architecture.

The GenWin-Architecture is a plug-in architecture described in [45]; it provides a generic mechanism to describe plug-ins as *applications* by their signature. On this basis, the GenWin-Architecture generates "the rest", i.e.

- a graphical user interface providing various views objects,
- a hierarchical desktop and object manipulation based on the "drag-and-drop"-metaphor and "point-and-click" into object views,
- a repository,
- and a session management.

The GenWin-Framework has been instantiated with two plug-in instantiations, namely an interface to the theorem prover Isabelle (called: IsaWin) and a specific extension of Isabelle, namely the transformation system TAS [46] described in Sec. 4.3. The integration allows for the logically sound export and import of proof obligations between the different plug-ins; for example, it is possible to perform standard transformational program development according the "divide-and-conquer"-scheme in TAS, split-off resulting side-conditions (such as termination of certain auxiliary functions) as proof-obligations, shift them via drag-and-drop into the IsaWin-plug-in, prove them there, and use the proof finally completed under IsaWin in order discharge the side-conditions still open under TAS. Thus, various views on proof-styles and various degrees of development abstractions can be supported within one framework without compromising logical consistency, since the underlying logical engine is identical as well as the underlying proof contexts.

5.3 The UniForM-Workbench.

The (prototypical) implementation of GenWin is based on a collection of parametric data structures in SML, i.e. plug-ins were described as SML functors. This has as consequence that the framework is essentially limited to SML implementations and inherits SML's limitations with respect to concurrency and exceptions.

The UniForM-Workbench [16, 39] has been designed to overcome these limitations; it is implemented in HASKELL and provides a "real" object exchange bus conceptually similar to CORBA; it provides excellent mechanisms to handle also exceptional behavior and global session management in a repository of a standard versioning system (CVS). In [44], we describe by a paradigmatic example how the necessary data-modeling can be done in order to integrate IsaWin-based Systems (be it IsaWin-like or TAS-like plug-ins based on the logical embeddings described in Sec.3.1, Sec.3.2 or Sec.3.3) into the more general UniForM-Workbench plug-in architecture.

6 Validation through a Case-Study

Embedding languages and developing support for specific methods as such does not suffice to establish our claim that shallow embeddings can be used as a technique for constructing correct tools supporting standard formal methods. Of course, there are successful, tool-oriented shallow embeddings (such as HOL itself which is embedded into Isabelle's built-in login Pure), but HOL and Pure are very closely related and historically, Pure was designed with hindsight to support HOL. With languages such as CSP, Z and OCL, the situation is different: These languages had been designed independently from a concrete proof-environment. On the other hand, a number of *deep* embeddings have been provided [53, 59, 13]. The presentations mirror rather directly formal textbook semantics which does obviously not represent a substantial problem. However, all these embeddings have not proven successful as a tool. For instance, in the case of NanoJava [53] — although used to prove important meta-theoretic properties such as completeness of a derived Hoare-Logics — , the largest published example that has been analyzed is a Java program of about 20 lines of code. In the other cited cases, the situation is not essentially different. Therefore, it is fair to classify these works as (usually deliberate!) proofs of concept and not as proofs of technology.

It is difficult to compare the effectiveness of embedding techniques with respect to their potential to implement proven correct tools for their object-language. A quantification based on costs for inferences in the object language is clearly possible (and speaks in favor of shallow embeddings), but is certainly not a sufficient condition for a proof of technology.

Therefore, we provided substantial case-studies stemming from the application field of computer security [22].

In this work, we present a method for the security analysis of realistic models over off-the-shelf systems and their configuration by formal, machine-checked

proofs. The presentation follows a large case study based on a formal security analysis of a CVS-Server architecture. The formalization of the architectural composition of this system heavily exploits the Z schema calculus (see Sec. 3.2), the resulting proofs therefore vitally depend on deduction support for these constructs (see Sec. 4.1).

The analysis is based on an abstract architecture (enforcing a role-based access control), which is refined to an implementation architecture (based on the usual discretionary access control provided by the POSIX environment). Both architectures serve as a skeleton to formulate access control and confidentiality properties.

Both the abstract and the implementation architecture are specified in the language Z. Based on a logical embedding of Z into Isabelle/HOL, we provide formal, machine-checked proofs for consistency properties of the specification, for the correctness of the refinement, and for security properties.

Meanwhile, another case study has been realized on the HOL-Z environment [11]. Interestingly, this case-study was in direct competition to previous model-checking based verification [12]; it turned out, that the overall time in performing this case-study in the HOL-Z approach (including both modeling and verification time) was not significantly longer as in the model-checking approach, while achieving substantially stronger results. This case-study shows convincingly that HOL-Z can be used for a wide spectrum of modeling problems, ranging from data modeling over behavioral modeling up to the verification of temporal system requirements.

7 Conclusion

We have presented a wide spectrum of techniques to make the embedding approach (in particular the approach based on shallow embeddings) applicable for the correct construction of tools for widely-used and standardized formal methods.

While the shallow embedding technique has been successfully applied for this purpose before (as in the Isabelle/HOL environment itself), we argue that there is a conceptual gap between embeddings of this type and standardized formal method languages that have not been designed with hindsight to a particular proof-environment. To overcome this gap is the overall goal of this work.

The gap consists in a threefold challenge:

1. Find a suitable, machine-oriented representation of the semantics of the object language. We strongly argue in favor of the shallow representation technique for the purpose of tool-construction and provide new techniques to overcome the resulting problems of this approach.
2. Provide mechanical support for the tool-construction as well as the particular *method* underlying a "formal method" (post-hoc verification, transformational refinement, post-hoc refinement, testing). We develop new proof-support (based on hand-programmed tactics controlling applications of derived rules) for various methods based on our representations.

3. Provide encapsulation techniques of prover engines extended by embeddings. This integration scenarios of different layers of abstraction and perfection (such as the tool-chain scenario of HOL-Z contrasted to the integrated GUI-scenario of TAS and IsaWin), for which we developed novel generic techniques and implementations of tools or prototypes.

Further, we provide evidence for at least one of our implementations, that the results are not only a proof-of-concept, but a proof-of-technology. For this purpose, we applied HOL-Z for two substantial case-studies stemming from the field of applied computer security.

One final remark, and the reader may apologize for its boldness: In our view, formal methods and its use for modeling techniques lie at the heart of computer science as a field, similar to informal mathematical modeling representing the "core" of physics ([42]). The work presented here gives further evidence for the thesis that there is in fact "unity behind the diversity" of techniques, tools, and methods, and that the use of this "unity" is not only of merely academical interest, but results in concrete technologies that will find their way into the industrial practice some day.

References

[1] NUPRL Project Home Page, Sept. `http://www.nuprl.org`.

[2] PROSPER Project Home Page: Proof and Specification Assisted Design Environments. `http://www.dcs.gla.ac.uk/prosper/`.

[3] The Coq proof assistant, May 2002. `http://pauillac.inria.fr/coq/`.

[4] ERGO: An Interactive Theorem Prover, Sept. 2002. `http://svrc.it.uq.edu.au/pages/Ergo.html`.

[5] Karsruhe Interactive Verifier Home Page, Sept. 2002. `http://i11www.ira.uka.de/~kiv/`.

[6] The Isabelle Home Page, Sept. 2002. `http://isabelle.in.tum.de`.

[7] The Z/EVES Home Page, Sept. 2002. `http://www.ora.on.ca/z-eves/welcome.html`.

[8] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, May 1986.

[9] A.W.Roscoe. *The Theory and Pactice of Concurrency*. Prentice Hall, 1997.

[10] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer Verlag, 1998.

[11] D. Basin, H. Kuruma, K. Takaragi, and B. Wolff. Verification of a signature architecture with HOL-Z. In *Formal Methods 2005*, volume 3582 of *LNCS*, pages 269–285. Springer Verlag, 2005.

[12] D. Basin, K. Miyazaki, and K. Takaragi. A formal analysis of a digital signature architecture. In S. Jajodia and L. Strous, editors, *Integrity and Internal Control in Information Systems, IV*, pages 31–48. Kluwer Academic Publishers, 2004.

[13] J. Bohn and W. Janssen. A strategic approach to transformational design. In *Industrial Benefit and Advances in Formal Methods (FME'96)*, volume 1051 of *LNCS*, pages 609–628, 1996.

[14] Borland Corp., USA. Together/J System Overview. `http://www.borland.com/together/`.

[15] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.
[16] Bremen Institute of Safe Systems (BISS). Universal formal methods workbench home page. `http://www.informatik.uni-bremen.de/uniform/`.
[17] A. Brucker and B. Wolff. UML/OCL – semantics, calculi, and applications in refinement and test. *Acta Informatica, conditionally accepted, Manuscript 0204*, 2003.
[18] A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, Feb. 2003.
[19] A. D. Brucker and B. Wolff. Testing distributed component bases systems using UML/OCL. In K. Bauknecht, W. Brauer, and T. Mück, editors, *Informatik 2001*, Tagungsband der GI/ÖCG Jahrestagung, pages 608–614. 2001.
[20] A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In H. Geuvers and F. Wiedijk, editors, *Types 2002, Proceedings of the workshop Types for Proof and Programs*, volume 2646 of *LNCS*, pages 59–77. Springer Verlag, Nijmegen, 2003.
[21] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, volume 3395 of *LNCS*, pages 16–32. Springer Verlag, Linz, 2005.
[22] A. D. Brucker and B. Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005.
[23] A. J. Camillieri. A higher order logic mechanization of the csp failure- divergence semantics. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990.*, Workshops in Computing. Springer Verlag, 1991.
[24] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
[25] E. M. Clarke, O. Grumberg, and P. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
[26] T. Community. The Argo/UML project home page. `http://argouml.tigris.org/`.
[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition edition, 2002.
[28] Eclipse.org. The Eclipse.org home page. `http://www.eclipse.org/`.
[29] J.-C. Filliatre. The why tool home page. `http://why.lri.fr/index.en.html`.
[30] C. E. Frank Pfenning. Higher-order abstract syntax. In *PLDI 1988*, pages 199–208, 1988.
[31] B. L. G. Huet. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.
[32] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge Press, July 1993.
[33] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.
[34] J. Grundy. Transformational hierarchical reasoning. *Computer Journal*, 39:291–302, 1996.
[35] J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thry, editors, *12th International*

Conference on Theorem Proving in Higher Order Logics, Nice, France, volume 1690 of *LNCS*, pages 113–130. Springer Verlag, 1999.

[36] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[37] IBM Corp. Rational/Rose. `http://www-306.ibm.com/software/awdtools/deve-loper/rosexde/`.

[38] C. B. Jones. *Systematic Software Development using VDM*. ISBN: 0-13-880733-7. Prentice Hall International, 1990.

[39] E. W. Karlsen. *Tool Integration in Functional Programming Language*. PhD thesis, Universität Bremen, 1998. `http://www.informatik.uni-bremen.de/uniform/wb/papers/ewk-thesis.ps.gz`.

[40] Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementation of transformation systems. In M.-C. Gaudel and J. Woodcock, editors, *FME 96 — Industrial Benefits and Advances in Formal Methods*, number 1051 in LNCS, pages 629–648. Springer Verlag, 1996.

[41] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs*, volume 1125 of *LNCS*. Springer Verlag, 1996.

[42] I. Lakatos, J. W. (Editor), and G. C. (Editor). *The Methodology of Scientific Research Programmes*, volume Volume 1 : Philosophical Papers (Philosophical Papers). Cambridge University Press, 1978.

[43] F. S. E. Ltd. Failures-divergence refinement – FDR2 user manual. Available at `http://www.formal.demon.co.uk/FDR2.html`.

[44] C. Lüth, E. W. Karlsen, Kolyang, S. Westmeier, and B. Wolff. HOL-Z in the UniForM-Workbench – a case study in tool integration for z. In J. Bowen, editor, *11. International Conference of Z Users ZUM'98*, volume 1493 of *LNCS*, pages 116–134. Springer Verlag, 1998.

[45] C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167–189, Mar. 1999.

[46] C. Lüth and B. Wolff. TAS — a generic window inference system. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 405–422. Springer Verlag, 2000.

[47] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*, pages 63–77. Springer Verlag, 2000.

[48] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[49] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[50] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[51] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.

[52] D. V. Oheimb, O. Muller, O. Slotosch, and T. Nipkow. Holcf = hol + lcf, Dec. 1998.

[53] D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer Verlag, 2002.

[54] OMG. Object Constraint Language Specification. [55], chapter 6.
[55] OMG. *Unified Modeling Language Specification (Version 1.4)*. 2001.
[56] T. O. M. G. (OMG). The omg's corba website. `http://www.corba.org/`.
[57] F. Pfenning. Logical frameworks home page. `http://-www2.cmu.edu/afs/cs/user/fp/www/lfs.html`.
[58] N. Rauch and B. Wolff. Formalizing java's two's-complement integral type in isabelle/hol. In *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier Science Publishers, 2003.
[59] R. Reetz. Deep Embedding VHDL. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 277–292. Springer Verlag, Sept. 1995.
[60] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.
[61] W. Reif, G. Schellhorn, and K. Stenzel. Proving system correctness with KIV. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 859– 862. Springer Verlag, 1997.
[62] P. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3:47–61, 1993.
[63] D. R. Smith. KIDS — a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024– 1043, 1991.
[64] G. Smith. *An object-oriented approach to formal specification*. PhD thesis, University of Queensland, 1992.
[65] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
[66] J. Stoy. *Denotational Semantics*. The MIT Press, Cambridge, Mass, 1977.
[67] Sun microsystems. J2EE: Enterprise JavaBeans Technology. `http://java.sun.com/products/ejb/`.
[68] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. The Component Software Series. Addison Wesley Professional, 2nd edition edition, 2002.
[69] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *FME 97*, volume 1313 of *LNCS*, pages 318–337. Springer Verlag, 1997.
[70] J. Warmer and A. Kleppe. *The Object Contraint Language: Precise Modelling with UML*. Addison-Wesley, 1999.
[71] J. Warmer, A. Kleppe, T. Clark, A. Ivner, J. Högström, M. Gogolla, M. Richters, H. Hussmann, S. Zschaler, S. Johnston, D. S. Frankel, and C. Bock. Response to the UML 2.0 OCL RfP. Technical report, 2001.
[72] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.
[73] D. Winterstein, D. Aspinall, and C. Lüth. Proof General/Eclipse. In *User Interfaces for Theorem Provers UITP'05*, Apr. 2005.
[74] B. Wolff. *A Generic Calculus of Transformations*. PhD thesis, Universität Bremen, Shaker-Verlag, BISS Monograph No. 4, 1997.

Part I

Selected Papers: Embeddings

A Corrected Failure-Divergence Model for CSP in Isabelle/HOL[1]

H. Tej, B. Wolff

Universität Bremen, FB3
Postfach 330440
D-28334 Bremen
{bu,ht}@informatik.uni-bremen.de

Abstract. We present a failure-divergence model for CSP following the concepts of [BR 85]. Its formal representation within higher order logic in the theorem prover Isabelle/HOL [Pau 94] revealed an error in the basic definition of CSP concerning the treatment of the termination symbol tick.
A corrected model has been formally proven consistent with Isabelle/-HOL. Moreover, the changed version maintains the essential algebraic properties of CSP. As a result, there is a proven correct implementation of a "CSP workbench" within Isabelle.

1 Introduction

In his invited lecture at FME'96, C.A.R. Hoare presented his view on the status quo of formal methods in industry. With respect to formal proof methods, he ruled that they "are now sufficiently advanced that a [...] formal methodologist could occasionally detect [...] obscure latent errors before they occur in practice" and asked for their publication as a possible "milestone in the acceptance of formal methods" in industry.

In this paper, we report of a larger verification effort as part of the UniForM project [Kri+95]. It revealed an obscure latent error that was not detected within a decade. It can not be said that the object of interest is a "large software system" whose failure may "cost millions", but it is a well-known subject in the center of academic interest considered foundational for several formal methods *tools*: the theory of the failure-divergence model of CSP ([Hoa 85], [BR 85]). And indeed we hope that this work may further encourage the use of formal proof methods at least in the academic community working on formal methods.

Implementations of proof support for a formal method can roughly be divided into two categories. In *direct tools* like FDR [For 95], the logical rules of a method (possibly integrated into complex proof techniques) are hard-wired into the code of their implementation. Such tools tend to be difficult to modify and to formally reason about, but can possess enviable automatic proof power in specific problem domains and comfortable user interfaces.

---

1 This work has been supported by the German Ministery for Education and Research (BMBF) as part of the project **UniForM** under grant No. FKZ 01 IS 521 B2.

The other category can be labelled as *logical embeddings*. Formal methods such as CSP or Z can be logically embedded into an LCF-style tactical theorem prover such as *HOL* [GM 93] or *Isabelle*[Pau94]. Coming with an open system design going back to Milner, these provers allow for user-programmed extensions in a logically sound way. Their strength is flexibility, generality and expressiveness that makes them to *symbolic programming environments*.

In this paper we present a tool of the latter category (as a step towards a future combination with the former). After a brief introduction into the failure divergence semantics in the traditional CSP-literature, we will discuss the revealed problems and present a correction. Although the error is not "mathematically deep", it stings since its correction affects many definitions. It is shown that the corrected CSP still fulfils the desired algebraic laws. The addition of fixpoint-theory and specialised tactics extends the embedding in Isabelle/HOL to a formally proven consistent proof environment for CSP. Its use is demonstrated in a final example.

2 The Failure Divergence Semantics

In this section, we follow closely the presentation of [Cam 91], whose contribution is a formal, machine-assisted version of a subset of CSP based on [BR 85] and [Ros 88] *without* the sequential operator, the parallel interleave operator and a proof-theory based on fixpoint induction. With [Cam 91], we share some major design decisions, in particular the choice of the alternative process ordering in [Ros 88] (see below).

In its trace semantics model it is not possible to describe certain concepts that commonly arise when reasoning about concurrent programs. In particular, it is not possible to express non-determinism, or to distinguish deadlock from infinite internal activity. The failure-divergence model incorporates the information available in the trace-semantics, and in addition introduces the notions of *refusal* and *divergence* to model such concepts.

Example 2.1: Non-Determinism

Let *a* and *b* be any two events in some set of events Σ. The two processes

$$(a \rightarrow Stop) \,\Box\, (b \rightarrow Stop) \tag{1}$$

and

$$(a \rightarrow Stop) \sqcap (b \rightarrow Stop) \tag{2}$$

cannot be distinguished under the trace semantics, in which both processes are capable of performing the same sequences of events, i.e. both have the same set of traces $\{\langle\rangle,\langle a\rangle,\langle b\rangle\}$. This is because both processes can either engage in *a* and then *Stop*, or engage in *b* and then *Stop*. We would, however, like to distinguish between a deterministic choice of *a* or *b* (1) and a non-deterministic choice of *a* or *b* (2).

This can be done by considering the events that a process can refuse to engage in when these events are offered by the environment; it cannot refuse either, so we say its *maximal refusal set* is the set containing all elements of Σ other than *a* and *b*, written $\Sigma\backslash\{a,b\}$, i.e. it can refuse all elements in Σ other than *a* or *b*. In the case of the non-deterministic process (2), however, we wish to express that if the environment offers the event *a* say, the process non-deterministically chooses either to engage in *a*,

/ to refuse it and engage in *b* (likewise for b). We say therefore, that process (2) has two maximal refusal sets, $\Sigma\backslash\{a\}$ and $\Sigma\backslash\{b\}$, because it can refuse to engage in either *a* or *b*, but not both. The notion of refusal sets is in this way used to distinguish non-determinism from determinism.

Example 2.2: Infinite Chatter

Consider the infinite process

$$\mu\ X.\ a \rightarrow X$$

which performs an infinite stream of *a*'s. If one now conceals the event *a* in this process by writing

$$(\mu\ X.\ a \rightarrow X) \setminus a \tag{3}$$

it no longer becomes possible to distinguish the behaviour of this process from that of the deadlock process *Stop*. We would like to be able to make such a distinction, since the former process has clearly not stopped but is engaging in an unbounded sequence of internal actions invisible to the environment. We say the process has diverged, and introduce the notion of a *divergence set* to denote all sequences events that can cause a process to diverge. Hence, the process *Stop* is assigned the divergence set {}, since it can not diverge, whereas the process (3) above diverges on any sequence of events since the process begins to diverge immediately, i.e. its divergence set is Σ^*, where Σ^* denotes the set of all sequences with elements in Σ. Divergence is undesirable and so it is essential to be able to express it to ensure that it is avoided.

2.3 The Original Version of CSP-Semantics

The Semantic Domain. In the model of CSP presented in [BR 85] a process communicates with its environment by engaging in events drawn from some *alphabet* Σ. In the failure-divergence semantics a process is characterised by:

- its *failures* — these are sets of pairs (s,X), where s is a possible sequence of events a process can engage in (a *trace*), and X is the set of events that process can refuse to engage in (the *refusals*) after having engaged in *s*,
- its *divergences* — these are the traces after which a process may diverge.

Processes are therefore represented by pairs *(F,D)*, where *F* is a failure set and *D* is a divergence set.

The failures and divergences of a process must satisfy six well-definedness conditions (following [Ros 88]): (i) the initial trace of a process must be empty, (ii) the prefixes of all traces of a process are themselves traces of that process, i.e. traces are *prefix-closed*, (iii) a process can refuse all subsets of a refusal set, (iv) all events which are impossible to perform in the next step can be included in a refusal set, (v) a divergence set is *suffix closed*, and (vi) once a process has diverged, it can engage in, or refuse, any sequence of events.

More formally, given a (possibly infinite) set of events Σ and sets *F* and *D* such that:

$$F \subseteq \Sigma^* \times \mathbb{P}(\Sigma)$$
$$D \subseteq \Sigma^*$$

then using a set theory and predicate calculus notation similar to that adopted in [Ros 88], the above six well-definedness conditions for processes are stated as:

$$\text{is\_well\_defined}(F,D) \overset{\text{def}}{\Leftrightarrow}$$

$$(\langle\rangle,\{\}) \in F \qquad \text{(i)}$$

$$\wedge \quad \forall s,t.\ (s \wedge t, \{\}) \in F \Rightarrow (s, \{\}) \in F \qquad \text{(ii)}$$

$$\wedge \quad \forall s,X,Y.\ (s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F \qquad \text{(iii)}$$

$$\wedge \quad \forall s,X,Y.\ (s, X) \in F \wedge (\forall c \in Y.\ ((s \wedge \langle c\rangle, \{\}) \notin F)) \Rightarrow (s, X \cup Y) \in F \qquad \text{(iv)}$$

$$\wedge \quad \forall s,t.\ s \in D \Rightarrow s \wedge t \in D \qquad \text{(v)}$$

$$\wedge \quad \forall s,t.\ s \in D \Rightarrow (s \wedge t, X) \in F \qquad \text{(vi)}$$

where $\langle\rangle$ denote the empty trace, and the notation $s \wedge t$ is used to represent the concatenation of two traces s and t.

In the model originally presented in [BR 85], the converse of (iii) is also a well-definedness condition. This condition, which is shown formally below, states that a set is refusable if all its finite subsets are refusable:

$$(\forall Y \in \mathbb{F}(X).\ (s,Y) \in F) \Rightarrow (s,X) \in F \qquad (4)$$

In [Ros 88], Roscoe explains that this condition can in fact be omitted from the definition of process, but if this is done, a coarser, more complex ordering on processes must be defined since the ordering used in [BR 85] is no longer a *complete* partial order otherwise. This ordering will be called *process ordering.*

We prefer to use the process ordering of [Ros 88] (extended in [RB 89]), since we plan to investigate combinations of Z and CSP. In such a combination, Z is used to specify system-transitions via pre-and postconditions. Therefore we need a model that can cope with unbounded nondeterminism. In such a setting, a separation of the process-ordering from the refinement ordering seems unavoidable (see the detailed discussion of the counter examples in [Ros 88]). We will show in chapter 5 how one can live with two orderings from a proof theory point of view.

The semantics of the operators. We will consider the set of well-formed processes over alphabet Σ as a type *process Σ.* Then the language of CSP can now be given by the following signature (using infix notation):

Stop, Skip : process Σ

$\_\rightarrow\_$: $\Sigma \times$ process $\Sigma \rightarrow$ process Σ (* prefix of single event *)

$\Box\_:\_\rightarrow\_$: $\mathbb{P}\ \Sigma \times (\Sigma \rightarrow$process $\Sigma) \rightarrow$ process Σ (* multi-prefix of events*)

$\_\backslash\_$: process $\Sigma \times \mathbb{P}\ \Sigma \rightarrow$ process Σ (* hide *)

$\_\Box\_$, $\_\sqcap\_$, (* det. & non-det. choice *)

$\_\ ;\_$: process $\Sigma \times$ process $\Sigma \rightarrow$ process Σ (* seq. composition*)

$\mu\ \_$: (process $\Sigma \rightarrow$ process $\Sigma) \rightarrow$process Σ (* recursion *)

…

The signature above is the precise equivalent of the "grammar" in [BR 85] in higher-order abstract syntax. We will write $\Box x : A \rightarrow P\ x$ for $(\Box\_:\_\rightarrow\_)\ A\ (\lambda x.P\ x)$.

In the traditional CSP-literature, a distinguishable particular element $\surd$ ("tick") within Σ is required. It is used to indicate the termination of a process. It is crucial that it has not been distinguished on the level of the definition of the semantic domains.

Let $\mathcal{D}$ be the projection into the divergences and $\mathcal{F}$ be the projection into the failures of a process. The semantics for the CSP-operators can now be given following the lines of the example below:

$$\begin{aligned}
\mathcal{D}(P;Q) &= \mathcal{D}\,P \cup \{\, s \frown t \mid \text{tick-free}(s) \wedge s\frown\langle\surd\rangle \in \text{traces}(\mathcal{F}\,P) \wedge t \in \mathcal{D}\,Q\} \\
\mathcal{F}(P;Q) &= \{(s, X) \mid \text{tick-free}(s) \wedge (s, X \cup \{\surd\}) \in \mathcal{F}\,P\} \\
&\cup \{(s \frown t, X) \mid (s\frown\langle\surd\rangle,\{\}) \in \mathcal{F}\,P \wedge \text{tick-free}(s) \wedge (t,X) \in \mathcal{F}\,Q) \\
&\cup \{(s, X) \mid s \in \mathcal{D}\,P)\}
\end{aligned}$$

where *traces* denotes the projection into the traces and the predicate *tick-free* discriminates traces not containing tick.

Of course, for any operator it has to be shown that the results of $\mathcal{D}$ and $\mathcal{F}$, when composed to a pair, form in fact an object of type *process*, i.e. it remains to be shown that failures and divergences produced by an operator according to definitions above respect the well-formedness condition of the semantical domain, i.e. *is_well_defined(*$\mathcal{F}$*P,* $\mathcal{D}$*P)*. (Theorem 1 in [BR 85]).

In fact, this is not possible for the sequential operator.

2.4 The Problem

The problem is that from the definition one can not prove the following part of *is_well_defined*:

$$(s \frown t,\{\}) \in \mathcal{F}(P;Q) \Rightarrow (s,\{\}) \in \mathcal{F}(P;Q)$$

Consider the following case:

- $(s \frown t, \{\}) \in \mathcal{F}(P;Q)$ and
- $s \frown t \in \mathcal{D}\,P$ and
- $s \notin \mathcal{D}\,P$ and
- s is not tick-free, i.e there exist s' and s'' such that $s = s' \frown \langle\surd\rangle \frown s''$

From the definition for ; and the *is_well_defined* we can only prove that:

$$(s',\{\}) \in \mathcal{F}(P;Q)$$

but we can say nothing about *(s,{})*.

The problem is independent from axiom (4).

Conceptually, this is a consequence of an incoherent treatment of tick-freeness in divergence sets and failure sets. Although this is extremely ugly, our intuition that ticks "can appear only at the end of a trace" ([Hoa 85], pp.57, paragraph 1.9.7) has to be formally represented in the notion of well-formedness (which was, to our knowledge, never done in the CSP-Literature).

This means that the sequential operator of CSP in the sense of the definition does not form a process. This problem has meanwhile been recognised by other researchers of the CSP community ([Ros 96]), together with the fact that the problem ranges over "traditional CSP literature". Roscoe independently found this error recently and proposes a solution similar to ours.

3 Isabelle/HOL

3.1 Higher Order Logic (HOL)

In this section, we will give a short overview of the concepts and the syntax. Our logical language HOL goes back to [Chu 40]; a more recent presentation is [And 86]. In the formal methods community, it has achieved some acceptance, especially in hardware-verification. HOL is a classical logic with equality formed over the usual logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and = for negation, conjunction, disjunction, implication and equality. It is based on total functions denoted by λ-abstractions like "$\lambda x.x$". Function application is denoted by $f\,a$. Every term in the logic must be typed, in order to avoid Russels paradox. Isabelle's type discipline incorporates polymorphism with type-classes (as in Haskell). HOL extends predicate calculus in that universal and existential quantification $\forall$x. P x rsp. $\exists$x. P x can range over functions.

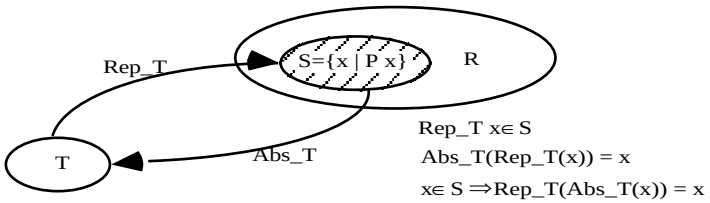
I3.2 Conservative Extensions in HOL

The introduction of new axioms while building a new theory may easily lead to inconsistency. Here, a *theory* is a pair of a signature Σ and a set of formulas *Ax* (the axioms). A theory extension can be characterised by a relation on theories:

$$(\Sigma, \mathrm{Ax}) \rightsquigarrow (\Sigma \cup \Delta\Sigma, \mathrm{Ax} \cup \Delta\mathrm{Ax})$$

Fortunately there are a number of syntactic schemes for theory extensions that maintain the consistency of the extended one — such schemes are called *conservative extensions schemes*. (For a more formal account the reader is referred to [GM 93]; one may also find a proof of soundness there). Some syntactic schemes for theory-increments $\Delta\Sigma$ and ΔAx are:

- the *constant definition* $c \equiv t$ of a fresh constant symbol *c* and a closed expression *t* not containing *c* and not containing a free type variable that does not occur in the type of *c*,
- the *type definition* (a set of axioms stating an isomorphism between a non-empty subset *S ={x ::R | P x}* of a base-type *R* and the type *T* to be defined),



- a set of equations forming a *primitive recursive scheme* over a fresh constant symbol *f*.

The basic idea of these extension schemes is to avoid logical paradoxes by avoiding general recursive axioms provoking them. Desired properties have to be derived from conservative extensions. We will build up all theories by the above extension schemes, which constitutes a consistency proof w.r.t. HOL.

3.3 Isabelle

Isabelle is a *generic* theorem prover that supports a number of logics, among them first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), the Logic of Computable Functions (LCF), and others. We only use its set-up for higher order logic (HOL). Isabelle supports natural deduction style. Its principal inference techniques are resolution (based on higher-order unification) and term-rewriting. Isabelle provides syntax for hierarchical theories (containing signatures and axioms).

In the sequel, all Isabelle input and output will be denoted in this FONT throughout this paper — enriched by the usual mathematical notation for ∀, ∃,... instead of ASCII-transcriptions.[2]

Isabelle belongs to the family of LCF-style theorem provers. This means it is a set of data types and function definitions in the ML-environment (or: "data-base"). The crucial one is the abstract data type "thm" (protected by the ML type discipline) that contains the formulas accepted by Isabelle as theorems. thm-objects can only be constructed via operations of the logical kernel of Isabelle. This architecture allows to provide user-programmed extensions of Isabelle without corrupting the logical kernel.

Technically, the proofs were done by ML-scripts performing sequences of kernel operations. These scripts were attached to the theory documents that constitute a larger system of theories, "the CSP-theory" in our case. While Isabelle is loading the theory documents and checking the proof-scripts, Isabelle can produce an HTML-document allowing to browse the CSP-theory.

4 Formalising CSP Semantics in HOL

Our formalisation of CSP profits from the powerful logical language HOL in several aspects:

- Higher order abstract syntax leads to a more compact notation avoiding auxiliary instruments like environments, updates, substitutions and process-and alphabet-variables. These issues were handled uniformly and precisely by the type-discipline.
- The data type invariant is_process (corresponding to is_well_defined in 2.3.1) can be encapsulated within a HOL-type. This leads to explicit treatment of notational assumptions and makes them amenable to static type checking.
- As we will see in the next chapter, HOL can cope with the issue of *admissibility* (as a prerequisite for fixpoint induction) in an extremely elegant way.

4.1 A Corrected Version for CSP Semantics

Whenever we changed a definition or a theorem, we will mark this by * in the sequel. The modified process invariant reads as follows[3]:

---

[2] We do not distinguish quantifications and implications at the different logical levels throughout this paper; see [Pau 94].

[3] In his "Notes on CSP" [Ros 96], Roscoe proposes two additional conditions. We have also proved formally that the CSP-theory is consistent on this basis.

$$\text{is\_process}(F, D) \stackrel{\text{def}}{\Leftrightarrow}$$

$$\begin{aligned}
&\wedge \quad (\langle\rangle,\{\}) \in F && \text{(i)}\\
&\wedge \quad \forall s,X.\ (s,X) \in F \Rightarrow \text{front-tick-free}(s) && (*)\\
&\wedge \quad \forall s,t.\ (s \hat{}\ t, \{\}) \in F \Rightarrow (s, \{\}) \in F && \text{(ii)}\\
&\wedge \quad \forall s,X,Y.\ (s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F && \text{(iii)}\\
&\wedge \quad \forall s,X,Y.\ (s, X) \in F \wedge (\forall c \in Y.\ ((s \hat{}\ \langle c\rangle,\{\}) \notin F)) \Rightarrow (s, X \cup Y) \in F && \text{(iv)}\\
&\wedge \quad \forall s,t.\ s \in D \wedge \text{tick-free}(s) \wedge \text{front-tick-free}(t) \Rightarrow s \hat{}\ t \in D && (v^*)\\
&\wedge \quad \forall s,t.\ s \in D \Rightarrow (s, X) \in F && \text{(vi)}
\end{aligned}$$

The condition * requires all traces to be front-tick-free (i.e. $\surd$ can occur at most at the end of a trace). Note that from (v *) and (vi) follows also front-tick-freeness for all divergences.

4.2 The Type Process

The encapsulation of the data type invariants *is_process* of the previous section within a type is accomplished by a type definition (see section 3.2). Note that Isabelle's notation for type constructor instances differs from the one used throughout this paper.

We introduce a type abbreviation $\text{trace}\ \Sigma$ as synonym for $(\Sigma \oplus \{\surd\})^*$ where $\oplus$ denotes the disjoint sum on sets. Further, a type abbreviation $\text{p}\ \Sigma$ will be used for the product of failures and divergences:

$$\text{p}\ \Sigma \stackrel{\text{def}}{=} \mathbb{P}(\text{trace}\ \Sigma \times \mathbb{P}\ (\Sigma \oplus \{\surd\})) \times \mathbb{P}\ (\text{trace}\ \Sigma)$$

The set of all these tuples of $\text{p}\ \Sigma$ represents the base type R of the type definition scheme. According to this extension scheme, fresh constant symbols are introduced:

$$\begin{aligned}
&\text{Abs_process} : \text{p}\ \Sigma \to \text{process}\ \Sigma\\
&\text{Rep_process} : \text{process}\ \Sigma \to \text{p}\ \Sigma
\end{aligned}$$

together with the new axioms:

$$\begin{aligned}
&\text{Rep_process}\ X : \{p.\ \text{is_process}\ p\} && \text{(Rep_process)}\\
&\text{Abs_process}(\text{Rep_process}(X)) = X && \text{(Abs_inverse)}\\
&\text{is_process}\ X \Rightarrow \text{Rep_process}(\text{Abs_process}(X)) = X && \text{(Rep_inverse)}
\end{aligned}$$

In Isabelle, this whole instance of the conservative extension scheme is abbreviated with the following statement in the theory ProcessType:

```
subtype (process) process Σ= "{p. is_process p}"
```

A first important theorem of this extension is[4]:

$$\text{is\_process}\ (\text{Rep\_process}\ P) \qquad \text{(is\_process\_Rep)}$$

---

[4] In fact, the methodology entails a proof obligation that the type is non empty, i.e. that there is a witness for which is_process holds. This trivial proof is omitted here.

We proceed with the definitions of the projections for failures and divergences:

$\mathcal{D}$ P ≡ snd (Rep_process P)
$\mathcal{F}$ P ≡ fst (Rep_process P)

where fst and snd are the usual projections into cartesian products.

The encapsulation of well-formedness within a type has the price that the constant definitions of the semantic operators are slightly unconventional. The definition of the prefix-operator in Isabelle theory notation, for instance, proceeds as follows:

```
Prefix = ProcessType +
consts   "→"  :: Σ → process Σ → process Σ          (infix 75)
defs
     Prefix_def    " a → P ≡
          Abs_process({(s,X) | s = ⟨⟩ ∧ ev a ∉ X} ∪
                              {(s, X) | s ≠ ⟨⟩ ∧ hd s = ev a ∧ (tl s,X)∈ 𝓕 P},
                              {d | d ≠ ⟨⟩ ∧ hd d = ev a ∧ tl s ∈ 𝓓 P} )"
end
```

The first line indicates that the theory Prefixis a hierarchical extension of the theory ProcessType. The pragma (infix 75) sets up Isabelle's powerful parsing machinery to parse the prefix operator the way it is used throughout this paper. The next axiom is declared to be a constant definition (Isabelle checks the syntactic side conditions) containing the abstracted tuple of failures and divergences, where hd and tl are the usual projection in lists and ev is just the injection of an element into $\Sigma \oplus \{\surd\}$.

From this definition, the traditional equations for $\mathcal{F}$ and $\mathcal{D}$ are *derived* as theorems:

$\mathcal{D}$(a → P) = {(⟨ev a⟩^s, X) | s ∈ $\mathcal{D}$ P}
$\mathcal{F}$(a → P) = {(⟨⟩,X) | ev a ∉ X}} ∪ {(⟨ev a⟩^s,X) | (s,X) ∈ $\mathcal{F}$ P}

The proof requires Rep_inverse and hence a proof of is_process for the prefix operator. We follow this technique to develop conservatively $\mathcal{F}$ and $\mathcal{D}$ for all operators.

4.3 The Semantics of the CSP-Operators

In the sequel, we will omit the technical definitions like Prefix_def and start with a listing of the derived theorems for the process projections, bearing in mind that they already subsume the proof of process well-formedness.

$$
\begin{array}{lll}
\mathcal{D}(\text{Bot}) & = \{d \mid \text{front-tick-free } d\} & (*)\\
\mathcal{F}(\text{Bot}) & = \{(s,X) \mid \text{front-tick-free } s\} & (*)\\
\mathcal{D}(\text{Skip}) & = \{\} & \\
\mathcal{F}(\text{Skip}) & = \{(\langle\rangle,X) \mid \surd \notin X\} \cup \{(\langle\surd\rangle,X)\} & \\
\mathcal{D}(\text{Stop}) & = \{\} & \\
\mathcal{F}(\text{Stop}) & = \{(\langle\rangle,X)\} & \\
\mathcal{D}(\Box x{:}A \rightarrow P\ x) & = \{(\langle ev\ a\rangle \hat{}\, s, X) \mid a \in A \wedge s \in \mathcal{D}\ P\} & \\
\mathcal{F}(\Box x{:}A \rightarrow P\ x) & = \{(\langle\rangle,X) \mid X \cap ev\ A = \{\}\} & \\
 & \cup \{(\langle ev\ a\rangle \hat{}\, s,X) \mid a \in A \wedge (s,X) \in \mathcal{F}\ P\} & \\
\mathcal{D}(P\ ;\ Q) & = \mathcal{D}\ P \cup \{s \hat{}\, t \mid s \hat{}\, \langle\surd\rangle \in traces(\mathcal{F}\ P) \wedge t \in \mathcal{D}\ Q\} & (*)\\
\mathcal{F}(P\ ;\ Q) & = \{(s,X) \mid \text{tick-free}(s) \wedge (s, X \cup \{\surd\}) \in \mathcal{F}\ P\} & \\
 & \cup \{(s \hat{}\, t,X) \mid (s \hat{}\, \langle\surd\rangle,\{\}) \in \mathcal{F}\ P \wedge (t,X) \in \mathcal{F}\ Q\} & \\
 & \cup \{(s,X) \mid s \in \mathcal{D}(P;Q)\} & (*)\\
\mathcal{D}(P \sqcap Q) & = \mathcal{D}\,P \cup \mathcal{D}\,Q & \\
\mathcal{F}(P \sqcap Q) & = \mathcal{F}\,P \cup \mathcal{F}\,Q & \\
\mathcal{D}(P \,\Box\, Q) & = \mathcal{D}\,P \cup \mathcal{D}\,Q & \\
\mathcal{F}(P \,\Box\, Q) & = \{(\langle\rangle,X) \mid (\langle\rangle,X) \in \mathcal{F}\,P \cap \mathcal{F}\,Q\} & \\
 & \cup \{(s,X) \mid s \neq \langle\rangle \wedge (s,X) \in \mathcal{F}\,P \cup \mathcal{F}\,Q\} & \\
 & \cup \{(\langle\rangle,X) \mid \langle\rangle \in \mathcal{D}\,P \cup \mathcal{D}\,Q\} & \\
\mathcal{D}(P \setminus A) & = \{s \mid \exists\ t\ u.\text{front-tick-free } u \wedge s=(\text{hide } t(ev\ A))\hat{}\, u \wedge & \\
 & \quad (t \in \mathcal{D}\ P \wedge (u = \langle\rangle \vee \text{tick-free } t) \vee & \\
 & \quad (\exists\ M.\ M \notin \mathbb{F}\ \{x.\text{True}\} \wedge t \in M \wedge & \\
 & \quad (\forall\ w \in M, w' \in M. t \leq w \wedge (w \leq w' \vee w' \leq w)) \wedge & \\
 & \quad (\forall\ w \in M.\ \text{hide } w\ (ev\ A) = \text{hide } t\ (ev\ A) \wedge & \\
 & \qquad w \in traces(\mathcal{F}\ P))))\} & (*)\\
\mathcal{F}(P \setminus A) & = \{(s,\ X) \mid \exists\ t.\ s = \text{hide } t\ (ev\ A) \wedge & \\
 & \qquad (t,\ X \cup ev\ A) \in \mathcal{F}\ P\} & \\
 & \cup \{(s,\ X) \mid s \in \mathcal{D}(P \backslash A)\ \} & (*)\\
\mathcal{D}(P\ [|A|]\ Q) & = \{s \mid \exists\ t\ u\ r\ w.\ \text{front-tick-free } w \wedge & \\
 & \qquad (\text{tick-free } r \vee w = \langle\rangle) \wedge s = r \hat{}\, w \wedge & \\
 & \qquad r\ \text{inter}((t,u),(ev\ A) \cup \{\surd\}) \wedge & \\
 & \qquad (t \in \mathcal{D}\ P \wedge u \in traces(\mathcal{F}\ Q) \vee & \\
 & \qquad t \in \mathcal{D}\ Q \wedge u \in traces(\mathcal{F}\ P))\} & (*)\\
\mathcal{F}(P\ [|A|]\ Q) & = \{(s,R) \mid \exists\ t\ u\ X\ Y.\ (t,X) \in \mathcal{F}\ P \wedge (u,Y) \in \mathcal{F}\ Q \wedge & \\
 & \qquad s\ \text{inter}\ ((t,u),\ (ev\ A) \cup \{\surd\}) \wedge & \\
 & \qquad R=((X \cup Y) \cap ((ev\ A) \cup \{\surd\})) \cup X \cap Y\} & \\
 & \cup \{(s,R) \mid s \in \mathcal{D}(P\ [|A|]\ Q)\} & (*)
\end{array}
$$

hide t A yields the trace obtained from t when concealing all events contained in A. The expression r inter ((t,u),A) means that r is obtained from t and u by synchronising their events which are contained in A and interleaving those which are not.

We adopt the more recent concept of the parallel interleave operator P[|A |]Q from the CSP-literature and define the parallel operator and the interleave operator as special cases:

$$P \,|||\, Q \equiv P[|\,\{}\,|]\, Q \qquad P \,||\, Q \equiv P\ [|\{x \mid True\}|]\ Q$$

4.4 The Generic Theory of Fixpoints

The keystone of any denotational semantics is its fixpoint theory that gives semantics to systems of (mutual) recursive equations. Meanwhile, many embeddings of denotational constructions in HOL-Systems have been described in the literature; in the Isabelle/HOL world alone, there is HOLCF [Reg 94]. However, HOLCF is a logic of *continuous functions*, while the fixpoint-theory is only a very small part of it. In contrast to HOLCF, we aim at a more lightweight approach that is parameterized (generic) with the underlying domain-theory (here: processes). Beyond the advantage of a separation of concerns, this paves the way for the reuse of this theory in other problem domains and for a future combination of CSP with pure functional programming. It is also possible with little effort to exchange the fixpoint-theory by another, for example, based on metric spaces via Banach-fixpoints.

Our formalisation of fixpoint theory in HOL will use a particular concept of Isabelle/HOL, namely polymorphism with (axiomatic) *type classes*. This is a constraint on a type variable (similar to the functional programming language Haskell) restricting it to the class of types fulfilling certain syntactic and semantic requirements.

For example, the type class $\alpha :: po$ (partial ordering) can restrict the class of all types α to those for which there is a symbol $\leq : \alpha \times \alpha \to bool$ that enjoy the property $x \leq x$ (refl_ord), $x \leq y \wedge y \leq x \Rightarrow x = y$ (antisym_ord) and $x \leq y \wedge y \leq z \Rightarrow x \leq z$ (trans_ord). Showing that a particular type (say nat with its standard ordering $\leq$) is an *instance* of this type-class, i.e. nat::po is a legal type assertion, requires the proof of the above properties follow from the definition of $\leq : nat \times nat \to bool$. Once this proof has been done while establishing the instance judgement, Isabelle can use this semantic information during static type checking.

We apply this construction to the class cpo that is an extension of po. It requires the symbol $\bot : \alpha$::cpo and the semantic properties $\bot \leq x$ (least) and directed X $\Rightarrow$ X$\neq${} $\wedge$ $\exists$b. X <<| b (complete). Here, directed : (a::po) set $\to$ bool and "is least upper bound" _<<|_:(a::po)set $\to$ a $\to$ bool are defined in the usual way for the class of partial orderings, together with lub : (a::po)set $\to \alpha$ defined as lub S $\equiv$ εx. S <<| x. For the class of cpo's, the crucial notions for continuity cont : (α::cpo $\to \beta$::cpo) $\to$ bool and the fixpoint operator fix : (α::cpo $\to \alpha$) $\to \alpha$ are defined in the usual way.

From the definition of continuity it is easy to show several proof-rules like cont(λx.x) (cont_id) and cont(λx.c) (cont_const_fun), stating the identity or any constant function to be continuous.

The first key result of the fixpoint theory is the proof of the fixpoint theorem:

$$cont\ f \Rightarrow fix\ f = f(fix\ f)$$

from the definition of $\mathsf{fix}\ f \equiv \mathsf{lub}(\bigcup_{i \in \mathbb{N}} f^i \bot)$. The second key result is the fixpoint induction theorem, that can be used as general proof principle (see chapter 5).

A third result consists in the fact that the definitions $x \leq y \equiv \mathsf{fst}\ x \leq \mathsf{fst}\ y \wedge \mathsf{snd}\ x \leq \mathsf{snd}\ y$ and $\bot \equiv (\bot, \bot)$ extend cpo's to product cpo's. From these definition the instance judgement for the type constructor "×" itself can be proved:

$$\mathsf{instance}\ \text{"}\times\text{"} : (\mathsf{cpo},\mathsf{cpo})\mathsf{cpo}$$

On this basis Isabelle's parser can parse mutual recursive definitions of the scheme:

$$\begin{array}{lll} \mathsf{letrec} & x_1 = E_1(x_1,\ldots,x_n) & \\ & \ldots & \\ & x_n = E_n(x_1,\ldots,x_n) & \\ \mathsf{in} & F(x_1,\ldots,x_n) & \end{array}$$

as $\mathsf{let}(x_1,\ldots,x_n)=\mathsf{fix}\ \lambda(x_1,\ldots,x_n).(E_1(x_1,\ldots,x_n),\ldots,E_n(x_1,\ldots,x_n))\ \mathsf{in}\ F(x_1,\ldots,x_n)$. Note that the necessary inference that $(x_1,\ldots,x_n)$ forms a cpo is done by Isabelles type inference and not by tactical theorem proving.

Similarly, the usual extension of cpo's to function spaces can be constructed. This adds arbitrary abstractions to an instance of the fixpoint theory with a concrete language; for CSP, this means an optional extension to "Higher Order CSP" allowing the expression of process schemes within this language (similar algorithmic schemes like *map* and *fold* in functional programming languages).

4.5 The Process Instance of the Fixpoint Theory

The crucial point of the instantiation is the definition of the process ordering. As already mentioned, instead of the usual refinement ordering (which is a partial ordering):

$$P \sqsubseteq Q \equiv \mathcal{F}P \supseteq \mathcal{F}Q \wedge \mathcal{D}P \supseteq \mathcal{D}Q$$

we use the more complex process ordering of [Ros 88] since otherwise the operators will not be continuous in presence of unbounded nondeterminism. A prerequisite is the definition "refusals after" $\mathcal{R}$: process $\Sigma \to$ trace $\Sigma \to \mathbb{P}(\mathbb{P}\ (\Sigma \oplus \{\surd\}))$:

$$\mathcal{R}P\ s \equiv \{\ X \mid (s,X) \in \mathcal{F}\ P\}$$

Then the process ordering is introduced as:

$$\begin{array}{lll} P \leq Q \equiv & \mathcal{D}P \supseteq \mathcal{D}Q & \\ & \wedge\ s \notin \mathcal{D}\ P \Rightarrow \mathcal{R}P\ s = \mathcal{R}Q\ s & \\ & \wedge\ \mu(\mathcal{D}\ P) \subseteq \mathit{traces}\ \mathcal{F}Q & \end{array}$$

where $\mu\ T$ denotes the set of minimal elements of a set T of finite traces. The difference between these orderings is that $\leq$ orders just approximation, but not non-determinism, i.e.:

$$\mathsf{Bot} \ \leq\ \mathsf{a}\to\mathsf{Bot} \ \leq\ \mathsf{a}\to\mathsf{a}\to\mathsf{Bot}\ \ldots$$

but:

$$\mathsf{a}\to\mathsf{Bot} \ \not\leq\ \mathsf{a}\to\mathsf{Bot} \sqcap \mathsf{b}\to\mathsf{Bot} \ \not\leq\ \mathsf{a}\to\mathsf{Bot} \sqcap \mathsf{b}\to\mathsf{Bot} \sqcap \mathsf{c}\to\mathsf{Bot} \ \not\leq\ \ldots$$

Note that the chain outlined above is ordered w.r.t. $\sqsubseteq$, however.

The well-known theorem:

$$P \leq Q \Rightarrow P \sqsubseteq Q \qquad \text{(ord\_imp\_ref)}$$

expresses that the process ordering is just a coarser ordering than the refinement ordering.

The definition of $\leq$ proves to be an instance of po. With Bot identified with $\bot$, the type α process is proven to form an instance of the type class cpo. As a consequence we inherit all definitions and theorems from the generic fixpoint theory. The CSP-operator μ is just identified with fix:(process $\Sigma \rightarrow$ process $\Sigma) \rightarrow$ process Σ.

A quite important consequence of ord_imp_ref is that the fixpoints (which are known to uniquely exist in the generic fixpoint theory) have a very particular form in the process-instance:

$$\text{fix f} = \text{Abs\_process } (\bigcap_{i \in \mathbb{N}} \mathcal{F}(f^i \text{ Bot}), \bigcap_{i \in \mathbb{N}} \mathcal{D}(f^i \text{ Bot})) \qquad \text{(fix\_eq\_lim\_proc)}$$

i.e. if a fixpoint exists w.r.t. $\leq$, than it coincides with the fixpoint w.r.t. $\sqsubseteq$.

The most complex part of the entire theory is the proof of continuity for the CSP-operators. The required properties have the following form:

cont F $\Rightarrow$ cont (λ x. a $\rightarrow$ F x)	(cont_prefix)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x $\Box$ G x)	(cont_ndet)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x $\sqcap$ G x)	(cont_det)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x ; G x)	(cont_seq)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x [|A |] G x)	(cont_parint)
cont F $\wedge$ finite A $\Rightarrow$ cont (λ x. F x \ A)	(cont_hide)
. . .	

Especially the last two theorems can pass as "highly non-trivial" even by mathematically rigorous standards; as formal proofs, they must be considered as hard. Phrases like "By Königs lemma follows the existence of finitely many traces of the form ... " required weeks of intensive work.

The collection of the above theorems (together with cont_id and cont_const_fun) is used to instantiate Isabelle's simp_tac procedure (see [Pau 94]), that applies them in a backward-chaining technique similarly to PROLOG-interpreters. This yields a tactical program that decides the continuity of arbitrary CSP-expressions with finite hide-sets as required for the application of the Knaster-Tarski theorem or for the fixpoint induction.

4.6 Laws

From the definitions of the CSP-operators the usual CSP-laws can be derived as formally proven theorems. Among them there is also the list drawn from [BR 85]:

$P \Box P = P$ $\quad$ $P \Box Q = Q \Box P$

$P \Box (Q \Box R) = (P \Box Q) \Box R$ $\quad$ $P \Box (Q \sqcap R) = (P \Box Q) \sqcap (P \Box R)$

$P \sqcap (Q \Box R) = (P \sqcap Q) \Box (P \sqcap R)$ $\quad$ $P \Box Stop = P$

$a \rightarrow (P \sqcap Q) = a \longrightarrow P \sqcap a \rightarrow Q$ $\quad$ $a \rightarrow P \Box a \rightarrow Q = a \rightarrow P \sqcap a \rightarrow Q$

$P \sqcap P = P$ $\quad$ $P \sqcap Q = Q \sqcap P$

$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$

$P \parallel Q = Q \parallel P$ $\quad$ $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$

$P \parallel (Q \sqcap R) = P \parallel Q \sqcap P \parallel R$

$a \rightarrow P \parallel b \rightarrow Q = Stop \quad \text{if } a \neq b$ $\quad$ $a \rightarrow P \parallel b \rightarrow Q = a \rightarrow (P \parallel Q) \text{ if } a = b$

$P \parallel Stop = Stop$

$P \,|||\, Q = Q \,|||\, P$ $\quad$ $P \,|||\, (Q \,|||\, R) = (P \,|||\, Q) \,|||\, R$

$P \,|||\, (Q \sqcap R) = P \,|||\, Q \sqcap P \,|||\, R$

$a \rightarrow P \,|||\, b \rightarrow Q = a \rightarrow (P \,|||\, b \rightarrow Q) \Box b \rightarrow (a \rightarrow P \,|||\, Q)$

$Skip \,;\, P = P$ $\quad$ $Stop \,;\, P = Stop$

$(a \rightarrow P); Q = a \rightarrow (P \,;\, Q)$ $\quad$ $P \,;\, (Q \,;\, R) = (P \,;\, Q) \,;\, R$

$P \,;\, (Q \sqcap R) = (P \,;\, Q) \sqcap (P \,;\, R)$ $\quad$ $(Q \sqcap R) \,;\, P = (Q \,;\, P) \sqcap (R \,;\, P)$

$P \setminus \{a\} \setminus \{a\} = P \setminus \{a\}$ $\quad$ $P \setminus \{a\} \setminus \{b\} = P \setminus \{b\} \setminus \{a\}$

$(a \rightarrow P) \setminus \{b\} = a \rightarrow P \setminus \{b\} \quad \text{if } a \neq b$ $\quad$ $(a \rightarrow P) \setminus \{a\} = (P \setminus \{a\})$

$(Q \sqcap R) \setminus A = Q \setminus A \sqcap R \setminus A$

Note that the law $P \,|||\, Stop = P$ (as in [BR 85]) does not hold as a consequence of its definition based on the parallel interleave operator. Instead, we have:

$$P|||Stop=P;Stop.$$

5 Proof Support for CSP

Fixpoint theory comes with a general induction principle called fixpoint induction. We will see that it can be expressed particularly elegant in HOL. Moreover, it will be shown that fixpoint induction can be used as proof principle for refinement proofs.

5.1 Fixpoint Induction

The idea of this proof principle is to induce a property *P* over ascending chains in directed sets. If *P* is *admissible*, i.e. if validity of *P* for all elements of a directed set *Y* always implies validity of *P* for the least upper bound of *Y*, then the task of proving a property *P* for a fixpoint fix *f* reduces to prove *P* for all its approximations.

Admissibility is a second order concept and can not be represented inside a first-order logic. In the days of the late Edinburgh LCF-prover, the task was resolved by built-in syntactical checks over predicates, the principles of which had been worked out by meta-theoretic reasoning. These checks were a constant source of errors and annoyance since they inherently conflicted with the overall design goal to keep the core of a theorem-prover small and simple.

In HOL admissibility adm: (α::cpo → bool) → bool is just an ordinary predicate (to our knowledge, the idea of an object-logical representation of admissibility is due to [Reg 94]) defined by:

adm P ≡ ∀Y. directed Y ⇒ (∀x : Y. P x) ⇒ P(lub Y) (adm_def)

which leads naturally to a list of theorems that implement the reminiscent syntactic checks in ordinary derived proof-rules *inside* the logic:

adm (λx.c)	(adm_const_fun)
adm P ∧ adm Q ⇒ adm (λx. P x ∧ Q x)	(adm_conj)
adm P ∧ adm Q ⇒ adm (λx. P x ∨ Q x)	(adm_disj)
cont f ∧ cont g ⇒ adm (λx. f x ≤ g x)	(adm_ord)
etc.	

Admissibility is used in the fixpoint induction principle in the following way:

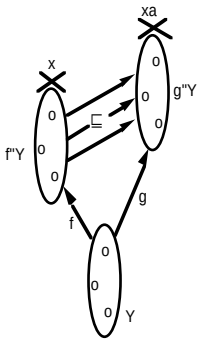
|[cont f ∧ adm P ∧ (∀ x. P x ⇒ P(f x)) |] ⇒ P (fix f) (fix_ind)

The crucial question arises, if the refinement ordering is also admissible. This is vital for the applicability of fixpoint induction for the highly desirable refinement proofs. To our knowledge, this question has not been risen so far in the literature.

Of course, such a property cannot be proven in the generic fixpoint theory (as all theorems above) but only in the process instance.

Proposition: The refinement ordering is admissible, i.e.

cont f ∧ cont g ⇒ adm (λx. f x ⊑ g x) (adm_ref_ord)



Proof-Sketch: Let f and g be continuous, Y be directed and let (∀x:Y. f x ⊑ g x) hold. Let f''Y and g''Y denote the image sets of Y w.r.t. f and g. Then the figure aside gives an overview over the situation.

Here *x* and *xa* denote the lub's w.r.t. ≤. As a consequence of ord_imp_ref and of transitivity of ⊑, both *x* and *xa* must be upper bounds w.r.t. ⊑ for f''Y. The question arises if they are also related via ⊑. The answer is positive as a consequence of fix_eq_lim_proc and the definition of ⊑, i.e. *x* is also *least* upper bound w.r.t. ⊑.

This fact gives us that living with two orders in CSP (as a price for unbounded nondeterminism) is perhaps inelegant and uncomfortable, but perfectly possible.

5.2 Take Lemmas

Fixpoint induction proofs are usually quite ingenious proofs. In this section we will discuss a more specialised proof-scheme that is more amenable to automated reasoning. This principle will also shed some light on the potential of model-checking techniques (seen from the perspective of symbolic reasoning).

The principle of take lemmas is enclosed in the take operator _↓_ : process Σ → nat → process Σ, that cuts a behaviour of a process up to a depth n, for example:

fix (λx. a → x) ↓ 1 = a → Bot.

The definition of this operator along the usual lines yields the characterising theorems:

$\mathcal{F}(P \downarrow n) = \mathcal{F}\,P \cup \{ (s, X) \mid s \in \mathcal{D}(P \downarrow n)\}$
$\mathcal{D}(P \downarrow n) = \mathcal{D}\,P \cup \{ s \frown t \mid |s|=n \wedge \text{tick-free } s \wedge \text{front-tick-free}(t) \wedge s \in \text{traces } P\}$

From there the following *cutting-rules* are derived:

$P \downarrow 0 = \text{Bot}$ $\quad$ $(a \rightarrow P) \downarrow n = a \rightarrow (P \downarrow n\text{-}1)$
$(Q \sqcap R) \downarrow n = (Q \downarrow n \sqcap R \downarrow n)$
…

The principal characteristic of this operator is that it is monotone w.r.t. $\leq$:

$n \leq m \Rightarrow P{\downarrow}n \leq P{\downarrow}m$

This fact allows us to specialise the fixpoint-induction to the $\leq$-take-lemma:

$\forall m\ ((\forall n.\ n < m \ \wedge\ P{\downarrow}n \sqsubseteq Q{\downarrow}n) \Rightarrow P{\downarrow}m \sqsubseteq Q{\downarrow}m) \ \Rightarrow\ P \sqsubseteq Q$

Note the strong similarity of this rule to Noetherian induction. Using this take-lemma, we can perform the following backward-proof example:

$\text{fix}(\lambda x.\ a \rightarrow x) \sqsubseteq \text{fix}(\lambda x.\ a \rightarrow x \sqcap \lambda x.\ b \rightarrow x)$
$\Leftarrow$ { by $\leq$-take-lemma,$\forall$-intro, $\Rightarrow$-intro}
$|[\forall n.\ n < m \ \wedge\ \text{fix}(\lambda x.\ a \rightarrow x){\downarrow}n \sqsubseteq \text{fix}(\lambda x.\ a \rightarrow x \sqcap \lambda x.\ b \rightarrow x){\downarrow}n]| \Rightarrow$
$\text{fix}(\lambda x.\ a \rightarrow x){\downarrow}m \sqsubseteq \text{fix}(\lambda x.\ a \rightarrow x \sqcap \lambda x.\ b \rightarrow x){\downarrow}m$
$\Leftarrow$ { by knaster-tarski}
$|[...]| \Rightarrow (a \rightarrow \text{fix}(...)){\downarrow}m \sqsubseteq (a \rightarrow \text{fix}(...) \sqcap b \rightarrow \text{fix}(...)){\downarrow}m$
$\Leftarrow$ { by cutting rules}
$|[...]| \Rightarrow a \rightarrow (\text{fix}(...){\downarrow}m\text{-}1) \sqsubseteq a \rightarrow (\text{fix}(...){\downarrow}m\text{-}1) \sqcap b \rightarrow (\text{fix}(...){\downarrow}m\text{-}1)$
$\Leftarrow$ { by refinement projection left}
$|[...]| \Rightarrow a \rightarrow (\text{fix}(...){\downarrow}m\text{-}1) \sqsubseteq a \rightarrow (\text{fix}(...){\downarrow}m\text{-}1)$
$\Leftarrow$ { by refinement monotonicity}
$|[\forall n.\ n < m \ \wedge\ \text{fix}(...){\downarrow}n \sqsubseteq \text{fix}(...){\downarrow}n]| \Rightarrow \text{fix}(...){\downarrow}m\text{-}1 \sqsubseteq \text{fix}(...){\downarrow}m\text{-}1$
$\Leftarrow$ { by arithmetic and assumption}
True

Even without knowing anything about tactical programming in Isabelle, it is not hard to see how this proof-technique can be mechanised. The essential difficulties are to unfold fix-terms only in a controlled way, to "drive inside" the take-operator occurrences while decreasing their offsets and to control the necessary backtracking for refinement projection left rsp. refinement projection right.

The technique resembles very much the usual graph-exploration techniques in labelled transition diagrams (as implemented in FDR). The nodes in the graph correspond to equivalence-classes on take-terms, the edges applications of the refinement monotonicity. If problematic pathological cases were avoided (so-called *non-contracting* bodies of fix like $\text{fix}(\lambda x.x)$), and if graph-regularity can be assured, this tactical program will be a (proven correct) decision procedure.

6 Example

The following example is drawn from [For 95], pp. 5. It specifies a process *COPY* that behaves like a one place buffer. Then an implementation using a separate sender *SEND* and receiver processes *REC*, communicating via a channel *mid* and an acknowledgement *ack*. Instead of using model-checking for a known, finite alphabet of events, we will prove via fixpoint induction for arbitrary alphabets that the implementation refines the specification. Note, however, that the alphabets must still

be finite because of the hiding operator in $SYSTEM$, which is known to be noncontinuous for infinite alphabets (see [BR 86]).

On the top-level of our CSP theory in Isabelle, new syntax for channels has been introduced. Hence *writing* c!a→P is represented by (c,a)→P and *receiving* c?x→P x is mapped to an appropriate representation with multi-prefixes.

Our can be represented in an Isabelle theory by introducing a data type for all involved channels. This can be done in an ML-like definition:

```
datatype channel = left | right | mid | ack
```

The process COPY : process (channel × Σ) is defined as follows:

```
COPY ≡ (letrec COPY = left?x → right!x → COPY in COPY    (COPY_def)
```

The definition of the implementation reads as follows:

```
SYSTEM ≡ (  letrec   SEND = left?x → mid!x → ack?y → SEND;
                     REC  = mid?x → right!x → ack!x → REC
            in       SEND [| SYN |] REC) \ SYN)    (SYSTEM_def)
```

where SYN ≡ {x | fst x = mid ∨ fst x = ack}.

Now we can state the desired proof-goal COPY ⊑ SYSTEM (under premise P : finite SYN) with COPY acting as specification of the behaviour of SYSTEM.

In the following presentation of the backward-proof, we suppress the required proofs of continuity (which were eliminated by an appropriate tactic). For convenience, we introduce G as abbreviation for the often re-occurring term:

```
(λu.  (left?x→mid!x→ack?y→fst u, mid?x→right!x→ack!x→snd u))
```

Then, the main steps of the refinement proof are:

```
COPY ⊑ SYSTEM
⇐   {by COPY_def, SYSTEM_def,  fix_ind, adm_ref_ord }
1)  Fixpoint induction base:
        Bot  ⊑ SYSTEM
    ⇐   {by Bot ⊑ X }
        True
2)  Fixpoint induction step:
        |[ x ⊑      (fst (fix G) [| SYN |] snd (fix G)) \ SYN]| ⇒
            left?xa → right!xa → x
        ⊑
            (fst (fix G) [| SYN |] snd (fix G)) \ SYN
    ⇐   {by knaster_tarski over both fix-terms, fst-snd-simplification}
        |[...]| ⇒
            left ? xa → right ! xa → x
        ⊑
            (left?x → mid!x → ack?y → fst (fix  G)
            [| SYN |]
             mid?x → right!x → ack!x → snd (fix G)) \ SYN
```

$\Leftarrow$ {by distributive laws of the hiding operator, the parallel interleave operator and the Mprefix operator}

$|[\ldots]| \Rightarrow$

$\quad$ left?xa $\rightarrow$ right!xa $\rightarrow$ x

$\sqsubseteq$

$\quad$ left?x $\rightarrow$ right!x$\rightarrow$((fst(fix G)|]SYN[|snd(fix G)\SYN)

$\Leftarrow$ {by monotonicity of multiprefix operator w.r.t refinement order $\sqsubseteq$ and by assumption}

True

The premise P was only used in the proof of admissibility, when applying adm_ref_ord. A careful analysis of its proof reveals that it can be strengthened to cont f $\wedge$ mono g $\Rightarrow$ adm (λx. f x $\sqsubseteq$ g x), while on the other hand a proof of monotonicity for the hide operator with arbitrary sets seems feasible. This seems to suggest that at least the class of typical refinements fix f $\sqsubseteq$ (fix g)\A (provided that f and g continuous) with one outermost hiding operator hiding away an arbitrary internal communication channel introduced by the refinement step can be handled also in the infinite case.

7 Conclusion

We have presented a corrected, shallow embedding of CSP into higher-order logic that nevertheless preserves the algebraic properties of CSP for which we have formal, machine-checked proofs. This embedding forms an implementation of a "CSP Workbench" that allows interactive theorem proving in CSP-specifications with infinite alphabet (complementary to the FDR-tool that allows automatic proofs on specialised, finite CSP-specifications). The collection of theories has been converted directly by Isabelle into a "textbook on CSP theory" available under "http://www.informatik.uni-bremen.de/~bu/isa_doc/CSP/doc/html/index.html".

Some remarks should be given on the amount of verification work. The theory presented so far required one man year (excluding a first attempt of five man months invested in a model much closer to [Hoa 85] that turned out to be infeasible). This effort could probably have been reduced by better expert advice, since our major problems came from wrong theoretic foundations, gaps in proofs etc. and not from the technicalities of "embedding" or proving. Although the effort still may be qualified as considerable, we see a need for more machine assisted verification work, since there is a tendency to dilute the formal core of a research programme, especially a successful one. In the meantime there are so many different variants of CSP, that they are very likely to be incompatible. Due to the high publication pressure, authors tend to modify the definitions according to their needs and cite the proofs from elsewhere ("proof is done analogously to [XY ??]"). In such a situation, research peers can shift more research effort to *canonical* theory representations that were verified by machine assistance.

We are not denying that formal proof activity without mathematical intuition is blind, but we would like to emphasise that intuition tends to delude more often in foundational theories of computer science that in other mathematical research areas, perhaps due to their discrete nature and resulting combinatorial complexity. The treatment of tick is an example for a unintuitive, combinatorically complex part of a complex theory. Obviously, the situation gets even worse if combinations of formal

methods — as envisaged by the UniForM project [Kri+95] — are undertaken. Nevertheless, such combination-methods are particularly desirable since "there is no single theory for all stages of the development of software [...]. Ideas, concepts, methods and calculations will have to be drawn from a wide range of theories, and they are going to have to work together consistently [...]" (again from Hoare's invited lecture at FME'96).

7.1 Future Work

We will investigate to prove the denotational semantics as described in this paper consistent with the operational semantics of FDR [For 95], i.e. we prove consistency with the formal specification of this tool (we are not planning to "prove FDR" w.r.t. this specification). As a result, one can embed the FDR-tool as a proof-oracle (external decision procedure) within Isabelle in order to build up a logically consistent, combined environment for the reasoning over CSP. This is particularly attractive, since both tools deliver complementary deduction support: Isabelle/CSP provides interactive proof support for infinite CSP, while FDR excels at automatic refinement proofs for specialised, finite CSP specifications. In such an environment, general requirements-engineering is possible, followed by a sequence of "massage steps" that make a specification amenable for FDR, concluded by combined proof-efforts of FDR and Isabelle/CSP.

We are interested in designing a transformational methodology in CSP. This means that a collection of "transformation rules" in the sense of [KSW 96a] should be designed that allow the construction of a CSP-process by identifying and refining *design-patterns.*

We work at a safe and semantically clean integration of CSP with other industry-standard specification languages like Z (whose representation in Isabelle/HOL has been worked out in [KSW 96b]). First conceptual studies for such an integration are [Fis 97].

Finally we admit that an encapsulation of the Isabelle/CSP embedding in an integrated *tool* is of crucial importance for further acceptance in industry. Following the lines of [KSW 96a], a generic user interface has been developed that can be instantiated with LCF-style theorem-prover in order to encapsulate them as a specialised tool (see [KLMW 96]). An instance of this technology with Isabelle/CSP has been envisaged. Moreover, an even wider goal of UniForM is to provide a workbench to integrate these tools and to provide them with inter-tool communication, version-management and development-management. We believe that this technology should ease the construction of powerful formal methods tools and simplify the technical side of interchanging information between them.

Acknowledgement. We would like to thank A.W.Roscoe for several hints helping us to bridge big steps in rigorous mathematical proofs. Prof. Bernd Krieg-Brückner, Thomas Santen, Sabine Dick, Christoph Lüth and Clemens Fischer read earlier versions of this paper.

References

[And 86] P.B. Andrews: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[BH 95] J. P. Bowen,M. J. Hinchey: Seven more Myths of Formal Methods: Dispelling Industrial Prejudices, in *FME'94: Industrial Benefit of Formal Methods*, proc. 2nd Int. Symposium of Formal Methods Europe, LNCS 873, Springer Verlag 1994, pp. 105-117.

[BR 85] S.D. Brookes, A.W. Roscoe: An improved failures model for communicating processes. In: S.D.Brookes (ed.): Seminar on Semantics of Concurrency. LNCS 197, Springer Verlag, pp. 281-305. 1985.

[Cam 91] A.J. Camillieri: A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics. G. Birtwistle (ed): *IVth Higher Order Workshop*, Banff 1990. Workshops in Computing, Springer Verlag, 1991.

[Chu 40] A. Church: A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940, pp. 56-68.

[Fis 97] C. Fischer: Combining CSP and Z. Submitted for publication.

[For 95] Formal Systems (Europe) Ltd: Failures-Divergence Refinement: FDR2, Dec.1995. Preliminary Manual.

[GM 93] M.J.C. Gordon,T.M. Melham: *Introduction to HOL: a Theorem Proving Environment for Higher order Logics*, Cambridge Univ. Press, 1993.

[Hoa 85] C.A.R.Hoare: Communication Sequential Processes.Prentice-Hall, 1985

[KLMW96] Kolyang, C. Lüth, T. Meier, B. Wolff: Generic Interfaces for Formal Development Support Tools. In: Workshop for Verification and Validation Tools, Bremen. to appear in LNCS.

[Kri+95] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, A. Baer, : Uniform Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995. See also the project home-page: http://www.informatik.uni-bremen.de/~uniform.

[KSW 96a] Kolyang, T. Santen, B. Wolff: Correct and User-Friendly Implementations of Transformation Systems. Proc. Formal Methods Europe, Oxford. LNCS 1051, Springer Verlag, 1996.

[KSW 96b] Kolyang, T. Santen, B. Wolff: A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy and J. Harrison (eds): Theorem Proving in Higher/Order Logics — 9th International Conference, LNCS 1125, pp. 283-298, 1996.

[Pau 94] L. C. Paulson: *Isabelle - A Generic Theorem Prover*. LNCS 828, 1994.

[RB 89] A.W. Roscoe, G. Barett: Unbounded Nondeterminism in CSP. In: M. Main, A.Melton,M.Mislove,D.Schmidt (eds): 9th International Conference in Mathematical Foundations of Programming Semantics. LNCS 442,pp. 160-193, 1989.

[Reg 94] F. Regensburger: HOLCF: Eine konservative Einbettung von LCF in HOL. Phd thesis, Technische Universität München. 1994.

[Ros 88] A.W. Roscoe: An alternative Order for the Failures Model. In: Two Papers on CSP. Technical Monograph PRG-67, Oxford university Computer Laboratory, Programming Research Group, July 1988.

[Ros 96] A.W. Roscoe, e-mail communication with the authors.

HOL-Z 2.0:
A Proof Environment for Z-Specifications

Achim D. Brucker
Albert-Ludwigs-Universität Freiburg
brucker@informatik.uni-freiburg.de

Frank Rittinger
Albert-Ludwigs-Universität Freiburg
rittinge@informatik.uni-freiburg.de

Burkhart Wolff
Albert-Ludwigs-Universität Freiburg
wolff@informatik.uni-freiburg.de

Abstract: We present a new proof environment for the specification language Z. The basis is a semantic representation of Z in a structure-preserving, shallow embedding in Isabelle/HOL. On top of the embedding, new proof support for the Z schema calculus and for proof structuring are developed. Thus, we integrate Z into a well-known and trusted theorem prover with advanced deduction technology such as higher-order rewriting, tableaux-based provers and arithmetic decision procedures. A further achievement of this work is the integration of our embedding into a new toolchain providing a Z-oriented type checker, documentation facilities and macro support for refinement proofs; as a result, the gap has been closed between a logical embedding proven correct and a *tool* suited for applications of non-trivial size.

Key Words: Theorem Proving, Refinement, Z

Category: D.2.1, D.2.4, F.3.1, F.4.1

1 Introduction

Tools for formal specification languages can roughly be divided into two categories: *straightforward design* which implements a specification environment directly in a programming language, and *embedded design* which implements it on the basis of a logical embedding in a theorem prover environment, e.g. Isabelle [Paulson, 1994]. Examples of the former are Z/EVES [ZEVES, 2003], KIV [KIV, 2003] or FDR [FDR, 2003], examples of the latter are VHDL [Reetz, 1995], HOL-Unity [Paulson, 2000], HOL-CSP [Tej and Wolff, 1997] and HOL-OCL [Brucker and Wolff, 2002].

The advantage of embedded designs such as HOL-Z (whose underlying conservative embedding of Z into the higher-order logic (HOL) instance of Isabelle has been described in [Kolyang et al., 1996]) is their solid logical basis: all symbolic computations on formulae are divided into "logical core theorems" (i.e. derived rules) and special tactical programs controlling their application. Thus, logical consistency of a tool for specification languages can be reduced to the consistency of the underlying meta-logic and the correctness of the underlying logical engine, which is in our case a well-known and accepted one. When scaling up to a tool, the problems with embedded designs are threefold:

1. A tool-oriented logical embedding must be designed for *effective deduction*. This usually conflicts with other design goals such as provability of meta-theoretic properties (e.g. completeness).

2. Embeddings often present the embedded language in the form of meta-logical formulae: this has negative effects on presentation and error-handling.

3. The embedding and the concrete prover may suggest unstructured proof attempts ("unfold everything into meta-logic, bust the pieces there ...") and an unnatural proof organization. This may be too low-level for larger developments.

In order to meet these problems, we improved our logical embedding called HOL-Z. The integrated environment — still called HOL-Z for simplicity — offers the following features:

1. HOL-Z is a "shallow embedding" [Kolyang et al., 1996]; types are handled on the meta-level, and many elements of Z are "parsed away" and represent no obstacle for deduction.

2. HOL-Z is based on a new front-end consisting of an integrated parser and type checker; this paves the way for professional documentation and high-level error-handling.

3. HOL-Z offers technical support of methodology (such as refinement, top-down proof development or proof obligation management), and support of particular "structured proof idioms" such as the schema calculus in Z.

Our first contribution in this paper consists in a proof calculus for the schema calculus of Z and its implementation based on derived rules. As the second contribution, we provide an integration of HOL-Z into a specific tool-chain in order to "scale up" the previous work on embedding Z into Isabelle/HOL to a proof environment that has been applied in several larger case studies.

2 Foundations

2.1 Isabelle/HOL

Higher-order logic (HOL) [Church, 1940; Andrews, 1986] is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, since e.g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like Standard ML (SML) or Haskell extended by logical quantifiers.

When extending logics, two approaches can be distinguished: the *axiomatic method* on the one hand and *conservative extensions* on the other. Extending HOL via axioms easily leads to inconsistency; given the fact that libraries contain several thousand theorems and lemmas, the axiomatic approach is too error-prone in practice. In contrast, a conservative extension introduces new constants (by *constant definitions*) and types (by *type definitions*) only via axioms of a particular form; a proof that conservative extensions preserve consistency can be found in [Gordon and Melham, 1993].

The HOL library provides conservative theories for the HOL-core based on type *bool*, for the numbers such as *nat* and *int*, for typed set theory based on τ *set* and a list theory based on τ *list*.

Isabelle [Paulson, 1994] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle

supports first-order logic (intuitionistic and classical), Zermelo-Fränkel set theory (ZF) and HOL, which we choose as a framework for HOL-Z.

Following the tradition of LCF-style provers, Isabelle consists of a logical engine encapsulated in an abstract data type *thm* in SML; any *thm* object has been constructed by trusted elementary rules in the kernel. Thus Isabelle supports user-programmable extensions in a logically safe way. A number of generic proof procedures (*tactics*), written in SML, have been developed. A special tactic is the simplifier based on higher-order rewriting and proof-search procedures based on higher-order resolution.

2.2 Z by Example

The formal specification language Z [Spivey, 1992] is based on typed set theory and first-order logic with equality. The syntax and the semantics are specified in an ISO-standard [ISOZ, 2002]; for future standardization efforts of operating system libraries or programming language semantics, Z is therefore a likely candidate. Z provides constructs for structuring and combining data-oriented specifications: schemas model the states of the system (*state schemas*) and operations on states (*operation schemas*), while the *schema calculus* is used to compose these sub-specifications to larger ones. We present these constructs using a standard example, Spivey's "birthday book". This simple system stores names and dates of birthdays and provides, for example, an operation to add a new birthday. In Z, abstract types for *NAME* and *DATE* can be declared that we use in a schema (consisting of a declaration part and a predicate part) to define the system state *BirthdayBook*. For transitions over the system state, the schema *AddBirthday* is used:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathsf{dom}\ birthday$$

AddBirthday

$$\Delta BirthdayBook$$
$$n? : NAME;\ d? : DATE$$
$$n? \notin known$$
$$birthday' = birthday \cup \{n? \mapsto d?\}$$

$\Delta BirthdayBook$ imports the state schema into the operation schema in a "stroked" and a "non-stroked" version: $BirthdayBook'$ and $BirthdayBook$. The resulting variables $birthday'$ and $birthday$ are conventionally understood as the states after and before the operation, respectively.

This system is refined to a more concrete one based on a state $BirthdayBook1$ containing two (unbounded) arrays and an operation that implements *AddBirthday* on this state:

*BirthdayBook*1

$$names : \mathbb{N} \pfun NAME$$
$$dates : \mathbb{N} \pfun DATE$$
$$hwm : \mathbb{N}$$
$$\forall i, j : 1 \mathinner{..} hwm \bullet i \neq j \Rightarrow names(i) \neq names(j)$$

*AddBirthday*1

$$\Delta BirthdayBook1$$
$$name? : NAME;\ date? : DATE$$
$$\forall i : 1 \mathinner{..} hwm \bullet name? \neq names(i)$$
$$hwm' = hwm + 1$$
$$names' = names \oplus \{hwm' \mapsto name?\}$$
$$dates' = dates \oplus \{hwm' \mapsto date?\}$$

The relation between abstract states (captured by the schema *Birthday*) and the concrete states (captured by *Birthday*1) is again represented in a schema in Z, namely

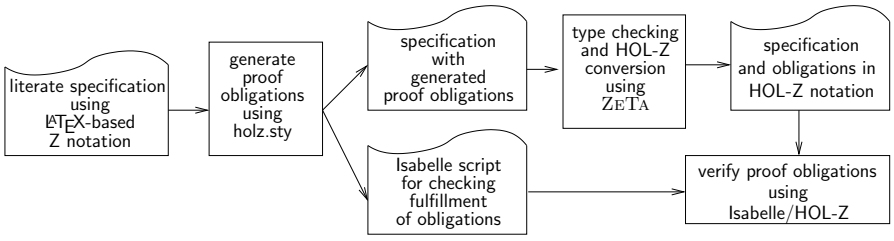


Figure 1: A Tool Chain supporting Literate Specification

the schema *Abs*; the relation defines *known* as the range of the *names*-array (upto high-water-mark *hwm*) and the relation from positionally associated *names* and *dates* as equal to the relation *birthday* from the abstract state *Birthday*:

$$
\begin{array}{|l}
\hline
\textit{Abs} \\
\textit{BirthdayBook} \\
\textit{BirthdayBook}1 \\
\hline
\textit{known} = \{i : 1 \mathinner{..} \textit{hwm} \bullet \textit{names}(i)\} \\
\forall\, i : 1 \mathinner{..} \textit{hwm} \bullet \textit{birthday}(\textit{names}(i)) = \textit{dates}(i) \\
\hline
\end{array}
$$

One can use the schema calculus to combine different operation schemas into one operation. For example, one could strengthen the *AddBirthday* operation with an operation schema *AlreadyKnown* which expresses the fact that the entry that should be added already exists in the birthday book:

$$\textit{Add} == \textit{AddBirthday} \lor \textit{AlreadyKnown}$$

The birthday book will be our running example throughout the rest of the paper.

3 A Tool Chain for Literate Specification

The core of HOL-Z, namely the logical embedding discussed in the next chapters, is now integrated into a chain of tools. We briefly describe the data flow of our tool-chain as depicted in Fig. 1; in the following sections we will describe the components of the tool-chain in more detail.

At the beginning, a normal LaTeX-based Z specification is created; the specification may contain formal text, macros for proof obligation generation and informal explanations in a mixed, "literate specification" style. Running LaTeX leads to the expansion of proof obligation macros, and also generates an Isabelle-script that checks that the obligations are fulfilled (to be run at a later stage). ZeTa takes over, extracts all Z definitions from the LaTeX source (including the generated ones) and type checks them or provides animation for some Z schemas. Our plug-in into ZeTa converts the specification (sections, declarations, definitions, schemas, ...) into SML-files that can be loaded into Isabelle. In the theory contexts provided by these files, usual Isabelle proof-scripts can be developed.

The elements of our tool chain can be technically organized in various ways. One way is to build a front-end by integrating ZeTa into XEmacs (which is our preferred

setting since, for example, a click on a type-error message leads to a highlighting of the corresponding source) and a back-end based on Isabelle. Another way is an organization into usual shell scripts, that allows for easy integration of the specification process into the general software development process, including in particular version management that allows for semantically checked specifications. In this setting, for example, one can assure that new versions of the specification document are accepted as main versions only when the proof obligation check scripts run successfully, etc.

3.1 The LaTeX-based Z Specification

The formal text in a specification document closely follows the LaTeX format of the Z standard described in [ISOZ, 2002]. In this section, we therefore focus on our add-on `holz.sty`, a macro package for generating proof obligations. We decided to use LaTeX itself as a flexible mechanism to construct and present proof obligations inside the specification — this may include consistency conditions, refinement conditions or special safety properties imposed by a special method for a certain specification architecture. Our LaTeX package `holz.sty` provides, among others, commands for generating refinement conditions as described in [Spivey, 1992]. For our running example of the birthday book's *AddBirthday* operation, we instantiate the *refinement condition* that *AddBirthday* is refined by the more concrete $AddBirthday1$ as follows:

```
\zrefinesOp[Astate=BirthdayBook, Cstate=BirthdayBook1,
            Aop=AddBirthday, Cop=AddBirthday1,
            Args={n?: NAME; d?: DATE}, Abs=Abs]{Add}
```

Here, `Astate` contains the schema describing the abstract state and `Cstate` holds the schema describing the concrete state. Based on this input, our LaTeX package automatically generates the following two proof obligations:

$$
\begin{aligned}
Add_1 == \forall\, & BirthdayBook;\ BirthdayBook1;\ n? : NAME;\ d? : DATE \bullet \\
& (\mathsf{pre}\ AddBirthday \land Abs) \Rightarrow \mathsf{pre}\ AddBirthday1 \\
Add_2 == \forall\, & BirthdayBook;\ BirthdayBook1;\ BirthdayBook1';\ n? : NAME; \\
& d? : DATE \bullet (\mathsf{pre}\ AddBirthday \land Abs \land AddBirthday1) \\
& \Rightarrow (\exists\, BirthdayBook' \bullet Abs' \land AddBirthday)
\end{aligned}
$$

These proof obligations are type-checked using ZETA and are converted to HOL-Z by our ZETA-to-HOL-Z converter.

3.2 The ZETA System

ZETA [Zeta, 2003] is an open environment for the development, analysis and animation of specifications. Specification documents are represented by *units* in the ZETA system that can be annotated with different *content* like LaTeX mark-up, type-checked abstract syntax, etc. The system is aware of dependencies between the units and attempts to exploit this when units change. An integration of ZETA into the editing environment XEmacs greatly facilitates changes and the management of consistency checking in large specifications. The contents of units is computed by adaptors, which can be plugged into the system dynamically.

Two plugins are available that are particularly important for our purpose: one consists in a *type checker* for Z based on LaTeX covering a large part of the Z standard; the other is an *animator* for Z that allows for the evaluation of Z expressions, in particular schemas. Thus, specifications can be tested easily during the specification work helping to avoid spurious errors.

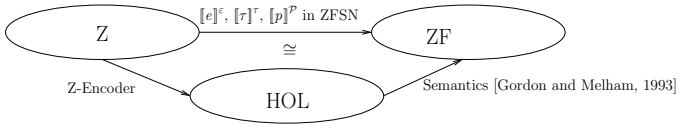


Figure 2: An Overview of Semantic Relations

3.3 ZETA-to-HOL-Z Converter

The converter consists of two parts: an adaptor that is plugged into ZETA and converts the type-checked abstract syntax of a unit more or less directly into an SML-file. On the SML side, this file is read and a theory context is built inside Isabelle/HOL-Z. This involves the conversion into the internal HOL-Z representation by the *Z-Encoder* (see Sec. 4.1), followed by an independent type checking of the result by Isabelle (ruling out that implementation errors in the Z-Encoder may yield inconsistency), followed by a check of conservativity conditions for schemas and some optimizations for partial function application in order to simplify later theorem proving.

In its present state, the converter can translate most Z constructs with the exception of user-defined generic definitions, arbitrary free types or less frequently used schema operators like hiding and piping.

4 Representing Z in Isabelle/HOL: The Foundations

In order to be self-contained, we present the foundations of the HOL-Z embedding. While most basic concepts of this embedding have been developed by one of the authors jointly with Santen and Kolyang [Kolyang et al., 1996], the implementation of the new "Z-Encoder" is a complete redevelopment; this also involves new machinery for converting types, bindings, and schema calculus constructs.

4.1 Conformance with "The Standard"

For any embedding of a logic, the question of the *faithfulness of the encoding of one calculus in another* has to be raised. This question seems to be very critical for HOL-Z since the semantics in the Z standard [ISOZ, 2002] (ZFSN) is based on Zermelo-Fränkel set theory (ZF) and not on typed set theory as HOL. ZFSN does not define a deductive system: It provides a semantics in set theory and requires "conformance" of a *deductive system* for Z, i.e. the soundness of all rules of the system with this semantics.

The core of the ZFSN semantics consists of the definition of the *partial* functions $[\![\tau]\!]^{\tau}$, $[\![e]\!]^{\varepsilon}$ and $[\![p]\!]^{\mathcal{P}}$ that assign to each element of each syntactic category (types τ, expressions e and predicates p) a *type* resp. a *value* (meaning). A calculus *conforms to the standard* if it reflects the semantic function *where it is defined*. The semantic functions are interpreted in an untyped universe of ZF. In the semantic universe, objects like $\{0, \{0\}\}$ may occur that are illegal in the typed set theory of HOL. This does not mean that $\{0, \{0\}\}$ is legal in Z; in fact, one of the major objectives of $[\![\tau]\!]^{\tau}$ is to rule out such expressions by a type-discipline that can be injectively mapped into the typed λ-calculus underlying HOL.

Fig. 2 outlines the semantic situation: Let HOL_{τ} be the set of HOL-terms of simple type τ_{λ}. Moreover, let Z_{τ} denote the set of Z-expressions of a Z-type τ_Z, and ZF the

class of sets in ZF into which all elements of Z_τ are mapped. The two type systems are both interpreted in a universe, i.e. a ZF-set. According to ZFSN, the Z universe is closed under Cartesian products and powerset-construction. According to [Gordon and Melham, 1993], the HOL universe is a set closed under function construction $A \to B$ and the type *bool*. The crucial point for the correctness of the overall approach is that both universes are *isomorphic*. Slightly simplified, the types τ_Z and τ_λ are defined as:

$$\begin{array}{ll} \tau_Z = \text{integer} & \tau_\lambda = \text{bool} \\ \quad \mid \tau_Z \times \cdots \times \tau_Z & \quad \mid \text{integer} \\ \quad \mid \langle\!\langle\, \tau_{n_1} \rightsquigarrow \tau_z, \ldots, \tau_{n_n} \rightsquigarrow \tau_Z \,\rangle\!\rangle & \quad \mid \tau_\lambda \times \tau_\lambda \\ \quad \mid \mathbb{P}\, \tau_Z & \quad \mid \tau_\lambda \to \tau_\lambda \end{array}$$

The injection from τ_Z to τ_λ is now defined as follows: integers are mapped to themselves, multiple Cartesian products $\tau_Z \times \cdots \times \tau_Z$ to binary products associated to the right, bindings (i.e. records) $\langle\!\langle\, \tau_{n_1} \rightsquigarrow \tau_Z, \ldots, \tau_{n_n} \rightsquigarrow \tau_Z \,\rangle\!\rangle$ to n-ary Cartesian products sorted by their *tag names* τ_{n_i}, and $\mathbb{P}\,\tau_Z$ to $\tau_\lambda \to$ *bool*. From this mapping, it can be seen that HOL-Z is slightly more liberal than τ_Z (it admits mixtures of Cartesian products and bindings that are not allowed in Z, for example), but the semantic domains are still isomorphic to each other. In particular, the typed set theory of Z is converted to the theory of typed characteristic functions in HOL[1]. Thus, the *Z-Encoder* maps all terms in Z_τ to specific HOL_τ-terms, such that the diagram in Fig. 2 commutes up to isomorphism for all τ. Our argument works for a monomorphic type universe. For more details and an extension to polymorphic types, see [Santen, 1998].

It is perhaps surprising to discover that the semantic basis of Z as described in the rather complex ZFSN is just an equivalent to the standard model of the typed λ-calculus. It remains to evaluate the syntactical, *notational* facet of Z can be handled by our *Z-Encoder* (to be discussed in the next section).

4.2 Encoding Schemas

Semantically, schemas are just sets of *bindings* of a certain type. However, a reference to a schema can play different *roles* in a specification: it can serve as *import* in the declaration list of other schemas (e.g. reference A in schema B in Fig. 3), it can serve as *set* (e.g. reference A or B in schema C in Fig. 3), or it can serve as *predicate* in the so-called schema calculus (see below). Moreover, references to schemas may be *decorated* by a stroke, which results in a renaming of the variables in a schema and of the tag names in the corresponding schema type by suffixing them with a stroke. Schema operators like ΔA are syntactic synonyms for $A \wedge A'$. Note that several occurrences of declarations (e.g. x_2 in schema B) in a schema are *identified* and their associated sets S_2 and T are intersected (provided that their underlying types are equal; otherwise, the whole declaration is illegal). It is this particular feature of Z that excludes a treatment of schemas by sets of "extensible records" [Naraschewski and Wenzel, 1998; Brucker and Wolff, 2002].

The first key idea for the design of HOL-Z is to compute a raw type, the *schema-signature* S_Σ for all expressions of schema type. More precisely, a schema-signature $S_\Sigma\, \mathbb{P}\langle\!\langle\, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \,\rangle\!\rangle$ is just the ordered list of tag names $[x_1, \ldots, x_n]$. In the Z-Encoder, schema-signatures are abstracted from τ_Z-types which are available whenever

[1] In our implementation, however, the situation is slightly more complex: the above said is true for schema types $\mathbb{P}\langle\!\langle\, \tau_{n_1} \rightsquigarrow \tau_Z, \ldots, \tau_{n_n} \rightsquigarrow \tau_Z \,\rangle\!\rangle$; all other $\mathbb{P}$-types are mapped to the type-constructor *set* — which is defined isomorphic to characteristic functions, but distinguished from them by the type-system of HOL. This optimization gives better access to HOL-libraries and a bit more type-safety in HOL-Z.

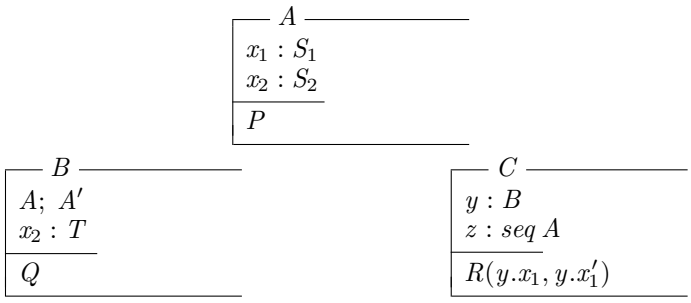


Figure 3: Schemas and their Use

ZeTa is used as front-end. When using the weaker typed e-mail format (as in locally stated proof-goals; see Sec. 5.5), they are approximated on the basis on schema syntax, on previously compiled schemas from an environment, and on computations of the effect of schema operators over schema subexpressions.

The second key idea is to represent schemas "as predicates" by default, i.e. as characteristic functions over bindings, that are represented as products. This is achieved by a pre-translator on parsed terms in the Z-Encoder, that makes implicit bindings in Z expressions — expressed by their schema-signature — explicit and generates coercions of schemas according to their role. For example, the schema declaration A, as depicted in Fig. 3, of type $\mathbb{P}\langle\!|\ x_1 \mapsto \tau_1, x_2 \mapsto \tau_2\ |\!\rangle$, is converted into the constant definition:

$$A \equiv \lambda(x_1, x_2) \bullet x_1 \in S_1 \wedge x_2 \in S_2 \wedge P$$

As a result of this presentation, the treatment of schema references as import or as set, as used in schema B, can be represented truthfully as follows:

$$B \equiv \lambda(x_1, x_2, x_1', x_2') \bullet A(x_1, x_2) \wedge A(x_1', x_2') \wedge x_2 \in T \wedge Q$$

while an expression $A \cup A$ will be represented by (asSet A)$\cup$(asSet A) (with the coercion asSet from characteristic functions to typed sets in HOL).

As mentioned, there is a further role in which schema references may be used: in the schema calculus, one may write a schema expression $A \wedge B$ which has the schema type $\mathbb{P}\langle\!|\ x_1 \mapsto \tau_1, x_2 \mapsto \tau_2, x_1' \mapsto \tau_1, x_2' \mapsto \tau_2\ |\!\rangle$. Such an expression will be represented by

$$\lambda(x_1, x_2, x_1', x_2') \bullet A(x_1, x_2) \wedge B(x_1, x_2, x_1', x_2')$$

The schema type of the conjunction of two schema expressions is the *union* of the schema signatures, provided that each tag name is associated with the same type. Thus, having "parsed away" the specific binding conventions of Z into standard λ-calculus, Isabelle's proof-engine can handle Z as ordinary HOL-formulae. There is no further "embedding specific" overhead such as predicates stating the well-typedness of certain expressions, etc; these issues are handled inside the typing discipline of HOL. Note, moreover, that our representation keeps the structure of the original Z specification — previous attempts [Bowen and Gordon, 1995] had been based on "flattening" (unfolding) of the schema notation — and allows for a controlled unfolding of schemas in the course of a proof.

So far, the presentation of binding is adequate for automatic proofs; however, in practice, realistic case studies require proofs with user interaction. This leads to the requirement that intermediate lemmas can be inserted "in the way of Z", intermediate

results are presented "Z-alike", and the proof style imposed by Z (cf. [Woodock and Davies, 1996]) can be mimicked. Therefore, we define a special "annotated" abstraction operator *SBinder* semantically equivalent to the pair-splitting λ-abstraction from the example above:

```
consts   SBinder0          ::   "[string , β ⇒ δ] ⇒ (β ⇒ δ)"
         "SBinder0 An P    ≡   P"
         SBinder           ::   "[string ,[β,γ] ⇒ δ] ⇒ ((β * γ)⇒δ)"
         "SBinder An P     ≡   (λ (x,y). (P x y))"
```

and introduce the notation:

$$\text{SB } \text{“}x\text{”} \rightsquigarrow x,\ \text{“}y\text{”} \rightsquigarrow y,\ \text{“}z\text{”} \rightsquigarrow z.P$$

for:

$$\text{SBinder “}x\text{”}(\lambda\, x.\text{SBinder “}y\text{”}(\lambda\, y.\text{SBinder0“}z\text{”}(\lambda\, z.P)))$$

Using these operators, the example above is pretty-printed by:

$$\text{SB “}x_1\text{”} \rightsquigarrow x_1,\ \text{“}x_2\text{”} \rightsquigarrow x_2,\ \text{“}x_1'\text{”} \rightsquigarrow x_3,\ \text{“}x_2'\text{”} \rightsquigarrow x_4 \bullet A(x_1, x_2) \wedge B(x_1, x_2, x_3, x_4)$$

where each field name is kept as a (semantically irrelevant) string in the representation. Thus, while the "real binding" is dealt by Isabelle's internal λ, which is subject to α-conversion, the *presentation* of intermediate results is done on the basis of the original field-names from the user's specification.

5 Proof Support for Z

Based on the semantic representation of Z in Isabelle/HOL presented in the previous section, we will now describe structured proof-support for Z.

5.1 The Mathematical Toolkit of Z

Z comes with a large toolkit of mathematical definitions concerning relations, functions, sets and bags one can build on when specifying software systems. Based on the observation that the semantic domains are equivalent, it is now straightforward to embed this "Mathematical Toolkit" conservatively in HOL.

The type of *relation* (written A↔B) is defined as the set of all pairs over A and B. Thus, in contrast to HOL, all functions are encoded by their graph. This allows for partial function spaces and for operations like the union of two functions.

The toolkit is presented as a suite of constant definitions (the technique is equivalent to [Bowen and Gordon, 1995]). On the right-hand side of the type definition some parsing information is given together with the binding values.

```
consts
  partial_func   ::"[α set,β set] ⇒ (α ↔ β) set"   ("_ ⇸ _"     [54,53] 53)
  total_func     ::"[α set,β set] ⇒ (α ↔ β) set"   ("_ → _"     [54,53] 53)
  partial_inj    ::"[α set,β set] ⇒ (α ↔ β) set"   ("_ ⤔ _"     [54,53] 53)
  func_overrid   ::"[α ↔ β, α ↔ β] ⇒ (α ↔ β)"     ("_ '(+') _" [55,56] 55)
defs
  total_func_def     "S → R   ≡{s. s ∈ S ⇸ R ∧ dom s= S}"
  partial_inj_def    "S ⤔ R   ≡{s. s ∈ S ⇸ R ∧ (∀ x1 x2 y. (x1,y) ∈s
                                  ∧ (x2,y) ∈s → x1= x2)}"
  func_overrid_def   "S (+) R ≡(dom R ◁ S) ∪ R"
```

Conformance of the set operators $\nrightarrow$,$\rightarrow$, ...,$\oplus$ of the mathematical toolkit is easy to verify: Just compare these definitions (and there are hundreds) with the ones in ZFSN. Furthermore, the laws given in [Spivey, 1992] can be derived as theorems. Especially the theorem:

$$\bigcap_{x:\{\}} P(x) = \{y.\text{true}\}$$

holds, in contrast to ZF where the result of this intersection over the empty index set is defined equal to {} because there are no universal sets in this untyped theory. In *typed* set theories like in Z or in HOL, the complement of a set is always defined.

5.2 Proof Support for the Schema Calculus

As discussed in the previous section, schemas can be used as predicates in the schema calculus, for which we implemented syntax and proof support. Besides the usual logical connectors $\wedge$,$\vee$,$\neg$,$\Rightarrow$ that may be used to connect schema expressions, there are also *schema-quantifiers* in the schema calculus. For example, the schema expression $\forall\, A \bullet B$ is a schema of type $\mathbb{P}(\!|\, x_1' \mapsto \tau_1, x_2' \mapsto \tau_2 \,|\!)$. In HOL-Z, it is represented by:

$$\text{SB}\,"x_1'" \rightsquigarrow x_3\,"x_2'" \rightsquigarrow x_4 \bullet \forall(x_1, x_2) : \text{asSet}\, A \bullet B(x_1, x_2, x_3, x_4)$$

Analogously, the following operators are defined:

- the existential quantifier $\exists\, A \bullet B$,
- the *hiding* operator $B \setminus (x_1, x_2)$ equivalent to $\exists\, M \bullet B$ (where M is a schema with empty predicate part and schema-signature $[x_1, x_2]$), and
- the $\mathsf{pre}\, B$ operator that hides all variables that have a stroke or a "!"-suffix.

The latter schema operator is motivated by the convention in Z to give variables denoting components of a successor state a stroke suffix, while variables denoting output get a "!"-suffix.

Schema quantifiers play an important role for the formulation of proof obligations and lemmas in Z. The proof obligation Add_1 for the refinement in the BirthdayBook example (see Sec. 3.1) is a pure schema expression. For inserting local lemmas into Isabelle, proof goals can be inserted directly by a suitably adapted parser (using the Z-Encoder internally) based on the compact ASCII-based e-mail format defined in ZFSN. For example, one can insert an already simplified version of Add_1 described in [Spivey, 1992, p. 138][2]:

zgoal thy
"$\forall$ *BirthdayBook*$\bullet\, \forall$ *BirthdayBook*1 $\bullet \forall$ n? $\in$*NAME*$\bullet \forall$d? $\in$*DATE*$\bullet$
(n? $\notin$ *known* $\wedge$ *known*= {n. $\exists\, i \in \#1..hwm.$ n=*names* i}
$\Longrightarrow (\forall i \in \#1..hwm.$ n? $\neq$ *names* i))";

which opens an Isabelle proof state.

Note that this statement, directly drawn from a prominent Z textbook, is strictly speaking *not* a Z-formula in the sense of ZFSN; since there are logical connectors that have a schema expression on one side and a HOL expression on the other, such mixed expressions can not be entered in the ZETA-frontend. Such use of mixed formulae in the course of proofs is in fact quite common in the Z literature (see also [Woodock and Davies, 1996]). Instead of developing a somewhat artificial, closed proof calculus

[2] For the purpose of our presentation we use the usual mathematical notation.

on schema expressions (as in [ISOZ, 2002; Henson and Reeves, 1998]), we opted for a calculus supporting such mixed forms.

The question has to be settled how the issue of binding is treated in mixed forms. Here, the general rule we adopted is that scopes introduced by schemas also extend to HOL subformulae, i.e. in $\forall S \bullet S'$, all bindings introduced by the schema-signature of S are also used to bind any free variables in S', regardless if it occurs in a schema expression or not. Moreover, the question has to be solved how schema expressions (i.e. expressions with a non-empty schema-signature) are treated logically at the top-level, since expression of the form are encoded into predicates over this schema-signature. Our answer is that we treat such "free variables" declared in a schema-signature but never bound as universally quantified; this is achieved by defining the $\vdash$ operator equivalent to the universal quantor $\forall$ of type $(\alpha \to \mathit{bool}) \to \mathit{bool}$ and adding it at the root of any schema expression. Conceptually, this means that $\vdash S$ (or: "valid S") has the meaning that the predicate must hold for all elements in the schema.

From the perspective of a mixed form calculus, it is quite clear what is needed for a joint calculus: for any construct of the schema calculus, a pair of introduction and elimination rules must be added. Since in the case of schema expressions, argument lists of predicates vary over schema-signatures, these rules are in fact rule schemes, whose individual instances are just equivalents for the usual (bounded) quantifiers and set comprehensions. In the following, we use $\mathbf{x}_i$ to denote a vector of variables $x_1, \ldots, x_n$; the juxtaposition $\mathbf{x}_i\mathbf{y}_i$ of two vectors represents their concatenation. With $\widetilde{\mathbf{x}}$ we denote a permutation of a vector $\mathbf{x}$. In the following, we represent the rule schemes of schema quantifiers in natural deduction style (to which Isabelle is mainly geared):

$$\frac{\bigwedge \mathbf{x}_i . S(\mathbf{x}_i)}{\vdash S}\ \text{turnstileI} \qquad \frac{\vdash S \qquad \begin{array}{c}[S(\mathbf{t}_i)]\\ \vdots\\ R\end{array}}{R}\ \text{turnstileE}$$

$$\frac{\bigwedge \mathbf{x}_i .\ T(\widetilde{\mathbf{x}_i\mathbf{y}_j})}{\forall S \bullet T}\ \text{sch\_allI(*)} \qquad \frac{\forall S \bullet T \qquad \begin{array}{c}[S(\mathbf{t}_i),\, T(\widetilde{\mathbf{t}_i\mathbf{y}_j})]\\ \vdots\\ R\end{array}}{R}\ \text{sch\_allE(*)}$$

$$\frac{S(\widetilde{\mathbf{y}_j\mathbf{t}'_i\mathbf{t}_k!})}{\mathit{pre}\ S}\ \text{preI(*)} \qquad \frac{\mathit{pre}\ S \quad \bigwedge \mathbf{x}'_i.\mathbf{x}_k!. \qquad \begin{array}{c}[S(\widetilde{\mathbf{y}_j\mathbf{x}'_i\mathbf{x}_k!})]\\ \vdots\\ R\end{array}}{R}\ \text{preE(*)}$$

$$\frac{S(\mathbf{t}_i) \quad S(\widetilde{\mathbf{t}_i\mathbf{y}_j})}{\exists S \bullet T}\ \text{sch\_exI(*)} \qquad \frac{\exists S \bullet T \quad \bigwedge \mathbf{x}_i . \qquad \begin{array}{c}[S(\widetilde{\mathbf{x}_i\mathbf{y}_j})]\\ \vdots\\ R\end{array}}{R}\ \text{sch\_exE(*)}$$

Note that the *turnstileI* rule introduces the equivalent of fresh free variables into a backward proof-state; consequently, schema expressions are not necessarily closed in our calculus. This motivates the following proviso (*) on most of the rules above: we require that $\mathbf{y}_k$ is the vector of free variables corresponding to the schema signature of the conclusion P (in the introduction rules) or the type of the first premise P (in the elimination rules); i.e. we require $\mathbf{y}_k = S_\Sigma(\tau)$ where τ is the type of P.

In HOL-Z, for each of these rule schemes a special tactic is provided for both forward and backward proof. While the former corresponds to a transformation on objects of type *thm* — representing formulae accepted by Isabelle as valid —, the latter is a

`tactic` that may be applied to the i-th subgoal of the current proof state. These tactics are collected in the SML package `ZProofUtils` and have the format:

```
val strip_turnstile : thm → thm
(* erases topmost turnstile ⊢ S *)

val strip_schball : thm → thm
(* erases topmost schema quantifier ∀ x: A • P x *)

val intro_sch_all_tac : int → tactic
(* pseudo introduction rule of a schema-universal quantifier
   ∀ S • P; in backwards reasoning, it eliminates a topmost
   schema-quantifier and replaces them by parameters, that
   were suitably renamed *)

val elimsch_all_tac : int → tactic
(* pseudo elimination rule of a schema-universal quantifier
   ∀ S • P; in backwards reasoning, it eliminates a topmost
   schema-quantifier in the assumption list and replaces them
   by schema variables. *)
```

An introduction and elimination rule pair for schema comprehensions $\{S \mid P \bullet E\}$ (semantically represented as $\{m \mid \exists \mathbf{x}_i : \mathrm{asSet} S \bullet P(\mathbf{x}_i) \wedge m = E(\mathbf{x}_i)\}$) is also provided. Its definition is straightforward and not really a semantic extension of HOL, merely a syntactic paraphrasing of HOL rules.

While the correctness of the calculus is assured by formal, machine-checked proofs of more atomic rules that were used inside the tactics implementing the above rule schemes, the question of (relative) completeness is more difficult to answer. Of course, since HOL includes the axiom of infinity, HOL is incomplete wrt. standard models as a consequence of Gödels incompleteness results. However, from the form of the rule schemes, it is obvious that they "transform" *all* schema expressions into standard higher-order predicate expressions in the course of a proof; this means, that for monomorphic expressions, the completeness result [Andrews, 1986, p. 197] applies here wrt. *Henkin models* and a calculus presented there. To the best of our knowledge, there is no (relative) completeness result for the polymorphic case and the precise form of rules used in Isabelle/HOL.

5.3 Interfacing Schema Expressions into Proof-Contexts

These tactics have been implemented and combined to new tactics, for example to a tactic that "strips off" all universal quantifiers (including schema quantifiers) and implications. These operators are available both in a forward and backward version and are declared as:

```
val stripS : thm → thm
(* erases topmost combination of operators as above *)

val stripS_tac : int → tactic
(* generalization of HOL's strip_tac - removes leading
   turnstiles, universal schema, bounded and unbounded
   quantifiers and implications ... *)
```

These tactical operations serve as proof-technical adaptors between Z-style lemma formats and a presentation in terms of the built-in logic `Pure` of Isabelle. If one thinks of a schema-signature as an interface to the parameters of its context proof state, both the forward and backward combinators work as kind of interface adaptors: in a backward proof, `stripS_tac` opens the local bindings hidden internally in the schema quantifiers by converting them to a parameter context, i.e. a vector of variables bound by Isabelle's meta-logic quantifier $\bigwedge$ (traditionally used to implement provisos in logical rules of the form "this variable must not occur in the assumptions", etc.). Conversely, a Z-style lemma may be logically "massaged" via `stripS` before introducing it into a backward proof; such a massage consists in erasing all schema binders and replacing bound variables in the formula by meta-variables that may be instantiated by the parameters of the proof context by Isabelle's resolution. We would like to emphasize again that all these highly non-trivial transformations on the binding structure in a mixed form of HOL-Z are based on rules derived from the definitions of the schema-logical quantifiers and thus proven correct within Isabelle. The applications of these elementary rules are controlled by tactical programs, that also apply elementary renaming tactics to present the bound variables of the proof-state in terms of user-defined names stemming from the specification.

5.4 Semantic Projections "on the fly"

From a pragmatic point of view, schemas represent nested containers of semantic knowledge of the specification. Experience shows that just expanding schemas of realistic size in the course of a proof is usually infeasible; proof states tend to become too large to be accessible to both interactive and automatic reasoning. For this, in the course of a proof, expansions of schemas should be avoided. Rather, if a particular consequence of a schema is needed, a *semantical projection lemma* should be used; for our example schema B (see Fig. 3), these are the following lemmas in mixed form:

$$\vdash B \Longrightarrow A \qquad \vdash B \Longrightarrow x_2 \in T \qquad \vdash B \Longrightarrow A' \qquad \vdash B \Longrightarrow Q$$

We provide special functions that generate semantic projections "on the fly" whenever they are needed:

```
val get_decl : theory → string → int → thm
val get_conj : theory → string → int → thm
```

The first lemma in the list of semantic projection lemmas for B can be generated by `get_decl thy "B" 1`, while the last is constructed by `get_conj thy "B" 1` ("give the first conjunct of the predicate part of schema B). Via the `stripS`-combinator, semantic projection lemmas can be converted into Isabelle's meta-logical format "on the fly" and therefore be used in a backward proof (see the example in the next section).

5.5 An Example for Structured Proofs in HOL-Z

In order to demonstrate the proof techniques introduced in the previous sections, we use a standard textbook proof [Spivey, 1992, p. 138] for the first proof obligation of the refinement of *AddBirthday* by *AddBirthday*1. Spivey argues that this theorem can be immediately reduced to the following simplified form:

zgoal thy
"$\forall$ *BirthdayBook*• $\forall$ *BirthdayBook*1 • $\forall$ n? $\in$ *NAME*•$\forall$d? $\in$ *DATE*•
(n? $\notin$ *known* $\wedge$ *known* = {n. $\exists$ *i*$\in$#1..*hwm*. n=*names* i}
$\Longrightarrow$ ($\forall$*i* $\in$ #1..*hwm*. n? $\neq$ *names i*))";

and the application of

```
by(stripS_tac 1);
```

transforms the goal into the following proof state:

1. $\bigwedge birthday\, known\, dates\ \ hwm\, names$ n? d? i.
   $[\![\ BirthdayBook(birthday,\ known);\ BirthdayBook1\ (dates,\ hwm,\ names);$
   n? $\in NAME$; d? $\in DATE$;
   n? $\notin\ known \wedge\ \ known = \{\text{n. } \exists\, i \in \#1..hwm.\ \text{n} = names\ i\}$;
   $i\ \in\ (\ \#1\ ..\ hwm)\ ]\!] \Longrightarrow$ n? $\neq\ names\ i$

Note that the quite substantial reconstruction of the underlying binding structure still leads to a proof state that is similar in style and presentation to [Woodock and Davies, 1996].

Besides the "schema calculus", Z offers a large library of set operators specifying relations, functions as relations, sequences and bags; this library (the *Mathematical Toolkit*) substantially differs in style from the Isabelle/HOL library, albeit based on the same foundations. For HOL-Z 2.0, we substantially improved this library and added many derived rules that allow for higher degree of automatic reasoning by Isabelle's standard proof procedures. For example, the goal above is simply "blown away" by:

```
auto();
```

which finishes the proof automatically.

Unfortunately, a more careful analysis of the initial proof obligation Add_1 (Sec. 3.1) and the "simplified" formulation above represents a gap in Spivey's proof. The implicitly assumed `lemma1`:

$$\vdash BirthdayBook \wedge\ (\forall i \in \#1..hwm.\ \text{n?} \neq names\ i) \Longrightarrow \text{pre AddBirthday1}$$

states that a valid concrete state $BirthdayBook1$ and the *syntactic precondition* (i.e. the conjoints in the predicate part of a schema that contain only occurrences of variables without stroke or with a "?"-suffix) implies the *semantic precondition* (i.e. *pre S* meaning "there is a successor state"). In other words, any reachable state $(\!|\ names' == a, hwm' == b, dates' == c\ |\!)$ fulfills the state invariant $BirthdayBook'$, i.e. $\forall \text{i}, \text{j} \in \#1..hwm'.\ \text{i} \neq \text{j} \Longrightarrow names'(\text{i}) \neq names'(\text{j})$. This proof constitutes in fact 80 percent of the overall proof task and is omitted here (see our example documentation in the HOL-Z 2.0 distribution).

Instead, we focus on a sample proof that shows how the bits and pieces can be brought together: We start the proof with the generated proof obligation Add_1; its formula is bound to a constant that is unfolded during the initialization of the proof:

```
{ goalw thy [Add_1def] "Add_1";}
```

$$\forall\ BirthdayBook\bullet\ (\forall\ BirthdayBook1\ \bullet (\forall\ \text{n?} \in NAME.\ \forall \text{d?} \in DATE.\ \text{pre AddBirthday} \wedge \text{Abs} \Longrightarrow \text{pre AddBirthday1})))$$

The next steps represent the opening of the bindings and some structural normalization:

$$\Longleftarrow \left\{ \begin{array}{l} \text{by (stripS\_tac 1);} \\ \text{by (Step\_tac 1);} \end{array} \right\}$$

```
∀birthday known dates  hwm names n? d?.
 ⟦ BirthdayBook(birthday, known); BirthdayBook1 (dates, hwm, names);
    n? ∈ NAME; d? ∈DATE;
    pre (AddBirthday(birthday, birthday', d?, known, known, n?));
    Abs (birthday, dates , hwm, known, names) ⟧
⟹ pre AddBirthday1
```

We apply `lemma1` and eliminate its premise `BirthdayBook` from the proof context:

```
⟸ { by (rtac (stripS lemma1) 1);
    br conjI 1;
    by (convert2hol_tac [] 1);    }
```

```
∀birthday known dates  hwm names n? d?.
 ⟦ BirthdayBook(birthday, known); BirthdayBook1 (dates, hwm, names);
    n? ∈ NAME; d? ∈DATE;
    pre (AddBirthday(birthday, birthday', d?, known, known, n?));
    Abs (birthday, dates , hwm, known, names) ⟧
⟹ ∀ i∈ #1 .. hwm. n? ≠ names i
```

Now we weaken the assumptions by applying `lemma2` (this simple lemma can be found in the HOL-Z 2.0 distribution and is not explained here), and by applying a semantic projector into schema `Abs` yielding its first conjunct:

```
⟸ { bd (stripS lemma2) 1;
    bd (stripS (get_conj thy "Abs" 1)) 1;  }
```

```
⋀birthday known dates  hwm names n? d?.
  ⟦ BirthdayBook(birthday, known); BirthdayBook1 (dates, hwm, names);
     n? ∈ NAME; d? ∈DATE; n? ∉known;
     known= known= {n. ∃ i∈#1..hwm. n=names i} ⟧
 ⟹ ∀ i∈ #1 .. hwm. n? ≠ names i
```

We are now in the position described before in Spivey's simplified proof, such that Isabelle's standard proof procedure can take over and complete the proof:

```
⟸ {auto();}
```

```
True
```

This closes our example proof. For the sake of the presentation, we deliberately chose the procedural proof-language of Isabelle and not the more recent, declarative one called *Isar*. We consider an integration into *Isar* as an add-on that complicates matters here. In any case, an integration into Isar would be a useful extension of the actual HOL-Z environment.

6 Conclusion and Further Work

We have presented HOL-Z, a tool-chain for writing Z specifications, type-checking them, and proving properties about them. In this new setting, we can write our Z specifications in the type setting system LaTeX, we can automatically generate proof

obligations, import both of them into a theorem prover environment, and use the existing proof mechanisms to gain a higher degree of automation. With the proof support for the schema calculus, realistic analysis of specifications, in particular refinement proofs, becomes feasible.

We applied HOL-Z to several large specifications, e.g. an architecture of CVS (the Concurrent Versions System) [Brucker et al., 2002] and the CORBA Security Service [Basin et al., 2002], with a focus on security analysis of CVS and CORBA. The large CORBA example (approx. 90 pages (!) that are converted and loaded in less than 5 minutes on a standard PC using PolyML) shows the feasibility of our approach for real world examples. The case studies also involved significant proofs of the refinement of an abstract architectural description to the implementation.

A consequence of our implementation of the converter is that there is no direct interaction between ZETA and HOL-Z. A closer integration of HOL-Z into ZETA would be desirable but has not been realized so far.

We will investigate if the introduction and elimination tactics can be integrated much deeper into Isabelle's `fast_tac` procedure; this would pave the way for a tableaux-based approach of reasoning over the "schema calculus" — which would be, to our knowledge, a new technique for automated deduction on Z specifications.

References

[**Andrews, 1986**] Andrews, P. B. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press.

[**Basin et al., 2002**] Basin, D., Rittinger, F., and Viganò, L. (2002). A formal analysis of the CORBA security service. In Bert, D., Bowen, J. P., Henson, M. C., and Robinson, K., editors, *ZB 2002: Formal Specification and Development in Z and B*, LNCS 2272, pages 330–349. Springer.

[**Bowen and Gordon, 1995**] Bowen, J. P. and Gordon, M. J. C. (1995). A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5–6):269–276.

[**Brucker et al., 2002**] Brucker, A. D., Rittinger, F., and Wolff, B. (2002). A CVS-Server security architecture — concepts and formal analysis. Technical Report 182, Albert-Ludwigs-Universität Freiburg.

[**Brucker and Wolff, 2002**] Brucker, A. D. and Wolff, B. (2002). A proposal for a formal OCL semantics in Isabelle/HOL. In Muñoz, C., Tahar, S., and Carreño, V., editors, *Theorem Proving in Higher Order Logics*, LNCS 2410, pages 99–114. Springer.

[**Church, 1940**] Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68.

[**FDR, 2003**] FDR (2003). Failures-divergence refinement – FDR2 user manual. `http://www.fsel.com/fdr2_manual.html`.

[**Gordon and Melham, 1993**] Gordon, M. J. C. and Melham, T. F. (1993). *Introduction to HOL.* Cambridge University Press.

[**Henson and Reeves, 1998**] Henson, M. C. and Reeves, S. (1998). A logic for the schema calculus. In Bowen, J. P., Fett, A., and Hinchey, M. G., editors, *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 172–191. Springer.

[**ISOZ, 2002**] ISOZ (2002). Z formal specification notation — syntax, type system and semantics. ISO/IEC 13568:2002, International Standard.

[**KIV, 2003**] KIV (2003). `http://i11www.ira.uka.de/~kiv/`.

[**Kolyang et al., 1996**] Kolyang, Santen, T., and Wolff, B. (1996). A structure preserving encoding of Z in Isabelle/HOL. In von Wright, J., Grundy, J., and Harrison, J., editors, *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 283–298. Springer Verlag.

[**Naraschewski and Wenzel, 1998**] Naraschewski, W. and Wenzel, M. (1998). Object-oriented verification based on record subtyping in Higher-Order Logic. In Grundy, J. and Newey, M., editors, *Theorem Proving in Higher Order Logics*, LNCS 1479, pages 349–366. Springer.

[**Paulson, 1994**] Paulson, L. C. (1994). *Isabelle: a generic theorem prover*. LNCS 828. Springer, New York.

[**Paulson, 2000**] Paulson, L. C. (2000). Mechanizing UNITY in Isabelle. *ACM Transaction on Computational Logic*, 1(1):3–32.

[**Reetz, 1995**] Reetz, R. (1995). Deep Embedding VHDL. In Schubert, E., Windley, P., and Alves-Foss, J., editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, LNCS 971, pages 277–292. Springer.

[**Santen, 1998**] Santen, T. (1998). On the semantic relation of Z and HOL. In Bowen, J., Fett, A., and Hinchey, M., editors, *ZUM '98*, LNCS 1493, pages 96–115.

[**Spivey, 1992**] Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science.

[**Tej and Wolff, 1997**] Tej, H. and Wolff, B. (1997). A corrected failure-divergence model for CSP in Isabelle/HOL. In Fitzgerald, J., Jones, C., and Lucas, P., editors, *FME 97*, LNCS 1313, pages 318–337. Springer.

[**Woodock and Davies, 1996**] Woodock, J. and Davies, J. (1996). *Using Z*. Prentice Hall.

[**Zeta, 2003**] Zeta (2003). `http://uebb.cs.tu-berlin.de/zeta/`.

[**ZEVES, 2003**] ZEVES (2003). `http://www.ora.on.ca/z-eves/welcome.html`.

UML/OCL— Semantics, Calculi, and Applications in Refinement and Test

Achim D. Brucker[1], **Burkhart Wolff**[2]

[1] Information Security, ETH Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland, e-mail: brucker@inf.ethz.ch
[2] Institut für Informatik, Universität Freiburg, George Köhler Allee 52, D-79110 Freiburg, Germany, e-mail: wolff@informatik.uni-freiburg.de

The date of receipt and acceptance will be inserted by the editor

Abstract We present a formal semantics of the Object Constraint Language (OCL) as a conservative shallow embedding in Isabelle/HOL striving for compliance with the OCL 2.0 standard. On this basis, we formally derive several equational and tableaux calculi for OCL, which form the basis for automatic proof support, including quantifiers ranging over infinite sets. We show applications of our proof environment to data refinement based on an adapted standard refinement notion, and by using tableaux calculi for automated test-case generation.
Keywords: Isabelle, UML, OCL, shallow embedding, theorem proving, refinement, specification based testing

1 Introduction

The Unified Modeling Language (UML) [42] has been widely accepted throughout the software industry for modeling object-oriented software systems and is successfully applied to diverse domains [30]. UML is supported by major CASE tools and integrated into an object-oriented software development process model that stood the test of time. The Object Constraint Language (OCL) [41,56,55] is a formal, textual extension of the UML, in particular of UML class diagrams. Being in the tradition of languages like Z [50] or VDM [27], OCL is also a data-oriented specification formalism. OCL is based on a three-valued logic with equality that is used to specify data invariants or pre-conditions and post-conditions of methods. Thus, OCL allows for

specifying constraints on the state of an object-oriented program consisting of object instances linked via references, i.e. its *object graphs*, and the transition relation over the state.

To achieve a maximum of acceptance in industry, UML/OCL is currently developed within an open standardization process by the Object Management Group (OMG); this process also led to a proposal for a mathematically rigorous semantics recently [56]. However, some details like the treatment of the underlying data-universes are handled semi-formally, and some aspects of the semantics (like recursion and late-binding) are not treated at all. Attempting to be a "practical formalism" [55], OCL has a particularly executable flavor which is useful when generating code for assertions or when animating specifications but adds some complexity to reasoning. Further, not all aspects of the language specification are consistent with the design rationale to favor executability.

The contribution of this paper is fivefold:

1. We present our work of a formal semantics [11] striving for compliance with [56]. Our semantics is purely based definitional axioms Isabelle/HOL [38]. In particular, we provide a *shallow embedding* that is suited as a basis for an OCL-tool in Isabelle.
2. Based on this embedding, called *HOL-OCL*, we provide several novel proof calculi consisting of derived rules constructed by machine-checked proofs, in particular equational and tableaux calculi.
3. We develop new proof automation, both on the level of the data structures implied by a class diagram (for which a parser is provided), and on the level of the derived rules. In particular, we integrate the tableaux calculus into Isabelle's proof procedures.
4. We present a recently developed refinement notion [9] similar to data refinement [50, 58], but adopted to OCL's logic; moreover, we show how this notion fits into our proof framework.
5. We present how our proof environment can be applied to refinement proofs and automated test-case generation based on partitioning analysis [16].

Additionally, we show that our semantics can be extended by general fixpoint semantics based on complete partial orders [57] for recursion and by temporal quantifiers, which would pave the way for OCL to annotate behavioral UML specifications such as state-charts. In the following, we describe our contributions in more detail.

In the OCL language specification [56], a semantic function is defined based on a mixture of mathematically rigorous definitions and prose such as "The OCL type Integer represents the mathematical concept of integer". As a starting point for our work, we present a

formal, machine-checked semantic model based on definitional axioms, also called *conservative embedding*, that strives for compliance with [56]; we deviate from it only in case of underspecification or logical problems or explicitly stated extensions. As a representation technique for this semantic model, we chose a *shallow embedding* [6], since we aim at efficient reasoning *in* OCL specifications and not at a meta-theoretic analysis of OCL[1]. In a shallow embedding, the types of OCL language constructs have to be represented by types of *higher-order logic* (HOL), type correctness in the representation therefore implies type correctness in OCL; no reasoning over the type correctness of OCL expressions is therefore necessary (or possible). However, a shallow representation of an object-oriented language represents a particular challenge for the "art of embedding languages in theorem provers" [39], where concepts such as undefinedness, mutual recursion between object instances, dynamic types, and extensible class hierarchies have to be managed. We meet this challenge by a particular encoding of subtyping in parametric polymorphism in HOL and a modular organization of our semantic representation.

In the OCL language specification [56] requires for the type Boolean a *Kleene Logic*, i.e. besides the constants true (t) and false (t) there is an explicit value undefined ($\bot$) that is assigned to expressions denoting illegal access to the underlying state or illegal arithmetic expressions. The logic underlying OCL is in fact a Strong Kleene Logic (SKL), i.e. the logical connectives evaluate to a defined value whenever possible: for example, $\bot \wedge \mathrm{f} = \mathrm{f}$. However, no calculus has been presented in the OCL literature that attempts to follow the language specification so far. While deduction systems have been developed only for small fragments such as the SKL core logic, we provide the first calculi for a more comprehensive subset of the language and provide machine-checked soundness proofs (by deriving them from the semantic definitions) together with an analysis of the completeness based on rigorous proofs.

The implementation of specialized (semi-) decision procedures for many-valued logics such as SKL has been investigated before, in particular based on analytic tableaux methods [28, 21, 23] or — due to the algebraic richness of SKL— on term-rewriting. Instead of starting an ad-hoc implementation from scratch, we decided to reuse the *generic*, i.e. largely logic-independent prover engine of Isabelle which is geared towards classical two-valued logics. In this paper, we present

[1] A *deep embedding* such as [43] (for a subset of Java) has typically the opposite characteristics.

a calculus for OCL in form of a labeled deduction system [19] that can be embedded inside this generic engine.

In a formal software development, a seamless transition from more abstract to more concrete object-oriented models, i.e. a formal refinement notion is necessary. While for "established" formal methods such as e.g. Z or CSP a wealth of refinement notions has been investigated and appropriate tool support is available, mechanically verifiable refinement notions for object-oriented models have raised merely academic interest so far (see [52,33,25,49,15] for general approaches, and [2] for a direct refinement to code). To our knowledge, we are the first that develop a refinement notion for OCL. We adapted Spivey's data refinement for Z [50] to OCL while attempting to exploit the three-valued logic instead of defining its effects away.

Since large parts of OCL are executable, OCL constraints have been used for run-time checking of classes and components [17,8]. In this paper, we apply the proof-procedures based on proven correct logical rules to implement known techniques [16] to *generate systematically* test-cases from a given OCL specification for its implementation. Thus, we present a specification-based (also called *black-box*) test setting for OCL. Again, we exploit the three-valence of OCL rather than avoiding it and show how to generate test-cases for testing normal behavior or faulty behavior.

Except a little detour consisting in a introduction into UML/OCL and our proof environment Isabelle/HOL in the second section, this paper follows the structure of our list of contributions: the third section is devoted to the conservative embedding of the semantics of UML/OCL, the forth section to the derivation of the proof calculi and its integration into Isabelle, and the fifth and sixth sections to the issues of refinement and test-case generation.

2 Preliminaries

2.1 A Guided Tour Through UML/OCL

The UML provides a variety of diagram types for describing dynamic (e.g. state charts, activity diagrams, etc.) and static (class diagrams, object diagrams, etc.) system properties. One of the more prominent diagram types of the UML is the *class diagram* for modeling the underlying data model of a system in an object oriented manner. The class diagram in our running example (inspired by [15]) in Fig. 1 illustrates a simple banking scenario describing the relations between the classes Bank, Account and its specialization InterestAccount. To

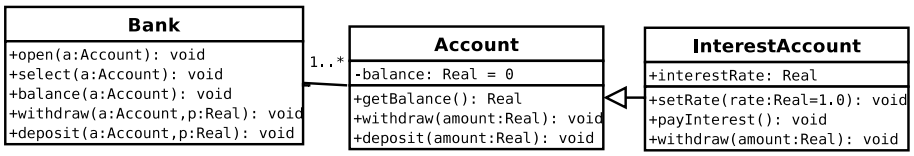


Figure 1 Modeling a simple banking scenario with UML

be more precise, the relation between instances of the classes Account and InterestAccount is called *subtyping*. A class does not only describe a set of *object instances*, i.e. record-like data consisting of *attributes* such as balance, but also functions (*methods*) defined over them.

It is possible to model relations between classes (*association*), possibly constrained by *multiplicities*. In Fig. 1, the multiplicities of the association between Account and Bank requires that every object instance of Account is associated with exactly one object instance of Bank. This captures the requirement that every account belongs to a unique bank. In the other direction, the association models that an instance of class Bank is related to a (non-empty) set of instances of class Account or its subtypes.

Understanding OCL as a data-oriented specification formalism, it seems natural to refine class diagrams using OCL for specifying invariants, pre-conditions and post-conditions of methods, e.g. we can give the specifications for the class Account as follows:

```
context Account::getBalance(): void
  post: result = balance

context Account::withdraw(amount: Real): void
  pre: amount >0
  post: balance = balance@pre - amount

context Account::deposit(amount: Real): void
  pre: amount > 0
  post: balance = balance@pre + amount
```

where in post-conditions `@pre` allows one to access the previous state.

It is characteristic for the object-oriented paradigm that the functional behavior of a class and all its methods are also accessible for all subtypes; this is called *inheritance*. A class is allowed to redefine an inherited method, as long as the method interface does not change; this is called *overwriting*, as it is done in the example for the method withdraw().

Moreover, we can refine the constraints of the overwritten method withdraw in the class InterestAccount by specifying:

```
context InterestAccount::withdraw(amount: Real): void
  pre: balance >= amount + 10
```

```
context InterestAccount::setRate(rate: Real): void
  post: interestrate = rate

context InterestAccount::payInterest(): void
  post: balance = balance@pre * (1+interestrate)
```

In UML, class members can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive data types. Moreover, OCL introduces also recursively specified methods [41]; however, at present, a dynamic semantics of a method call is missing (see [10] for a short discussion of the resulting problems).

Note, that many diagrammatic UML-features can be translated to OCL-expression without loosing any information, e.g. associations can be represented by introducing implicit set-valued attributes into the objects with an suitable data invariant describing the multiplicity. These transformations are already described in the UML-standard [42]. For our work we assume, that these transformation were already applied to the UML/OCL-model, e.g. we have not to provide a special handling for associations in our proof environment.

2.2 The Syntax of OCL.

In this paper we will use a simplified concrete syntax for OCL which is given by in EBNF notation (see Tab. 1). In particular, we omit many syntactic variants resulting from naming expressions.

Note that this simplified concrete syntax used to denote our examples contains some redundancies: the variant `expr->simpleName` is semantically equivalent with dereferencing `expr.simpleName`, it is a tribute to the OCL convention to distinguish the application of operations on collections such as `X->union(Y)`. In principle, this also holds for the prefix and infix operators. However, the semantics of this applications may be call-by-name or call-by-need; this is handled for each operator individually.

2.3 Formal and Technical Background of HOL in Isabelle

2.3.1 The Meta-Language HOL Classical higher-order logic (HOL) [13, 3] is based on a typed version of the λ-calculus. In this paper, we use types τ which are defined as $\tau ::= \alpha :: \xi \mid \chi(\tau, \ldots, \tau)$, where the set of *type variables* α is ranging over $\alpha, \beta, \ldots$, where the set of *type classes* ξ is ranging over e.g. *term* and *order*, and where the set of *type constructors* χ contains $\_ \to \_$, bool, integer, set, etc.

$$
\begin{aligned}
\mathit{invSpec} &::= \texttt{context}\ \mathit{pathName}\ \texttt{inv}\ :\ \mathit{expr}\\
\mathit{opSpec} &::= \texttt{context}\ \mathit{operation}\ \texttt{pre}\ :\ \mathit{expr}\ \texttt{post}\ :\ \mathit{expr}\\
\mathit{operation} &::= [\mathit{pathName}\ ::]\ \mathit{NAME}\ (\ [\mathit{varDecl}\ \{,\ \mathit{varDecl}\}]\ )\ [:\ \mathit{type}]\\
\mathit{varDecl} &::= \mathit{NAME}\ [:\ \mathit{type}]\ [=\ \mathit{expr}]\\
\mathit{type} &::= \mathit{pathName} \mid \mathit{collKind}\ (\ \mathit{type}\ )\\
\mathit{expr} &::= \mathit{literal} \mid -\mathit{expr} \mid \texttt{not}\ \mathit{expr} \mid \mathit{expr}\ \mathit{infixOp}\ \mathit{expr}\\
&\quad \mid \mathit{pathName}\ [\texttt{@pre}] \mid \mathit{expr}.\mathit{NAME}\ [\texttt{@pre}] \mid \mathit{expr}->\mathit{NAME}\\
&\quad \mid \mathit{expr}([\{\mathit{expr},\}\mathit{expr}]) \mid \mathit{expr}(\mathit{varDecl}|\mathit{expr})\\
&\quad \mid \mathit{expr}->\mathit{bindOp}(\mathit{varDecl}[;\mathit{varDecl}]|\mathit{expr})\\
&\quad \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr}\ \texttt{endif}\\
&\quad \mid \texttt{let}\ \mathit{varDecl}\ \{,\ \mathit{varDecl}\}\ \texttt{in}\ \mathit{expr}\\
\mathit{infixOp} &::= * \mid / \mid \texttt{div} \mid \texttt{mod} \mid + \mid - \mid < \mid > \mid <= \mid >= \mid = \mid <>\\
&\quad \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} \mid \texttt{implies}\\
\mathit{bindOp} &::= \texttt{iterate} \mid \texttt{forall} \mid \texttt{exists}\\
\mathit{literal} &::= \texttt{Integer} \mid \texttt{Real} \mid \texttt{String} \mid \texttt{true} \mid \texttt{false} \mid \texttt{OclUndefined}\\
&\quad \mid \mathit{collKind}\{\{\mathit{collLitPart},\}\mathit{collLitPart}\}\\
\mathit{collKind} &::= \texttt{Set} \mid \texttt{Bag} \mid \texttt{Sequence} \mid \texttt{Collection} \mid \texttt{OrderedSet}\\
\mathit{collLitPart} &::= \mathit{expr} \mid \mathit{expr}..\mathit{expr}\\
\mathit{pathName} &::= [\mathit{pathName}::]\mathit{NAME}
\end{aligned}
$$

Table 1 Formal Grammar of OCL (fragment)

Annotations with the default type class *term* can be omitted, i.e. instead of $\alpha :: \mathit{term}$ we may just write α. Further, instead of $\_ \to \_(\tau_1, \tau_2)$ we write $\tau_1 \to \tau_2$. The terms of HOL are λ-terms defined as $\Lambda ::= C \mid V \mid \lambda V.\Lambda \mid \Lambda\Lambda$, where C is the set of *constants* like True, False, etc. and where V is the set of *variables* like x,y,z. *Abstractions* and *applications* are written $\lambda x.e$ and $e\ e'$ or $e(e')$ respectively. A subset of λ-terms may be *typed*, i.e. terms may be associated to types by an inductive type inference system similar to the programming language Haskell or (to a lesser extent) SML. We do not give a formal definition of the type inference system here and refer the interested reader to [37], where also a type inference algorithm is described. Throughout this paper, we will only show type-checked λ-terms and use an intuitive understanding of types.

The logical terms of HOL are based on the logical connectives $\neg$, $\wedge$, $\vee$ and $\to$ which are constants of type bool $\to$ bool or bool $\to$ bool $\to$ bool (also written as $[\text{bool}, \text{bool}] \to \text{bool}$). Quantifiers are represented by higher-order abstract syntax; this means that $\forall$ and $\exists$ are usual constants of type $(\alpha \to \text{bool}) \to \text{bool}$ and that terms of

the form $\forall(\lambda x.P\ x)$ were written $\forall x.P$ ($\exists$ analogously). The Hilbert operator $\varepsilon x.P$ returns an arbitrary x that makes $P\ x$ true; in itself, the Hilbert operator turns HOL into a classical logic. Further, there is the logical equality $=$ of type $[\alpha, \alpha] \to$ bool. HOL may be interpreted in standard or non-standard models assigning types to carrier sets, logical operators in functions over them, etc.; since a notion of model for HOL is not necessary in this paper, we only refer to [20] here.

The logic of HOL is based on a surprisingly small set of axioms or elementary inference rules; besides implication introduction, modus ponens and $(P = \text{True}) \vee (P = \text{False})$, only the usual laws for axiomatizing equality (reflexivity, symmetry, transitivity, extensionality and substitutivity) are needed. Finally, the axiom of infinity allows for specifying a type constructor *ind* to have an infinite carrier set and builds the basis of natural numbers and data types.

This logical core of HOL can be extended by solely using definitional axioms — so called *conservative extensions*; c.f. [20] — to a rich specification language comprising a typed set theory (by abstracting characteristic functions), least and greatest fixpoints, well-founded orderings and well-founded recursion, products and sums, natural numbers, integers, and even a theory on real numbers and non-standard analysis.

Pragmatically, HOL can be viewed as a typed functional programming language like Haskell extended by an extensional equality and logical quantifiers. The libraries offer a "semantic toolkit" consisting of mathematical standard structures enabling to give formal semantics to programming and specification languages.

2.3.2 The Logical Framework Isabelle Isabelle [38] is a logical framework providing a logical core language based on an intuitionistic fragment of higher-order logic. As such, it implements the same term-language and type system as previously described for HOL. The built-in logical core language comprises a congruence $\equiv$, a meta-implication $\Longrightarrow$ and a meta-quantifier $\bigwedge x.Px$. The meta-implication is used to represent *logical rules*: a Horn-clause $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, written $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$, is viewed as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}":

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}} \qquad \frac{\begin{array}{c}[A_1]\\ \vdots \\ A_2\end{array}}{A}$$

To the right, we see the natural deduction rule "if A_2 can be inferred from assumption A_1, infer A" which is represented as the "second-order" Horn-clause $(A_1 \Longrightarrow A_2) \Longrightarrow A$.

In general, the meta-quantifier is used to represent eigenvariables and turns out to be flexible mechanism to represent skolemizations for quantifiers; dually, Isabelle's term language comprises meta-variables (denoted $?x, ?y, ?z, \ldots$) that represent "terms to be substituted" during proof. For example, the universal quantifier for HOL is captured by the two rules[2]:

$$\frac{\forall x.P(x)}{P(?x)} \qquad \frac{\bigwedge x.P(x)}{\forall x.P(x)}$$

The deduction engine of Isabelle is based on higher-order resolution; this means that the meta-variables are substituted during the inferences as needed (and usually automatically). Isabelle is integrated into the programming language SML and can be extended by user-programmed tactics in a logically safe way.

The "raison d'être" of Isabelle is to encode other logical languages, both with respect to their their syntax as well as their deductive system. The syntax of a language can be described using higher-order syntax and powerful pretty-printing mechanisms. The deductive system may be specified by logical rules in the built-in logical core language, let it be particular axioms or derived rules. Many logics have been encoded this way; HOL is only one of many possible choices.

3 A Formal Semantics of UML/OCL in Isabelle/HOL

As part of the OCL standard, a semi-formal semantics is given via a semantic interpretation function. As usual, the semantic interpretation function I maps *syntactic expressions* e of the language and *contexts* γ to values in some semantic domain $\mathscr{D}$. In general, contexts can be *environments* (traditionally mapping variable symbols to values) or *stores* (usually mapping references to values) or combinations thereof. As we will see, our semantic construction is in large parts independent from the precise form of the context.

In the OCL standard, the semantic function is built over expressions accessing two underlying states. In post-conditions, path-expressions can access the *current state* and the *previous state*. Accesses on both states may be arbitrarily mixed, e.g. `self.x@pre.y`

[2] The presentation of the first rule is in fact slightly simplified: in the rule, *all* free variables are treated as meta-variables; however, applying a rule in Isabelle usually leads to immediate substitutions of them *except* $?x$.

denotes an object that was constructed by dereferencing in the previous state and selecting attribute `x` in it, while the next dereferencing step via `y` is done in the current state. Thus, method specifications represent state transition relations, i.e. the context will be a pair of system states.

The concepts of semantic function I and semantic domain $\mathscr{D}$ are realized in a shallow embedding as follows: consider a semantic function defined for the case of a binary operator (as in [56, Def. 5.40(iv)]):

$$I[\![e + e']\!]\ \gamma = I[\![e]\!]\ \gamma \oplus I[\![e']\!]\ \gamma$$

where $+$ is a syntactic symbol of the logical language to be defined (the *object-language*, e.g. OCL) and $\oplus$ is an operation of the logical language in which the semantic function is described (the *meta-language*, e.g. HOL). Introducing a particular combinator $\mathit{lift}_2\, f\ X\ Y\ \gamma = f(X\ \gamma)(Y\ \gamma)$ allows for the following alternative representation of the scheme above:

$$I[\![e + e']\!] = \mathit{lift}_2(\oplus)(I[\![e]\!])(I[\![e']\!])$$

The core idea of a shallow embedding is to implement I in this equation by the identity, which has a number of consequences:

1. Definitions consequently have the form: $e + e' = \mathit{lift}_2\ \oplus\ e\ e'$ or just: $+ = \mathit{lift}_2\ \oplus$ (exploiting extensionality).
2. Variables of the object language are represented by variables of the meta language denoting functions from context to values. Environments in the traditional sense are lifted to the meta level.
3. The type of $e + e'$ is identified with the type τ of the semantic domain. Using a type-indexed family of domains $\mathscr{D}^\tau$, the meta language implements a type structure of the object language.

In this paper, instead of the "textbook-style" description of semantics using I, we will use the "combinator-style" semantics shown equivalent above. Besides brevity and conciseness, this style enables one to formalize certain aspects of the semantic definition once and for all by defining suitably combinators for them. Here, this aspect is concerned with the parameter passing of the context; in the following sections, these aspects are the handling of strictness, and undefined values. Moreover, we develop generic theorems for these combinators, which greatly facilitates the automatic derivation of many properties resulting from the definitions. Further, it is straight-forward to generate textbook-style semantics from combinator-style semantics via unfolding the combinators and inserting I automatically.

We now turn to the previously mentioned type-indexed domains and its resulting typing discipline, which should be an approximation

of the type discipline of OCL. On the one hand, in order to represent the typing of expressionsin a shallow-style, for each type τ of OCL a $\mathscr{D}^\tau$ is needed. On the other hand, there must be a universe $\mathscr{U}$ of domains in order to model the underlying store as a function of type ref $\rightarrow \mathscr{U}$. Conceptually, the universe can be understood as the sum $\mathscr{D}^{\tau_1} + \cdots + \mathscr{D}^{\tau_n}$ and attribute accessor functions and their combinations (hence: path expressions) are projections from a state over the universe to type-indexed domains. This is complicated by the fact that τ_i may be a subtype of τ_j which should be reflected by the set inclusion of the corresponding domains: $\mathscr{D}^{\tau_i} \subset \mathscr{D}^{\tau_j}$. Further, adding a class to a class hierarchy leads to a new type τ_{n+1}; thus, extending a universe naively to $\mathscr{D}^{\tau_1} + \cdots + \mathscr{D}^{\tau_{n+1}}$ results in a different type. This is unsatisfactorily both for theoretical and practical reasons: theoretically, the relation between these universes should be explicit (although constructions like "the set of all universes" are too large in a set-theoretic sense), and pragmatically, it should not be necessary to rerun of all proof scripts whenever we extend a class diagram of a system.

For the rest of the section, we proceed as follows: after introducing basic combinators we present the principles of the *extensible semantic universe* $\mathscr{U}$ for a class-hierarchy; this universe construction leads to a *semantic coding scheme* that gives semantic to OCL path expressions to attributes. Based on $\mathscr{U}$, concepts like the *system state* and *relations* over the object were built together with state access operations such as $X.\texttt{allInstances}()$. We describe the semantics in the expression language for the built-in operators like $\texttt{if}\, c\, \texttt{then}\, e\, \texttt{else}\, e'\, \texttt{endif}$, for the logical operators like $x\, \texttt{and}\, y$ or library operators like $X\, \texttt{->union}\, (Y)$. Finally, we describe the semantics of the method specification together with method invocation.

3.1 Preliminaries: Lift Combinators

In this section, we will complete our treatment of *context lifting combinators* that are a prerequisite of our shallow representation. The following type constructor $VAL_\sigma(\tau)$ makes this process explicit

$$\textbf{types} \quad VAL_\sigma(\tau) = \sigma \rightarrow \tau$$

The lift combinators that are representing the passing of context implied by the semantic interpretation function are defined as follows:

$$\begin{aligned} &lift_0 \quad :: \alpha \rightarrow VAL_\sigma(\alpha) \\ &lift_0\, f \equiv \lambda\, st\, .f \end{aligned}$$

$$
\begin{aligned}
&\mathit{lift}_1 \quad :: (\alpha \to \beta) \to \mathit{VAL}_\sigma(\alpha) \to \mathit{VAL}_\sigma(\beta)\\
&\mathit{lift}_1\ f \equiv \lambda X\ \ st\,.f(X\ \ st)\\
&\mathit{lift}_2 \quad :: ([\alpha, \beta] \to \gamma) \to [\mathit{VAL}_\sigma(\alpha), \mathit{VAL}_\sigma(\beta)] \to \mathit{VAL}_\sigma(\gamma)\\
&\mathit{lift}_2\ f \equiv \lambda X\ Y\ \ st\,.f(X\ \ st)(Y\ \ st)
\end{aligned}
$$

The types of these combinators reflect their purpose: they "*lift*" operations from HOL to semantic functions that operate on the context.

3.2 Preliminaries: Undefinedness and Strictness Combinators

In OCL the notion of explicit undefinedness is part of the language, both for the logic and the basic values [56, p. 2–9]. This postulates the strictness of all operations (the logical operators are explicit exceptions) and rules out a modeling of undefinedness through underspecification. Thus, the language has a similar flavor to LCF or SPECTRUM [7] and represents a challenge for automated reasoning.

To handle undefinedness systematically, we introduce a special type class bot which requires that each type in this class possesses a particular constant $\bot$. For all types in this class, concepts such as *definedness* $\mathit{DEF}(x) \equiv (x \neq \bot)$ or *strictness of a function* $\mathit{isStrict}(f) \equiv (f(\bot) = \bot)$ are introduced. We define a combinator

$$\mathit{strict}\ f\ x \equiv \mathbf{if}\ \mathit{DEF}(x)\ \mathbf{then}\ f(x)\ \mathbf{else}\ \bot$$

that produces a strict version out of an arbitrary functions f. Again, we favor combinator-style to textbook-style semantics. The operator $\oplus$ (used in Sec. 3.1 and corresponding to $I(+)$ in [56, A-11]) is defined in *HOL-OCL* by

$$\oplus \equiv \mathit{strict}(\lambda x.\, \mathit{strict}(\lambda y. \lfloor \lceil x \rceil + \lceil y \rceil \rfloor))$$

where $+$ is now the usual addition from the HOL-theory Integer. In the semantics of OCL [56, page A-11] this definition is presented as:

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

By unfolding the combinator *strict*, the reader may easily convince himself that these two definitions are equivalent, up to the issue of type lifting (requiring $\lfloor \_ \rfloor$ and $\lceil \_ \rceil$; cf. [57]) which has been consequently ignored throughout [56].

We represent type lifting in our framework by a type constructor that assigns to each type τ a *lifted type* $\tau_\bot$. All types lifted by this type

constructor are automatically in the type class bot but not necessarily vice versa. The function $\lfloor\_\rfloor : \alpha \to \alpha_\perp$ denotes the injection, the function $\lceil\_\rceil : \alpha_\perp \to \alpha$ its inverse. The case distinction function

$$\text{upCase}(f, c, x) = \begin{cases} c & \text{if } x = \perp \\ f(k) & \text{if } x = \lfloor k \rfloor \end{cases}.$$

will also be written:

$$\text{case } x \text{ of } \lfloor k \rfloor \Rightarrow f(k) \;\mid\; \perp \Rightarrow c.$$

3.3 Extensible Object Models and Path Expressions

The main goal of our encoding scheme of class hierarchies is to provide *typed object constructor* and *object accessor functions* for a given set of classes to be inserted into an existing hierarchy. The coding scheme will be presented in two steps: in this section, we will describe *raw* constructors and accessors, while in Sec. 3.3.4 a scheme for accessors reflecting the OCL types is presented.

3.3.1 Managing Holes in Universes Our solution to the extendibility problem of universes is based on the observation that potential extensions both of new classes and inheriting ones can be represented by type-variables. Pictorially spoken, we add domains into universes with "holes", which can be filled by new classes. Consequently, theorems established over a concrete class-diagram implicitly hold over all extensions of it, which will result in instantiations of these variables.

Since objects can be viewed as records consisting of *attributes*, and since the object alternatives can be viewed as variants, it is natural to construct the "type of all objects", i.e. the semantic universe $\mathscr{U}_x$ corresponding to a certain class hierarchy x, by Cartesian products and by type sums (based on the constructors $Inl : \alpha \to \alpha + \beta$ and $Inr : \beta \to \alpha + \beta$ from the meta-language HOL).

In our scheme, a class can be extended in two ways: either, an alternative to the class is added *at the same level*, which corresponds to the creation of an alternative subtype of the supertype (the β-*instance*), or a class is added *below*, which corresponds to the creation of an initial subtype (the α-*instance*). This process is illustrated in Fig. 2; in the first line a UML class diagram only consisting out of one class A together with a abstract representation (a jigsaw-piece) characterizing the two extension possibilities is presented. The type $\mathscr{U}^1$ of the universe describing this class diagram is also given.

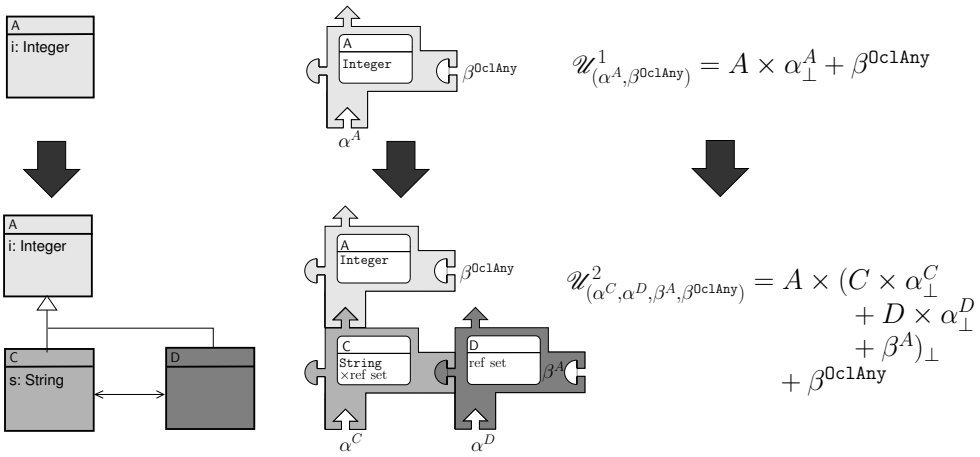


Figure 2 Extending Class Hierarchies and Universes with Holes

The insertion of a class corresponds to *filling a hole ν by a record T* is implemented by the particular type instance $\nu \mapsto ((T \times \alpha_\perp) + \beta)$. Thus, if we extend the universe $\mathscr{U}^1$ by two new classes A and B which are both subtypes of A, we can construct a new universe $\mathscr{U}^2$ which is just a type instance of $\mathscr{U}^1$. In particular: an α-extension with C β-extended by D. Thus, properties proven over $\mathscr{U}^1$ also hold for $\mathscr{U}^2$.

The initial configuration of any class hierarchy is given by the OCL standard; the library for this basic configuration is described in Sec. 3.4.4. This configuration corresponds to the indexed universes consisting of the real numbers, strings, and bool:

$$\begin{aligned} \text{Real} &= \text{real}_\perp & \text{Boolean} &= \text{bool}_\perp \\ \text{String} &= \text{string}_\perp & \texttt{OclAny}_\alpha &= \alpha_\perp \end{aligned}$$

combined into the universe $\mathscr{U}_\alpha$:

$$\mathscr{U}_\alpha = \text{Real} + \text{Boolean} + \text{String} + \texttt{OclAny}_\alpha$$

Note that the α-extensions are all extended by $\perp$ elements. Thus, there is a uniform way to denote "closed" objects, i.e. objects whose potential extension is not used. As a consequence, it is possible to determine the *dynamic type of an object* (its runtime type in a concrete system state) by testing for closing $\perp$'s. For example, the `OclAny` type has exactly one object represented by $Inr(Inr(Inr\,\perp))$ in any universe $\mathscr{U}_\alpha$. Thus, for each class C a test function isC can be generated that determines the dynamic type of an object. Note that all user-defined classes are subtypes of `OclAny` which is represented by the fact that an extension of the class hierarchy leads to a universe that is instance of $\mathscr{U}_\alpha$.

3.3.2 Outlining the Coding Scheme: Types and Constructors OCL types were mapped to types in HOL by the conversion function $\widehat{\tau}$; with respect to the basic type constructors Real, Boolean, String introduced in the previous section and the constructors `Set`, `Sequence` and `Bag` to be introduced in the subsequent Sec. 3.4.4, this mapping is one-to-one. User-defined class types, however, were mapped to the abstract type ref representing *object identifiers* (i.e. references) in the sense of OCL. Note that the resulting type discipline described so far is more liberal than OCL, where types of user-defined classes only match if the corresponding classes are in subclass relation. We will refine this in the second layer.

Now, assume a user-defined class T with previously defined superclass S with the attributes $t_1 : \tau_1, \ldots, t_n : \tau_n$. The types $T_{\text{base}} = \widehat{\tau_1} \times \cdots \times \widehat{\tau_n}$ and $T = S \times T_{\text{base}}$ are introduced, whereas T also contains the inherited attributes. By a filling of the α-variable of S or the β-variables of the chain of sons by the record T_{base}, the new universe $\mathscr{U}^{x+1}$ is constructed from $\mathscr{U}^{x}$.

On this basis, a constructor $mkT : T \to \mathscr{U}^{x+1}$ is generated, which embeds a record of type T into the actual version of the universe (e.g. $mkBoolean$: Boolean $\to \mathscr{U}_\alpha$, defined by $mkBoolean \equiv Inr \circ Inl$). Accordingly, a test $isT : \mathscr{U}^{x+1} \to$ bool, checking the dynamic type, and an accessor $getT : \mathscr{U}^{x+1} \to T$, representing the corresponding projection, are generated. And a definition for a constant $T : \text{set}(\mathscr{U}^{x+1})$ by its *characteristic set* of T, i.e. $\{x :: \mathscr{U}^{x+1} |\ isT(x)\}$ is generated. Note that since $isT(x) \to isS(x)$, the subtype relation can be expressed predicatively on the level of characteristic sets: $T \subset S$.

Data invariants I are represented by making the constructor partial with respect to I, i.e. the constructor will be defined only for input tuples T that fulfill I.

3.3.3 System States and System Transitions We turn now to the construction of the standard context in OCL, namely state transition pairs over the universe of semantic objects. All accessor functions and their compositions forming OCL's path expressions can be interpreted in this context (see next section). The state is defined as a partial mapping from object identities ref to objects in a universe, and "state transitions" as pairs over states:

$$\begin{aligned} \textbf{types}\ \ & state(\alpha) = \text{ref} \to \alpha\ \text{option} \\ & st(\alpha) = state(\alpha) \times state(\alpha) \end{aligned}$$

Standard contexts motivate a particular form of context lifted types:

$$V_\alpha(\tau) = VAL_{state(\mathscr{U}_\alpha) \times state(\mathscr{U}_\alpha)}(\tau)$$

Thus, all expressions with OCL type τ will be represented by an *HOL-OCL* expression of type $VAL_\alpha(\widehat{\tau})$; e.g. all *logical* *HOL-OCL* expressions have the type VAL_α(Boolean).

3.3.4 Outlining the Coding Scheme: Accessors Now we define the accessor functions with the HOL types that we give each translated OCL-expression. We introduce the type mapping $\widetilde{\tau}$, which is similar to $\widehat{\tau}$ but replaces all ref-occurrences by $\mathscr{U}_x$ (where x represent the current state of the universe). An OCL type `Set`(A) will then be approximated by $set(\mathscr{U}_x)$. Let $\widehat{\tau_1} \times \cdots \times \widehat{\tau_n}$ be the type of an OCL constructor or function, than the representation in HOL will have the type $V_\alpha(\widetilde{\tau_1}) \times \cdots \times V_\alpha(\widetilde{\tau_n})$. We will close the gap between the approximative HOL types for references of user-defined classes by introducing *type predicates* T`->includes`(x) (see Sec. 3.5 for details), where violations of these type predicates were treated as undefinedness; hence, on the level of the logics we do not distinguish in our model if a reference is simply undefined in the store or is referencing an object of wrong type. Thus, the logical overhead for representing types as predicates is reduced to undefinedness reasoning that is unavoidable in OCL anyway.

In a simple example, an accessor $obj.t : T$ in an object $obj.C$ must have the HOL type $\mathscr{U}_x \rightarrow VAL_x(T)$. This is ensured by the following construction: first we get the reference to obj with $getC$. If the reference is not $\bot$ (tested via upCase) we project the required attribute (by its position in the record structure, e.g. via snd). If the attribute is valid in the current state (which can be similarly tested via optionCase as states are maps) we dereference (using image for a single object) and lift it to get an object. Outermost we have an additional abstraction to meet the required type.

If the class has a class invariant, a test for the invariant must be added (violations are considered as undefinedness). If the accessor yields an OCL type with references to other classes (e.g. in `Set`(A)), these references must be accessed and inserted into the surrounding collection; this may involve smashing (see the discussion of collection types in Sec. 3.4.4).

Following this extended code scheme, we can define conservatively new accessors over some extended universe whenever we extend a class hierarchy; note that our technique enables for mutual data recursion that is introduced by extending class hierarchies while maintaining as much static type-checking as possible.

3.3.5 Encoding the Running Example To encode our earlier presented UML model (see Fig. 1), we declare the type for the class Account

(abbreviated by A). Due to space reasons, we skip InterestAccount (abbreviated by IA) and Bank (abbreviated by B). An account is a tuple describing the balance (of type Monetary) and the encoded association end *bank* (of type set(ref)). For our first universe, with the two "holes" α and β, we define:

$$\begin{aligned} \textbf{types} \quad \text{AccountType} &= \text{Monetary} \times \text{set(ref)} \\ \mathsf{Account} &= \text{AccountType}_{\perp} \\ \mathscr{U}^1_{(\alpha^{\mathsf{A}},\beta^{\mathtt{OclAny}})} &= \mathscr{U}_{(\text{AccountType}\times\alpha^{\mathsf{A}}_{\perp}+\beta^{\mathtt{OclAny}})} \end{aligned}$$

We need the raw constructor for an account object. Note that this function "lives" in the universe $\mathscr{U}^3_{(\alpha^{\mathsf{IA}},\beta^{\mathsf{A}},\alpha^{\mathsf{B}},\beta^{\mathtt{OclAny}})}$ which contains *all* classes from Fig. 1.

$$\begin{aligned} &mkAccount : \text{AccountType} \to \mathscr{U}^3_{(\alpha^{\mathsf{IA}},\beta^{\mathsf{A}},\alpha^{\mathsf{B}},\beta^{\mathtt{OclAny}})} \\ &mkAccount \equiv mkOclAny \circ \lfloor\_\rfloor \circ Inl \circ \lambda x.(x, \perp) \end{aligned}$$

For defining the accessor, we strictly follow the description given in the previous section, e.g. for the association end *bank* (of type Bank) in the current state s':

$$\begin{aligned} .bank : \mathscr{U}^3_{(\alpha^{\mathsf{IA}},\beta^{\mathsf{A}},\alpha^{\mathsf{B}},\beta^{\mathtt{OclAny}})} &\to VAL_{(\alpha^{\mathsf{IA}},\beta^{\mathsf{A}},\alpha^{\mathsf{B}},\beta^{\mathtt{OclAny}})}(\text{Bank}) \\ obj.bank \equiv \lambda(s,s').\,\text{upCase}(\lfloor\_\rfloor \circ (\text{image}(\lambda x.\,\text{optionCase}\perp \\ getBank(s'\ x))) \circ \text{snd})(\perp)(getAccount(obj)) \end{aligned}$$

Where snd selects the second attribute (*bank*) of the class Account. Analogously, we define the association end $.bank@\mathtt{pre}$ for accessing the value in the previous state s:

$$\begin{aligned} .bank@\mathtt{pre} : \mathscr{U}^3_{(\alpha^{\mathsf{IA}},\beta^{\mathsf{A}},\alpha^{\mathsf{B}},\beta^{\mathtt{OclAny}})} &\to VAL_{(\alpha^{\mathsf{IA}},\beta^{\mathsf{A}},\alpha^{\mathsf{B}},\beta^{\mathtt{OclAny}})}(\text{Bank}) \\ obj.bank@\mathtt{pre} \equiv \lambda(s,s').\,\text{upCase}(\lfloor\_\rfloor \circ (\text{image}(\lambda x.\,\text{optionCase}\perp \\ getBank(s\ x))) \circ \text{snd})(\perp)(getAccount(obj)) \end{aligned}$$

The two definitions differ only in the states used within *getBank*.

3.3.6 Implementation Details For HOL-OCL, we implemented a *class loader*, i.e. a compiler, which loads a description of a class diagram and generates a theory containing declarations and definitions for accessors, types and characteristic sets, including the infrastructure for method invocations (see Sec. 3.5) and theorems about definedness and membership in characteristic sets, etc. Our class-loader handles the extension of an existing class hierarchy with new classes and also expands *associations* and other UML constructs. At present, our class

diagrams are stored in a Standard ML based syntax. A parser for reading files in a simple text-format that is also used by the USE tool [46] and a deeper integration into standard CASE tools (e.g. ArgoUML [1]) will be provided in future.

3.4 Semantics for OCL Expressions

We proceed by defining OCL expression semantics roughly following the concrete syntax of OCL as presented in Tab. 1.

3.4.1 System State Access Operations Based on state transitions, we define the only form of universal quantification of OCL: the operator `allInstances` extracts all objects of a "type" (represented by its characteristic set) from the current state. The standard specifies `allInstances` as being undefined for `Integer` or `Real` or `String` in order to avoid infinite sets in an ad-hoc manner; a compliant definition following the intention of [56] looks as follows:

$$\begin{aligned}\texttt{allInstances} &: \mathrm{set}(\mathscr{U}_\alpha) \to V_\alpha(\texttt{Set}(\beta))\\ \texttt{allInstances}\, T \equiv \lambda(s,s').\, &\mathbf{if}(T = \mathrm{Integer} \vee T = \mathrm{Real} \vee T = \mathrm{String})\\ &\quad \mathbf{then} \perp \mathbf{else}\, \mathbf{if}(T = \mathrm{Boolean})\\ &\quad\quad \mathbf{then} \lfloor \mathrm{Boolean} \rfloor\, \mathbf{else} \lfloor T \cap (\mathrm{ran}\, s') \rfloor\end{aligned}$$

Note that $\mathrm{set}(\tau)$ denotes the constructor of the HOL type set, whereas $\texttt{Set}(\tau')$ constructs a OCL set (to be defined in Sec. 3.4.4).

However, we propose a different version $\texttt{allInstances}'()$ which omits the ad-hoc constraints and returns the (infinite) characteristic sets of types (see Sec. 3.3.2): $\texttt{allInstances}'()$ is just the identity. Defining OCL operators like $\texttt{oclIsNew} : \mathscr{U}_\alpha \to V_\alpha(\mathrm{Boolean})$ or `oclIsTypeOf` is now routine; the former checks if an object is defined in the current but not in the previous state, the latter redefines isT.

3.4.2 Equalities Defining an equality for *HOL-OCL* in combinator style is now easy. There are two possibilities: a logical equality called *strong equality* ($\triangleq$) and a strict version of it called *weak equality* ($\doteq$) that is executable and therefore the default in [56]:

$$\begin{aligned}\triangleq &\equiv \mathit{lift}_2(\lambda xy. \lfloor x = y \rfloor)\\ \doteq &\equiv \mathit{lift}_2(\mathit{strict}(\lambda x.\, \mathit{strict}(\lambda y. \lfloor x = y \rfloor)))\end{aligned}$$

The OCL syntax uses `=` for the weak equality; we propose to extended the OCL syntax by using `==` for the strong equality. Since the predicate

for definedness is part of the OCL logic (see Sec. 3.4.3) one of these equalities may be alternatively defined as a shortcut in OCL itself.

Another issue to be raised here is the semantics of equality; are two objects equal only if their object identifier is equal or are two objects equal if their values are equal? The OCL semantics is not specific here since equality is defined as equality over values [56, Def. 5.2.2], and since objects are values, but object identifiers are not distinguished from object values [56, (Def. 5.1.2.1)]. The definition of $\triangleq$ and $\doteq$ above results in shallow value equality (cf. [5]). However, since many object-oriented programming languages are centered around referential equality (which gave the motivation to opt for the latter in [2]), *HOL-OCL* can be configured such that the above definition leads to referential equality. The trick is done by adding an extra field in each object which contains its own unique object identifier; this can be assured by specific constructors that have access to the reference management in the system state.

3.4.3 Logical Operators We turn now to a key section of the OCL semantics: the SKL logics. According to the OCL standard (which follows SPECTRUM here), the logic operators have to be defined as Kleene-Logic, requiring that any logical operator reduces to a defined logical value whenever possible.

In itself, the logic is completely independent from an underlying concept of contexts which is also reflected by their types. An OCL formula is a either true, false or undefined depending on its underlying context. Logical expressions are just special cases of OCL expressions and must produce Boolean values. Consequently, the general type of logical formulae is:

$$\textbf{types} \quad \texttt{Boolean}_\alpha = \mathit{VAL}_\alpha(\text{Boolean})$$

The logical constants `true` and `false` can be defined as constant functions that yield the lifted value for meta-logical undefinedness, truth or falsehood, i.e. the HOL values of the HOL type bool. Moreover, the predicate `def` checks for any OCL expression X whether its value (evaluated in the context c) is defined or not.

$$\begin{aligned}
\texttt{def} &: \mathit{VAL}_\alpha(\beta_\perp) \to \texttt{Boolean}_\alpha & \texttt{def}(X) &\equiv \lambda c.\lfloor \mathit{DEF}(X\ c)\rfloor \\
\not\perp_{\mathscr{L}} &: \texttt{Boolean}_\alpha & \not\perp_{\mathscr{L}} &\equiv \mathit{lift}_0(\perp) \\
\texttt{true} &: \texttt{Boolean}_\alpha & \texttt{true} &\equiv \mathit{lift}_0(\lfloor \text{True}\rfloor) \\
\texttt{false} &: \texttt{Boolean}_\alpha & \texttt{false} &\equiv \mathit{lift}_0(\lfloor \text{False}\rfloor)
\end{aligned}$$

We indicate the undefined constant with $\not\perp_{\mathscr{L}}$ whereas `OclUndefined` of type `OclVoid` is used in the OCL standard. Further note that

def	
true	true
false	true
$\not\perp$	false

not	
true	false
false	true
$\not\perp$	$\not\perp$

and	false	$\not\perp$	true
false	false	false	false
$\not\perp$	false	$\not\perp$	$\not\perp$
true	false	$\not\perp$	true

or	false	$\not\perp$	true
false	false	$\not\perp$	true
$\not\perp$	$\not\perp$	$\not\perp$	true
true	true	true	true

Table 2 Truth tables for the OCL operations `def`, `not`, `and`, and `or`.

actual versions of OCL [56] define operations for checking, if a value x is defined or not, e.g. $\texttt{OclIsUndefined}(x)$ evaluates to `true` if x is undefined and `false` otherwise. Thus, we provide two operations, $\texttt{OclIsUndefined}(x)$ and $\texttt{OclIsDefined}(x)$ with $\texttt{OclIsUndefined}(x) \equiv \texttt{not}(\texttt{OclIsDefined}(x))$.

Defining the strict $\texttt{not} : \texttt{Boolean}_\alpha \to \texttt{Boolean}_\alpha$ and the non-strict $\texttt{and} : [\texttt{Boolean}_\alpha, \texttt{Boolean}_\alpha] \to \texttt{Boolean}_\alpha$ is now straight-forward:

$$\begin{aligned}
\texttt{not} &\equiv \mathit{lift}_1(\mathit{strict}(\lfloor_\rfloor \circ \neg \circ \lceil_\rceil)) \\
S \texttt{ and } T &\equiv \lambda c.\, \mathbf{if}\ \mathit{DEF}(S\ c)\ \mathbf{then}\ \mathbf{if}\ \mathit{DEF}(T\ c)\ \mathbf{then} \lfloor \lceil S\ c \rceil \wedge \lceil T\ c \rceil \rfloor\ \mathbf{else} \\
&\qquad \mathbf{if}(S\ c = \lfloor \mathrm{False} \rfloor)\ \mathbf{then} \lfloor \mathrm{False} \rfloor\ \mathbf{else} \perp\ \mathbf{else} \\
&\qquad\qquad \mathbf{if}(T\ c = \lfloor \mathrm{False} \rfloor)\ \mathbf{then} \lfloor \mathrm{False} \rfloor\ \mathbf{else} \perp
\end{aligned}$$

From these definitions, the elementary reduction rules (*R-rules*) of the truth table were derived (see Tab. 2 for details). These tables correspond exactly to the definitions in the OCL standard. The logical connectives `or`, `xor`, and `implies` can be defined as usual: $S \texttt{ or } T \equiv \texttt{not}((\texttt{not}\ S) \texttt{ and } (\texttt{not}\ T))$, $S \texttt{ and } T \equiv (S \texttt{ or } T) \texttt{ and not}(S \texttt{ and } T)$, and $S \texttt{ implies } T \equiv (\texttt{not}\ S) \texttt{ or } T$. In OCL, quantifiers are considered as operations on collections and will therefore be discussed in the sequel.

3.4.4 Expressions: Standard Operations of the Library Beside the logic, the OCL standard defines a library of data types consisting of the basic data types (`Integer`, `Real` and `String`), special types (`OclAny`, `OclVoid`). The construction of them is straight-forward; the paradigm of these definitions has already been presented in Sec.3.2; an automatic technique for deriving standard theorems such as commutativity and associativity has been described in [12].

The OCL library considers `Set`, `Sequence` and `Bag` as subclasses of the abstract class `Collection`. We focus on `Set` in the sequel; the other cases are analogous. OCL [56, Sec. A.2.5.2] allows for collections to include $\not\perp$, i.e. the constructor of sets and the membership tests are non-strict. This has several undesired consequences for executabil-

ity[3]. Instead of a standard compliant version of sets `Set(A)`, we will therefore present *smashed sets* `SSet`(A) throughout this paper which can be directly implemented in e.g. Java; the derivation of our compliant version `Set(A)` is straight-forward by leaving out the *smashing combinators* $smash, Rep_{SSet}$, and Abs_{SSet} in the definitions.

Smashing data-structures is a key-concept in denotational semantics [34,57]. For example, pairs are smashed if $(a, \bot)$ is identified with $\bot$ as in e.g. Java or SML, or, with respect to sets, $\{a, \bot\} = \bot$. Formally in OCL, smashing is introduced by a smashing operator inducing a quotient construction over lifted sets. The smashing operator yields for any data-construction $\bot$ whenever it "contains" $\bot$:

$$\begin{array}{ll} \textbf{constdefs} & smash :: [[\beta :: \text{bot}, \alpha :: \text{bot}] \to \text{bool}, \alpha] \to \alpha \\ & smash\ contain\ X \equiv \textbf{if}\ contain\ \bot\ X\ \textbf{then}\ \bot\ \textbf{else}\ X \end{array}$$

For the case of sets, the containment relation is instantiated with $\check{\in}$ on lifted sets $\text{set}(\alpha)_\bot$. We build the type `SSet`(X) as the set of all smashed data, i.e. on all data where *smash* is the identity:

$$\textbf{typedef}\quad \texttt{SSet}(\alpha) = \{X :: \text{set}(\alpha :: \text{bot})_\bot .\ smash\, \check{\in} X = X\}$$

where $x \check{\in} X \equiv_{DEF} X \wedge x \in \lceil X \rceil$. Note that $\alpha :: \text{bot}$ is the class of all types that possess a $\bot$-element. A type definition is a conservative extension scheme (cf. Sec. 2.3), where a bijection between the smashed sets over α and the type `SSet`(α) is stated. In more detail, the two constants Abs_{SSet} of type $\text{set}(\alpha)_\bot \to$ `SSet`(α) and Rep_{SSet} of type `SSet`$(\alpha) \to \text{set}(\alpha)_\bot$ and the bijection axioms are postulated:

$$\begin{array}{lll} smash\, \check{\in} y = y & \Longrightarrow & \text{Rep}_{\text{SSet}}(\text{Abs}_{\text{SSet}}\, y) = y \\ & & \text{Abs}_{\text{SSet}}(\text{Rep}_{\text{SSet}}\, x) = x \end{array}$$

Apart from smashing sets, the definitions of set operations such as `includes`, `union` or `intersection` follow the usual pattern, e.g.:

$$\begin{array}{rl} \texttt{->includes} \equiv & lift_2(strict(\lambda X.\ strict(\lambda x. \lfloor x : \lceil \text{Rep}_{\text{SSet}}\, X \rceil \rfloor))) \\ \texttt{->union} \equiv & lift_2(strict(\lambda X.\ strict(\lambda Y. \\ & \quad \text{Abs}_{\text{SSet}}(\lfloor \lceil \text{Rep}_{\text{SSet}}\, X \rceil \cup \lceil \text{Rep}_{\text{SSet}}\, Y \rceil \rfloor))))) \end{array}$$

[3] So far, $\lightning$ can be interpreted as exception *and* non-termination of recursion; the language except the `def` and $\triangleq$-construct is still executable in this interpretation. The standard's definition of non-strict sets rules out all set operators in this setting as executable operations.

For the quantifiers `forall` and `exists` we follow the definitions for SKL already presented in [29]. They have the type $(VAL_\alpha(\texttt{SSet}(\beta)) \to \texttt{Boolean}_\alpha) \to \texttt{Boolean}_\alpha$ and are represented as follows:

$$
\begin{aligned}
S\texttt{->forall(}P\texttt{)} \equiv \lambda\, st\,.\, &\textbf{if}\ DEF(S\ \ st)\ \textbf{then}\\
&\quad \textbf{if}\ \forall x : \lceil \mathrm{Rep}_{\mathrm{SSet}}(S\ \ st)\rceil.P(\lambda\, st\,.x\ st = \texttt{true})\\
&\quad \textbf{then}\ \texttt{true}\\
&\quad \textbf{else if}\ \exists x : \lceil \mathrm{Rep}_{\mathrm{SSet}}(S\ \ st)\rceil.P(\lambda\, st\,.x\ st = \texttt{false})\\
&\qquad \textbf{then}\ \texttt{false}\\
&\qquad \textbf{else}\ \not\perp_{\mathscr{L}}\\
&\quad \textbf{else}\ \not\perp_{\mathscr{L}}\\
S\texttt{->exists(}P\texttt{)} \equiv\ &\texttt{not(}S\texttt{->forall(}\lambda x.\,\texttt{not(}P(x)\texttt{))}
\end{aligned}
$$

This definition deviates from the standard for three reasons: first, the standard's definition based on `->iterate`, a foldr-like algorithm on collection types is not necessarily conservative[4]. Second, we intended to provide a definition for infinite sets, such that e.g. types can be handled appropriately. Third, since $X\,\texttt{->includes}\,(x)$ implies $\texttt{def}(x)$, OCL-calculi are greatly simplified. Throughout this paper, we will therefore assume smashed semantics for all collection types.

3.5 Operation Specifications vs. Method Invocation

In the previous sections, we described the semantics of built-in operators or library methods. We turn now to user-defined methods and their invocation. The semantics of an operation specification:

$$[\![\texttt{context}\ C :: \mathrm{op}(p_1 : T_1, \ldots, p_n : T_n) : T_{n+1}\ \texttt{pre} : P\ \texttt{post} : Q]\!] = R$$

is defined as relation on states (σ, σ'); making syntactic side-conditions explicit the following definition conforms to [56]:

$$
\begin{aligned}
&R\ \texttt{self}\ p_1 \ldots p_n\ \texttt{result} \equiv \{(\sigma,\sigma')\mid\\
&\quad (\sigma,\sigma') \models T'_0\ \texttt{->includes}\ (\texttt{self})\\
&\wedge (\sigma,\sigma') \models T'_1\ \texttt{->includes}\ (p_1) \wedge \cdots\\
&\wedge (\sigma,\sigma') \models T'_n\ \texttt{->includes}\ (p_n)\\
&\wedge (\sigma,\sigma') \models T'_{n+1}\ \texttt{->includes}\ (\texttt{result})\\
&\wedge (\sigma,\sigma') \models P\ \texttt{self}\ p_1 \ldots p_n \wedge (\sigma,\sigma') \models Q\ \texttt{self}\ p_1 \ldots p_n\ \texttt{result}\}
\end{aligned}
$$

[4] It has to be shown that its instance is associative, commutative and idempotent in order to be well-defined.

where $(\sigma, \sigma') \models A \equiv I[\![A]\!](\sigma, \sigma') = \lfloor \mathrm{True} \rfloor$ and `self` is the usual argument *referring to the contextual instance* [56] and `result` the return value of a method.

Since the type discipline in *HOL-OCL* is too coarse so far, we introduce *type predicates* T' `->includes` (p_i) in order to close the gap between the *HOL-OCL* and the OCL type discipline. We convert a type T_i to $T_i' \equiv \mathit{lift}_0(\overline{T_i})$ with the *conversion function* $\overline{\tau}$. For its definition, three cases have to be distinguished:

1. *Basic Types* such as OCL type `String` are mapped to the constant `String` $\equiv \lfloor \mathrm{Abs}_{\mathrm{Set}}(\{x :: \mathrm{String} \mid \mathrm{True}\}) \rfloor$ (`Integer`, `Real`, etc. are treated analogously)
2. *User-defined Class Types* were represented by their characteristic sets (cf. Sec. 3.3.2)
3. *Collection Type Cases* such as $\overline{\mathtt{Set}(X)}$ are converted to $\mathtt{Set}(\overline{X})$ where the constant function $\mathtt{Set}(Y)$ is defined as the set of all subsets of Y, i.e. $\lfloor \mathrm{Abs}_{\mathrm{Set}}\{y \mid \lceil y \rceil \subseteq \lceil Y \rceil\} \rfloor$ (`Sequence`, `Bag`, etc. are treated analogously).

Note that, by construction, T_i' has HOL-type $\mathit{VAL}_\alpha(\mathtt{Set}(\widetilde{T_i}))$ (cf. Sec. 3.3.4). Further note, that since $x :: \mathrm{String} \in \{x :: \mathrm{String} \mid \mathrm{True}\}$ holds trivially, for all types constructed over OCL base types and OCL collection types, the type predicates T_i' `->includes` (x_i) are trivially true and can be reduced automatically to `def`(x_i) based on the results of the HOL type check. Thus, in these common cases, type predicates do not represent an obstacle for deduction; only when subtyping between class types is involved, reasoning over them may be necessary.

In *HOL-OCL*, we use a slightly different formulation of R which is both technically more elegant and conceptually more powerful:

$$
\begin{aligned}
R'\ \mathtt{self}\ p_1 \ldots p_n\ \mathtt{result} \equiv\ & \\
& T_0'\ \mathtt{->includes}\ (\mathtt{self})\ \underline{\mathtt{and}}\ T_{n+1}'\ \mathtt{->includes}\ (\mathtt{result}) \\
\underline{\mathtt{and}}\ & T_q'\ \mathtt{->includes}\ (p_1)\ \underline{\mathtt{and}} \cdots \underline{\mathtt{and}}\ T_n'\ \mathtt{->includes}\ (p_n) \\
\underline{\mathtt{and}}\ & P\ \mathtt{self}\ p_1 \ldots p_n\ \underline{\mathtt{and}}\ Q\ \mathtt{self}\ p_1 \ldots p_n\ \mathtt{result}
\end{aligned}
$$

where $\underline{\mathtt{and}}$ is the *strict* logical conjunction. Instead of a relation on states (i.e. a set of state pairs isomorphic to a function of state-pairs to bool), R' is defined as a function mapping state pairs to Boolean. A state transition may thus be mapped to $\lfloor \mathrm{True} \rfloor$, $\lfloor \mathrm{False} \rfloor$ or $\bot$ which we interpret as *possible*, *impossible* and *unknown* transitions, who are not restricted by the method specification. Distinguishing impossible from undefined transitions paves the way for more powerful refinement notions for OCL specifications. This distinction turns R' simply

into a richer structure; erasing it transforms R' into R again as the obvious theorem reveals:

$$R\,\texttt{self}\,p_1 \ldots p_n\,\texttt{result} = \big\{(\sigma,\sigma')|(\sigma,\sigma') \models R'\,\texttt{self}\,p_1 \ldots p_n\,\texttt{result}\big\}$$

We now turn to the counterpart of method specification: invocation. OCL is intended to be call-by-value language in general. Beyond that, the semantics of OCL is deliberately underspecified in several regards: first, OCL provides only a method specification construct which means a relation between input and output states, arguments and results — some concrete function has to be chosen arbitrarily that fits to this relation. Second, in the presence of overloading, it is: "the programming language may choose an arbitrary overloaded method that matches", which is combined with the pragmatic guideline to follow Liskov's principle [31]. We believe that the following definition scheme follows the intention of the standard. For each method m, we assume an *invoke*-operation defined as:

$$\begin{aligned}[\![X.m(p_1,\ldots,p_n)]\!] = \mathbf{if}\,(\texttt{def}(X)\ \texttt{and}\ \texttt{def}(p_1)\ \texttt{and}\ldots\texttt{and}\ \texttt{def}(p_n))\\ \mathbf{then}\ invoke_m\ X\ p_1\ \ldots\ p_n\ \mathbf{else}\ \not\perp_{\mathscr{C}}\end{aligned}$$

where

$$\begin{aligned}&invoke_m\ X\ p_1 \ldots p_n\ (s,s') \equiv\\ &\qquad \varepsilon\texttt{result}.(s,s') \models (choose_m X)\ X\ p_1 \ldots p_n\ \texttt{result}\\ &choose_m(X)(s,s') \equiv ovltab_m(\varepsilon CS \in \mathrm{dom}(ovltab_m).X(s,s') \in CS)\end{aligned}$$

Here, HOL's Hilbert operator (c.f. 2.3) is used to model both the underspecification of the invocation as well as the non-determinism of an implementation wrt. its method specification. The overloading table $ovltab_m$ is a finite map that associates to a particular type (represented by its characteristic set) the method specification R'.

3.6 Possible Extensions

Our semantics has been constructed in a modular way. We will exploit this modularity by discussing modifications or extensions.

3.6.1 Alternative Logical Connectives We consider the introduction of strict versions of the logical connectors as useful — the introduction of a strict conjunction $\underline{\texttt{and}}$, for example, helps on the methodological level as well as on the level of code generation.

With respect to the logical implication in SKL, several possibilities have been investigated in the literature:

$$A \texttt{ implies } B \equiv \texttt{not } A \texttt{ or } B \tag{1a}$$

$$A \texttt{ implies}' B \equiv (\texttt{undef } A) \texttt{ or } (\texttt{not } A) \texttt{ or } B \tag{1b}$$

$$A \texttt{ implies}'' B \equiv (\texttt{not } A) \texttt{ or } (A \texttt{ and } B) \tag{1c}$$

For these variants of implies we derived the truth tables, their behavior differs only for undefined operands. The difference is that the variant (1b) evaluates to `true` if the assumption is undefined, while (1c) evaluates to $\not\perp_{\mathscr{L}}$. The former has been proposed for proof-theoretic reasons [21], while the latter was suggested as a consequence of methodological criticism [26]. While we see no clear advantage for the latter [9], the former will be discussed in more detail in Sec. 4.

3.6.2 Method Invocation and General Recursion. In our view, the underspecification of overload resolution and recursion is a major obstacle in the further development of semantic libraries (such as the *mathematical toolkit* of Z) for OCL, which hamper its use both in programming as well as theorem proving environments. The problem is best presented with an example, assume a method `fac` of class `p`:

```
context p.fac(i:Integer):Integer
pre:  true
post: return = if i >= 0 then 1 else i*self.fac(i-1)
```

The standard makes no clear statement how to interpret recursive function: is it an illegal statement (the recursion loops for negative arguments), or should it yield $\not\perp_{\mathscr{L}}$ which was the intention in the OCL Manifesto [14], or should it yield $\not\perp_{\mathscr{L}}$ only for negative arguments. In the presence of overloading, the situation is worse since the operational interpretation of a method invocation is underspecified; approximations to the semantics can only by done under the quite broad side-constraint "Liskov's principle will always be respected in any overloading" (albeit a formalization of this is possible in our framework, we believe it is highly unpractical).

With respect to overloading, we argue for choosing the method associated to the *least* characteristic set (see the extended version of [11]); this mirrors the semantics in object oriented languages such as Java and is more deterministic than the standard. Note that such a particular choice for the invocation semantics does not impose any constraint on the object oriented implementation language.

With respect to recursion semantics, we argue in favor of standard denotational fixpoint semantics as in [57]. Technically, this means that

the type class bot is extended to the class of complete partial orders (*cpo*), for which operationally approximately fixpoints exist. There are several HOL theories available that provide this type class extension and a suitable body of theorems, (see [35,51]) whose integration is straight-forward.

As an alternative to denotational recursion semantics, one could use well-founded recursion [57]; this would require additional syntax providing a measure function and a mechanism that assures that inner calls in a recursive definition were either applied to arguments with "smaller" measures or were $\bot_{\mathscr{C}}$.

3.6.3 Alternative Quantifiers. The operator `allInstances` is a projection into a finite state and thus always restricted to a finite set or sequence. For infinite types such as integers or strings, the operator is defined to be undefined. From the point of view of a higher-order logic proof environment for OCL, it is both highly desirable and feasible to omit these restrictions, which makes algebraic laws like

```
Integer.allInstances‘().forall(x,y | x + y == y + x)
```

representable in *HOL-OCL*. Moreover, it is also desirable and straightforward to introduce a second-order quantifier `allMethods`:

```
allMethods(m(p:Integer):Boolean |
      Integer.allInstances().forall(x, y, k |
         m(k) and forall(y | m(y) implies m(y + 1))
            implies (y >= k implies m(y))))
```

which represents a standard induction scheme over integers larger k.

Such a `allMethods`-construct would not only enable us to formulate induction schemes for the infinite types inside OCL, in connection with general recursion it also allows for a class to be constrained to an inductive data type such as trees — a necessity already pointed out in the OCL Manifesto [14]. Technically, this can be done by a function computing the "depth" of the data type and requiring in the class invariant that this function is always defined. Intuitively, this excludes the possibility of "cycles" in an object graph.

A highly interesting extension line of OCL are temporal quantifiers. Instead of considering just one state transition from one object graph to the next, it is also possible to admit *infinite sequences* or *traces* of states. In our definition of *HOL-OCL*, this would only require slight changes in the semantics of path-expressions; most semantic definitions in this paper refer to $VAL_\sigma(T)$ and not to $V_\sigma(T)$ and are therefore independent from the structure of the underlying contexts. A context may therefore be changed from a state transition $st(\mathscr{U}_\alpha)$ to a Kripke-structure nat $\rightarrow$ $state(\mathscr{U}_\alpha)$. Thus, *temporal quantifiers* such

as $\Box P$ (in all subsequent states, P holds) or $\Diamond P$ (there is a subsequent state where P holds) could be introduced into OCL as suggested by [18]. This extension of OCL makes it possible to annotate behavioral specifications in UML such as state-charts or sequence diagrams, or to define their semantics with temporal OCL.

4 A Proof Calculus for OCL

We will now develop several deduction systems for OCL. In particular, we define two equational calculi (UEC and LEC) usable for interactive proofs or proofs by hand, and a tableaux calculus (LTC) geared towards automatic reasoning in OCL. We will only present rules *derived* within Isabelle from the semantic definitions of the previous section. Therefore we can guarantee the logical soundness, with respect to the core logic, of all these rules. Finally, these three calculi were used to instantiate Isabelle's (two-valued) generic proof-procedures yielding decision procedures for certain OCL fragments.

4.1 Validity and Judgments

Since all OCL terms of OCL type T are represented as $I[\![X]\!] : \alpha \to T$ or just $X : VAL_\alpha(T)$, the HOL equality $=$ induces via $I[\![X]\!] = I[\![Y]\!]$ (or just $X = Y$) a congruence on OCL terms. We call this *universal congruence* on OCL since it means equality for two terms *for all contexts* due to the extensionality of the HOL equality; a *local congruence* is induced by $I[\![X]\!]\, st = I[\![Y]\!]\, st$ and makes reasoning over specific contexts possible, in particular over state transitions $st = (\sigma, \sigma')$ or classes thereof. With respect to formulae, i.e. OCL terms of type $VAL_\alpha(\text{Boolean})$, three cases are possible: $I[\![X]\!]\, st$ may yield $\lfloor \text{True} \rfloor$, $\lfloor \text{False} \rfloor$ or $\bot$. This leads to the following definitions of *local validity judgments* of an OCL formula:

$$st \models_{\mathsf{t}} X \equiv (X\ st = \lfloor \text{True} \rfloor)$$
$$st \models_{\mathsf{f}} X \equiv (X\ st = \lfloor \text{False} \rfloor)$$
$$st \models_{\mathsf{u}} X \equiv (X\ st = \lfloor \bot \rfloor)$$

We will use its semantically equivalent definitions:

$$st \models_{\mathsf{t}} X \equiv (X\ st = \mathtt{true}\ st)$$
$$st \models_{\mathsf{f}} X \equiv ((\mathtt{not}\ X)\ st = \mathtt{true}\ st)$$
$$st \models_{\mathsf{u}} X \equiv ((\mathtt{not}(\mathtt{def}(X)))\ st = \mathtt{true}\ st)$$

which have the advantage that $\models_{\mathsf{t}}$ subsumes $\models_{\mathsf{f}}$, and $\models_{\mathsf{f}}$ subsumes $\models_{\mathsf{u}}$. We say for $st \models_{\mathsf{t}} X$ that a context st is *(locally) valid* for formula X (or *locally invalid* or *locally undefined*, respectively). Note that $st \models_{\mathsf{t}} X$ is equivalent to the satisfaction notion $(\sigma_{\text{pre}}, \sigma_{\text{post}}) \models X$ of pre-conditions and post-conditions in the OCL standard [56, Def.5-32]. In the context of tableaux calculi for multi-valued logics (see Sec. 4.5.1), distinguishing these three validity judgments paves the way for labeled deduction systems [24] for OCL enabling to represent a three-valued logic in a two-valued format of rules.

First, let us point out that universal and local congruences are related via three "bridge theorems":

$$\frac{\bigwedge st.(st \models_{\mathsf{t}} X) = (st \models_{\mathsf{t}} Y) \quad \bigwedge st.(st \models_{\mathsf{f}} X) = (st \models_{\mathsf{f}} Y)}{X = Y} \tag{2a}$$

$$\frac{\bigwedge st.(st \models_{\mathsf{t}} X) = (st \models_{\mathsf{t}} Y) \quad \bigwedge st.(st \models_{\mathsf{u}} X) = (st \models_{\mathsf{u}} Y)}{X = Y} \tag{2b}$$

$$\frac{\bigwedge st.(st \models_{\mathsf{f}} X) = (st \models_{\mathsf{f}} Y) \quad \bigwedge st.(st \models_{\mathsf{u}} X) = (st \models_{\mathsf{u}} Y)}{X = Y} \tag{2c}$$

Note that since a validity statement like $st \models_{\mathsf{t}} X$ has the type bool in HOL, all equalities in the premises of these rules can be seen as logical equivalences $st \models_{\mathsf{t}} X \Leftrightarrow st \models_{\mathsf{t}} Y$; as such, they can be decomposed into implications from left to right and vice versa.

4.2 A Universal Equational Calculus for OCL

The basis of an universal equational calculus (UEC) for OCL are Horn-clauses over universal congruences; due to the rich algebraic structure of SKL, UEC allows for logical reasoning in formulae and local validity judgments. A proof of a formula in UEC is simply a derivation of a formula to `true`.

Based on the elementary reduction rules (R-rules) for the logical operators (see Sec. 3.4.3), it is not difficult to derive the laws of the surprisingly rich algebraic structure of Kleene-Logics: both `and` and `or` enjoy associativity, commutativity and idempotency. The logical operators also satisfy both distributivity and de Morgan laws. It is essentially this richness and algebraic simplicity that we will exploit in the applications (see Sec. 5).

$$\begin{array}{rr}
\texttt{false and } X = \texttt{false} & \texttt{not}(\texttt{not } X) = X \\
\texttt{true and } X = X & (X \texttt{ or } Y) \texttt{ and } Z = (X \texttt{ and } Z) \texttt{ or } (Y \texttt{ and } Z) \\
\texttt{true or } X = \texttt{true} & Z \texttt{ and } (X \texttt{ or } Y) = (Z \texttt{ and } X) \texttt{ or } (Z \texttt{ and } Y) \\
\texttt{false or } X = X & (X \texttt{ and } Y) \texttt{ or } Z = (X \texttt{ or } Z) \texttt{ and } (Y \texttt{ or } Z) \\
X \ op \ X = X & Z \texttt{ or } (X \texttt{ and } Y) = (Z \texttt{ or } X) \texttt{ and } (Z \texttt{ or } Y) \\
X \ op \ Y = Y \ op \ X & (X \texttt{ and } Y) = \texttt{not}(\texttt{not}(X) \texttt{ or not}(Y)) \\
X \ op \ (Y \ op \ Z) = (X \ op \ Y) \ op \ Z & \texttt{not}(X \texttt{ and } Y) = \texttt{not}(X) \texttt{ or not}(Y) \\
\text{where } op \in \{\texttt{and}, \texttt{or}\} & \texttt{not}(X \texttt{ or } Y) = \texttt{not}(X) \texttt{ and not}(Y)
\end{array}$$

(a) Lattice

$$\begin{array}{rr}
X \texttt{ implies false} = \texttt{not } X & X \texttt{ implies true} = \texttt{true} \\
\texttt{false implies } X = \texttt{true} & \texttt{true implies } X = X \\
\multicolumn{2}{c}{\texttt{def}(X) = \texttt{true} \Rightarrow (X \texttt{ implies } X) = \texttt{true}} \\
\multicolumn{2}{c}{X \texttt{ implies } (Y \texttt{ and } Z) = (X \texttt{ implies } Y) \texttt{ and } (X \texttt{ implies } Z)} \\
\multicolumn{2}{c}{X \texttt{ implies } (Y \texttt{ or } Z) = (X \texttt{ implies } Y) \texttt{ or } (X \texttt{ implies } Z)} \\
\multicolumn{2}{c}{(X \texttt{ and } Y) \texttt{ implies } Z = X \texttt{ implies } (Y \texttt{ implies } Z)} \\
\multicolumn{2}{c}{(X \texttt{ or } Y) \texttt{ implies } Z = (X \texttt{ implies } Z) \texttt{ and } (Y \texttt{ implies } Z)} \\
\multicolumn{2}{c}{X \texttt{ implies } (Y \texttt{ implies } Z) = Y \texttt{ implies } (X \texttt{ implies } Z)}
\end{array}$$

(b) Logic

$$\frac{P \not\perp_{\mathscr{C}} = P' \not\perp_{\mathscr{C}} \quad P \texttt{ true} = P' \texttt{ true} \quad P \texttt{ false} = P' \texttt{ false} \quad \text{cp}(P) \ \ \text{cp}(P')}{P \ X = P' \ X}$$

(c) Trichotomy

$$\frac{\texttt{def}(X) = \texttt{true} \quad \texttt{def}(Y) = \texttt{true}}{\texttt{def}(X \ op \ Y) = \texttt{true}}, \text{ where } op \in \{\texttt{def}, \texttt{not}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{implies}\}.$$

(d) Definedness(Fragment)

$$\frac{}{\text{cp}(\lambda X.X)} \qquad \frac{}{\text{cp}(\lambda X.C)} \qquad \frac{\text{cp}(P) \quad \text{cp}(P')}{\text{cp}(\lambda X.(P \ X) \ op \ (P' \ X))}$$

(e) Context Passing

Table 3 The Propositional Universal Equational Calculus (UEC)

The logical implication is also representable in this equational reasoning style, which is particularly intuitive and therefore greatly facilitates "by-hand-proofs", see Tab. 3(b); these rules form the core of the logical calculus. However, the crucial assumption rule ($\texttt{def}(X) = \texttt{true} \Rightarrow (X\,\texttt{implies}\,X) = \texttt{true}$) that allows one to deduce that a fact follows from a list of assumptions leads to a complication:

$$\begin{aligned}
&A_1\ \texttt{and}\cdots\texttt{and}\ A_k\ \texttt{and}\ B\ \texttt{and}\ A_{k+1}\ \texttt{and}\cdots\texttt{and}\ A_n\ \texttt{implies}\ B\\
=&A_1\ \texttt{and}\cdots\texttt{and}\ A_n\ \texttt{and}\ B\ \texttt{implies}\ B\\
=&A_1\ \texttt{and}\cdots\texttt{and}\ A_n\ \texttt{implies}\ (B\ \texttt{implies}\ B)\\
=&A_1\ \texttt{and}\cdots\texttt{and}\ A_n\ \texttt{implies}\ \texttt{true} \qquad [\texttt{def}(B) = \texttt{true}]\\
=&\texttt{true}
\end{aligned}$$

This means that a side-calculus for the definedness predicate is needed, moreover, this means that each application of the assumption rule leads to a subproof over the definedness of the assumption.

It is worth to consider an alternative definition of `implies` (1b) on page 25, which was investigated in [21]. The counterparts to rules presented in Tab. 3(b) holds for this version of implication, except two details:

1. We have $(X\ \texttt{implies}'\ X) = \texttt{true}$ such that the subproof for $\texttt{def}(B) = \texttt{true}$ is not necessary. The handling of this implication in proofs is therefore more intuitive.
2. However, there is no free lunch: the problem is only shifted to the "reductio ad absurdum"-rule $(X\ \texttt{implies}\ \texttt{false}) = \texttt{not}\ X$, whose counterpart requires now the proviso for definedness.

In classical logic, whenever we know that the conclusion does not hold, only a contradiction in the assumptions can make the implication valid. Results on the proof complexity [21] are therefore related to the concrete tableaux calculi and not to the definition of implies.

We turn now to rules that infer the definedness of an OCL-term; the general scheme for these inferences is presented in Tab. 3(d). Note, that even `not` and `if _ then _ else _ endif` follow this principle:

$$\frac{\texttt{def}(X) = \texttt{true} \quad \texttt{def}(Y) = \texttt{true} \quad \texttt{def}(Z) = \texttt{true}}{\texttt{def}(\texttt{if}\ X\ \texttt{then}\ Y\ \texttt{else}\ Z\ \texttt{endif}) = \texttt{true}}$$

$$\texttt{def}(X) = \texttt{true} \Rightarrow \texttt{def}(\texttt{not}\ X) = \texttt{true}$$

The rule Tab. 3(d) is completed by the canonical rule set:

$$\texttt{def}(\texttt{def}(X)) = \texttt{true}$$

$$\begin{aligned}
\texttt{def}(X\ \texttt{and}\ Y) &= (\texttt{def}(X)\ \texttt{and}\ \texttt{def}(Y))\ \texttt{or}\ (\texttt{def}(X)\ \texttt{and}\ \texttt{not}\ X)\\
&\quad\ \texttt{or}\ (\texttt{def}(Y)\ \texttt{and}\ \texttt{not}\ Y)\\
\texttt{def}(X\ \texttt{or}\ Y) &= (\texttt{def}(X)\ \texttt{and}\ \texttt{def}(Y))\\
&\quad\ \texttt{or}\ (\texttt{def}(X)\ \texttt{and}\ X)\ \texttt{or}\ (\texttt{def}(Y)\ \texttt{and}\ Y)\\
\texttt{def}(X\ \texttt{xor}\ Y) &= \texttt{def}(X)\ \texttt{and}\ \texttt{def}(Y)\\
\texttt{def}(X\ \texttt{implies}\ Y) &= (\texttt{def}(X)\ \texttt{and}\ \texttt{def}(Y))\\
&\quad\ \texttt{or}\ (\texttt{def}(X)\ \texttt{and}\ \texttt{not}\ X)\ \texttt{or}\ (\texttt{def}(Y)\ \texttt{and}\ Y)
\end{aligned}$$

$$\begin{aligned}
&\texttt{def}(\texttt{if}\ X\ \texttt{then}\ Y\ \texttt{else}\ Z\ \texttt{endif}) =\\
&\texttt{def}(X)\ \texttt{and}\ ((X\ \texttt{and}\ \texttt{def}(Y))\ \texttt{or}\ (\texttt{not}\ X\ \texttt{and}\ \texttt{def}(Z)))
\end{aligned}$$

4.2.1 A Further Proof Principle of LEC*: Trichotomy.* An interesting technique for proving $P\ X = P'\ X$ is based on a case split over $\not\perp_{\mathscr{L}}$, `true` or `false`. The enabling rule Tab. 3(c) is called *trichotomy*; it requires a particular constraint over the treatment of the implicit context *st* inside P and P'. In principle, it would suffice to require that *st* is changed "on its way through P and P'" in the same way. However, since all OCL constructs including the logical connectives are lifted over the contexts (see Sec. 3.1), we apply a slightly stronger restriction, namely that *st* is unchanged, i.e. P or P' are *context passing* with respect to *st*. It turns out that this concept is necessary for other calculi, too. Formally, we can define context passing $\mathrm{cp}(P)$:

$$\mathrm{cp}(P) \equiv \exists E.\forall X\ st.P\ X\ st = E(X\ st)$$

Because all operators constructed by the lifting combinators $lift_0$, $lift_1$ and $lift_2$ are context passing, all OCL operators enjoy the invariance properties in Tab. 3(e). Thus, the property of being context passing can easily inferred in a backward proof whose size is equal to the size of the term. These inferences use inherently higher-order concepts.

4.2.2 Completeness of UEC

Definition 1. *The* propositional fragment of OCL *is the inductively defined set of* OCL*-formulae (i.e. the set of* OCL*-terms of type* $\texttt{Boolean}_\alpha$*) that is built over variables,* `true`*,* `false`*,* $\not\perp_{\mathscr{L}}$*,* `def`*,* `undef`*,* `not`*,* `and`*,* `or`*,* `xor`*,* `implies`*, and* `if _ then _ else _ endif`*.*

Definition 2. *A proof in* UEC *is a derivation of a formula of the form* $A = C$ *with* $C \in \{\texttt{true}, \texttt{false}, \not\perp_{\mathscr{L}}\}$*.*

Definition 3 (Completeness with respect to a rule set S). *If $A = C$ is a valid HOL formula, then there is a derivation applying reflexivity, transitivity, and substitutivity of $=$ together with all R-rules and rule set S.*

Theorem 1. *UEC is complete with respect to UEC for propositional HOL-OCL.*

Proof. Let $A = C$ be a valid proof statement with $C \in \{\texttt{true}, \texttt{false}, \not\perp\!\!\!\not\perp\}$. By induction over the number of variables occurring in A, we show that there is a finite reduction to $C = C$.

Basis: If no variables occur, i.e. $|\mathit{vars}(A)| = 0$, then we only apply R-rules. Since this rule set yields a canonical rewrite system, the thesis holds.

Step: Assume that the thesis holds for all terms B with $|\mathit{vars}(B)| < |\mathit{vars}(A)|$. Then we show that the thesis holds for A. Let $x \in \mathit{vars}(A)$. Instantiating the trichotomy-rule (Tab. 3(c)) by $[X \rightsquigarrow x]$ and applying it yields the following five proof obligations:

1. $A\,[x \rightsquigarrow \not\perp\!\!\!\not\perp] = C$
2. $A\,[x \rightsquigarrow \texttt{true}] = C$
3. $A\,[x \rightsquigarrow \texttt{false}] = C$
4. $\mathrm{cp}(\lambda z.A\,[x \rightsquigarrow z])$
5. $\mathrm{cp}(\lambda z.C)$

For the first three cases, the hypothesis applies. The last case follows from the center rule of Tab. 3(e). The fourth obligation can be discharged by at most n applications of the rules of Tab. 3(e), where n is the size of A.

□

Note that this proof only uses elementary rules and trichotomy (plus cp related rules). Its constructive flavor gives rise to a particular decision procedure we implemented in HOL-OCL .

4.3 A Local Equational Calculus for OCL (LEC)

Analogously to universal equality, a local validity calculus can be developed. Tab. 4(a) shows the general scheme of LEC congruence rules. For several operators, stronger logical rules can be derived, that accumulate semantic knowledge for sub-derivations from the context in which they are applied in; these rules are presented in Tab. 4(b). This information can be used by the third group of rules in Tab. 4(c), which allows for generalizing sub-terms in larger contexts (which must be context-passing) according to assumptions.

$$\frac{A\ st = A'\ st \quad B\ st = B'\ st}{(A\ \ op\ \ B)\ st = (A'\ \ op\ \ B')\ st}$$

(a) Congruence Rules (for $op \in \{\texttt{not}, \texttt{or}, \texttt{xor}\}$)

$$\frac{st \vDash_{\mathsf{t}} \texttt{def}(A) \qquad \begin{array}{c}[st \vDash_{\mathsf{t}} A]\\ \vdots \\ B\ st = B'\ st\end{array}}{(A\ \ op\ \ B)\ st = (A\ \ op\ \ B')\ st}$$

$$\frac{\begin{array}{c}[st \vDash_{\mathsf{t}} A]\\ \vdots \\ B\ st = B'\ st\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{f}} A]\\ \vdots \\ C\ st = C'\ st\end{array}}{(\texttt{if}\ A\ \texttt{then}\ B\ \texttt{else}\ C\ \texttt{endif})\ st = (\texttt{if}\ A\ \texttt{then}\ B'\ \texttt{else}\ C'\ \texttt{endif})\ st}$$

(b) Context Rules (for $op \in \{\texttt{and}, \texttt{implies}\}$)

$$\frac{st \vDash_{\mathsf{t}} A \qquad P\ \texttt{true}\ st = P'\ \texttt{true}\ st \qquad \mathrm{cp}(P) \qquad \mathrm{cp}(P')}{P\ A\ \ st = P'\ A\ \ st}$$

$$\frac{st \vDash_{\mathsf{f}} A \qquad P\ \texttt{false}\ st = P'\ \texttt{false}\ st \qquad \mathrm{cp}(P) \qquad \mathrm{cp}(P')}{P\ A\ st = P'\ A\ st}$$

$$\frac{st \vDash_{\mathsf{u}} A \qquad P\ \lightning\ st = P'\ \lightning\ st \qquad \mathrm{cp}(P) \qquad \mathrm{cp}(P')}{P\ A\ st = P'\ A\ st}$$

(c) Local Validity Propagation

Table 4 The Local Equational Calculus LEC

The calculus LEC is particularly suited for backward-proofs; when applied bottom-up, formulae were decomposed deterministically via the congruence and the context rules. During this process, semantic context knowledge is accumulated in the assumption list, which can be exploited via the propagation rules who replace sub-terms by `true`, `false`, or $\lightning$ which leads in in combination with GEC in practice to drastic simplifications of the current proof-goal.

A proof in LEC is a derivation that leads to $A\ st = \texttt{true}\ st$, which is notationally equivalent to $st \vDash_{\mathsf{t}} A$.

4.4 Reasoning over OCL-Equalities

When extending the propositional fragment of OCL by strong and weak equality (where strong equality implies weak equality), the usual rules of an equational theory with reflexivity, symmetry and transitivity except substitutivity holds. Our variant of substitutivity needs again that the term-context P is context passing:

$$\frac{st \models_{\mathsf{t}} a \; rel \; b \quad st \models_{\mathsf{t}} P \; a \quad \mathrm{cp}(P)}{st \models_{\mathsf{t}} P \; b}$$

where $rel \in \{\doteq, \triangleq\}$. The side-condition $\mathrm{cp}(P)$ captures the fact that our congruences hold on *HOL-OCL*-terms, but not on arbitrary HOL terms. Note that from $st \models_{\mathsf{t}} a \doteq b$ follows the definedness of a and b.

4.5 A Local Tableaux Calculus for OCL (LTC)

The tableaux methodology is one of the most popular approaches to design and implement proof-procedures. While originally geared towards first-order theorem proving, in particular for non-clausal formulae accommodating equality, renewed research activity is being devoted to investigating tableaux systems for intuitionistic, modal, temporal and many-valued logics, as well as for new families of logics, such as non-monotonic and substructural logics. Many of these recent approaches are based on a special labeling technique on the level of judgments, called labeled deduction [19,54]. Of course, labeling can also be embedded into a higher-order, classical meta-logic. Being a special case of a many-valued logic, tableaux calculi for SKL based on labeled deduction have been extensively studied [29,24,23]. In the following, we present a version for SKL roughly following [29] that can be processed by Isabelle's generic proof procedures, which are geared towards natural deduction. We show the completeness of this core calculus, and outline optimized versions that are more efficient.

Tableau proofs may be viewed as trees where the nodes are lists of formulae. Tableau rules extend the leaves of a tree by a new subtree, i.e. by adding one or more leaves below, where the latter case is called "branching" and is used for case splits. Classical tableau rules are purely analytic: each rule captures the full logical content of the expanded connective. Backtracking from a rule application is never necessary. The goal of the process is to construct trees in a deterministic manner, where the leaves can eventually all be detected as "closed", i.e. a logical contradiction is detected. This last step,

$$\neg(st \vDash_{\mathsf{t}} A) = (st \vDash_{\mathsf{f}} A) \vee (st \vDash_{\mathsf{u}} A) \qquad \neg(st \vDash_{\mathsf{f}} A) = (st \vDash_{\mathsf{t}} A) \vee (st \vDash_{\mathsf{u}} A)$$
$$\neg(st \vDash_{\mathsf{u}} A) = (st \vDash_{\mathsf{t}} A) \vee (st \vDash_{\mathsf{f}} A) = st \vDash_{\mathsf{t}} \mathtt{def}(A)$$

(a) Judgment Negation

$$\frac{st \vDash_{\mathsf{t}} \mathtt{def}(A) \qquad \begin{array}{c}[st \vDash_{\mathsf{t}} A]\\ \vdots \\ R\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{f}} A]\\ \vdots \\ R\end{array}}{R} \qquad \frac{\begin{array}{c}[\neg(st \vDash_{\mathsf{f}} A)]\\ \vdots \\ st \vDash_{\mathsf{t}} A\end{array}}{st \vDash_{\mathsf{t}} \mathtt{def}(A)} \qquad \frac{st \vDash_{\mathsf{u}} A}{st \vDash_{\mathsf{f}} \mathtt{def}(A)} \qquad \frac{st \vDash_{\mathsf{f}} \mathtt{def}(A)}{st \vDash_{\mathsf{u}} A}$$

(b) Definedness Introduction and Elimination

$$\frac{st \vDash_{\mathsf{f}} (\mathtt{not}\ A)}{st \vDash_{\mathsf{t}} A} \qquad \frac{st \vDash_{\mathsf{t}} A}{st \vDash_{\mathsf{f}} (\mathtt{not}\ A)} \qquad \frac{st \vDash_{\mathsf{u}} (\mathtt{not}\ A)}{st \vDash_{\mathsf{u}} A} \qquad \frac{st \vDash_{\mathsf{u}} A}{st \vDash_{\mathsf{u}} (\mathtt{not}\ A)} \qquad \frac{st \vDash_{\mathsf{t}} (\mathtt{not}\ A)}{st \vDash_{\mathsf{f}} A} \qquad \frac{st \vDash_{\mathsf{f}} A}{st \vDash_{\mathsf{t}} (\mathtt{not}\ A)}$$

(c) Negation

$$\frac{st \vDash_{\mathsf{t}} A\ \mathtt{and}\ B \qquad \begin{array}{c}[st \vDash_{\mathsf{t}} A, st \vDash_{\mathsf{t}} B]\\ \vdots \\ R\end{array}}{R} \qquad \frac{st \vDash_{\mathsf{f}} (A\ \mathtt{and}\ B) \qquad \begin{array}{c}[st \vDash_{\mathsf{f}} A]\\ \vdots \\ R\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{f}} B]\\ \vdots \\ R\end{array}}{R}$$

$$\frac{st \vDash_{\mathsf{t}} A \quad st \vDash_{\mathsf{t}} B \qquad \begin{array}{c}[\neg(st \vDash_{\mathsf{f}} B)]\\ \vdots \\ st \vDash_{\mathsf{f}} A\end{array}}{st \vDash_{\mathsf{t}} (A\ \mathtt{and}\ B)\ st \vDash_{\mathsf{f}} (A\ \mathtt{and}\ B)} \qquad \frac{\begin{array}{c}[st \vDash_{\mathsf{t}} \mathtt{def}(B))]\\ \vdots \\ st \vDash_{\mathsf{f}} \mathtt{def}(A)\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{t}} \mathtt{def}(A)]\\ \vdots \\ st \vDash_{\mathsf{t}} A\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{t}} \mathtt{def}(B)]\\ \vdots \\ st \vDash_{\mathsf{t}} B\end{array}}{st \vDash_{\mathsf{u}} (A\ \mathtt{and}\ B)}$$

$$\frac{st \vDash_{\mathsf{u}} (A\ \mathtt{and}\ B) \qquad \begin{array}{c}[st \vDash_{\mathsf{u}} A, st \vDash_{\mathsf{u}} B]\\ \vdots \\ R\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{u}} A, st \vDash_{\mathsf{t}} B]\\ \vdots \\ R\end{array} \qquad \begin{array}{c}[st \vDash_{\mathsf{t}} A, st \vDash_{\mathsf{u}} B]\\ \vdots \\ R\end{array}}{R}$$

(d) Conjunction Introduction and Elimination

$$\frac{st \vDash_{\mathsf{t}} A \quad st \vDash_{\mathsf{f}} A}{R} \qquad \frac{st \vDash_{\mathsf{t}} A \quad st \vDash_{\mathsf{u}} A}{R} \qquad \frac{st \vDash_{\mathsf{f}} A \quad st \vDash_{\mathsf{u}} A}{R} \qquad \frac{st \vDash_{\mathsf{u}} \mathtt{def}(A)}{R} \qquad \frac{st \vDash_{\mathsf{t}} A}{st \vDash_{\mathsf{t}} \mathtt{def}(A)} \qquad \frac{st \vDash_{\mathsf{f}} A}{st \vDash_{\mathsf{t}} \mathtt{def}(A)}$$

(e) Contradictions

Table 5 The Core of LTC

however, may be combined with the non-deterministic search for a substitution making this contradiction possible.

4.5.1 A Natural Deduction Tableaux Calculus for OCL. The particular format of a rule as a Horn-clause is the reason for the well-known symmetry of rule-sets (similar to sequent calculi): for each logical connective, two corresponding rules — called *introduction* and *elimination* rules — have to be introduced; the former act on conclusions of a goal, the latter on one of the assumptions.

As already mentioned, the Tableaux Method requires decomposition rules that capture the full logical content of the expanded connective. It is instructive to consider the example of the disjunction introduction rule $(\neg B \Longrightarrow A) \Longrightarrow A \vee B$ respectively the disjunction elimination rule $[\![A \vee B; A \Longrightarrow R; B \Longrightarrow R]\!] \Longrightarrow R$ for HOL, which are usually presented in the textbooks as:

$$\frac{\begin{array}{c}[\neg B]\\ \vdots\\ A\end{array}}{A \vee B} \qquad \frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots\\ R\end{array} \quad \begin{array}{c}[B]\\ \vdots\\ R\end{array}}{R} \tag{4}$$

Using the introduction rule, a proof state (which has again the format of a Horn-clause) $X \Longrightarrow Y \vee Z$ can be transformed into $[\![X; \neg Z]\!] \Longrightarrow Y$, while a goal $[\![X \vee Y; A]\!] \Longrightarrow Z$ can be transformed via the elimination rule into the goals $[\![X; A]\!] \Longrightarrow Z$ and $[\![Y; A]\!] \Longrightarrow Z$. Note that the former proof state transformation does not lose the information that the goal is satisfiable if Z holds (this leads to a contradiction in the assumption list). The latter transformation performs a case distinction.

Keeping these remarks in mind, the presentation of LEC in Tab. 5(a) is pretty much straight-forward: first, we present a group of rules that translates negative judgments, then follow the groups of tableaux rules for definedness, `not` and `and`, and we conclude with a group of rules for closing clauses.

Negative judgments can be replaced by HOL-disjunctions as a consequence of the following fundamental property for judgments, namely that judgment is either valid or invalid or undefined:

$$(st \vDash_{\mathsf{t}} A) \vee (st \vDash_{\mathsf{f}} A) \vee (st \vDash_{\mathsf{u}} A)$$

which justifies the identities presented in Tab. 5(a).

The next group of rules is concerned with definedness. Four (Tab. 5(b)) of the six cases are straight-forward, while the latter two constitute contradictions and are presented as closure rules later.

Now, we consider the case for `not` in Tab. 5(c). These rules are a consequence of $\texttt{not}(\texttt{not}(X)) = X$ and eliminate these situations at the root of a formula. These elimination rules have a particularly simple form and can therefore be used directly as so-called *destruction rules* that are used to weaken assumptions in a Horn-clause directly. Note that the last two rules of the group are notational equivalences resulting from our notational conventions for $\models_{\mathtt{t}}$; they are thus not explicitly inserted into the rule set.

Finally, we describe the rules that close such Horn-clauses. Built-in in HOL, there are already three rules that detect satisfiable Horn-clauses, namely the (HOL) not-elimination, classical contradiction not-introduction rules and assumption (from left to right):

$$\frac{\neg P \qquad P}{R} \qquad \frac{\begin{array}{c}[\neg P]\\ \vdots\\ \text{False}\end{array}}{P} \qquad \frac{\begin{array}{c}[P]\\ \vdots\\ \text{False}\end{array}}{\neg P} \qquad \frac{P}{P}$$

Besides these HOL-logical rules for closing a goal, there are also OCL-logical rules motivated by the satisfiability of a Horn-clause (Tab. 5(e)).

This gives rise to a very flexible format of a proof state in LTC; it is a Horn-clause of one of the two forms:

$$\frac{H_1 \ldots H_i, \neg(H_{i+1}) \ldots \neg(H_m)}{H_{m+1}} \qquad \frac{H_1 \ldots H_i, \neg(H_{i+1}) \ldots \neg(H_m)}{\neg(H_{m+1})}$$

where H_i has the form $st \models_{C_i} A_i$. Standard proof states in UEC can be converted automatically in proof states of this form via one of the bridge-theorems (2a)-(2c). The elimination and introduction rules shown above reduce or split proof steps of this form in logically equivalent steps to new ones. Eventually, the process results in a sequence of Horn-clauses with labeled literals.

4.5.2 An Example Derivation. In Tab. 6 a little example is presented, showing LTS at work. For a proof state, we write $A_1, \ldots, A_n \vdash A_{n+1}$ where the left-hand side of the $\vdash$-symbol denotes the assumption list and the right-hand side the goal to show. In this example, the proof starts with no assumptions and the commutativity of `and`, denoted as $\vdash A\,\texttt{and}\,B = B\,\texttt{and}\,A$. The proof proceeds in a deterministic, bottom-up manner (backward-proof), which finally leads to proof states that can be closed via the closing rules. At the beginning, the bridge-theorem (2b) is applied in order to start the proof-process. We omit the meta-quantifiers which are irrelevant in this sample proof. Note, that Isabelle proves such properties completely automatic.

$$\dfrac{\dfrac{\dfrac{}{st \vDash_{\mathsf{t}} B,\ st \vDash_{\mathsf{t}} A \vdash st \vDash_{\mathsf{t}} A}\quad \dfrac{}{st \vDash_{\mathsf{t}} B,\ st \vDash_{\mathsf{t}} A \vdash st \vDash_{\mathsf{t}} B}}{st \vDash_{\mathsf{t}} B \text{ and } A \vdash st \vDash_{\mathsf{t}} A \quad st \vDash_{\mathsf{t}} B \text{ and } A \vdash st \vDash_{\mathsf{t}} B}\quad \cdots \quad \textit{subtree} \quad \cdots}{\dfrac{st \vDash_{\mathsf{t}} A \text{ and } B \vdash st \vDash_{\mathsf{t}} B \text{ and } A}{\vdash st \vDash_{\mathsf{t}} A \text{ and } B \Leftrightarrow st \vDash_{\mathsf{t}} B \text{ and } A}\quad \dfrac{st \vDash_{\mathsf{t}} B \text{ and } A \vdash st \vDash_{\mathsf{t}} A \text{ and } B}{\vdash st \vDash_{\mathsf{u}} A \text{ and } B \Leftrightarrow st \vDash_{\mathsf{u}} B \text{ and } A}\quad \dfrac{st \vDash_{\mathsf{u}} B \text{ and } A \vdash st \vDash_{\mathsf{u}} A \text{ and } B}{\vdash st \vDash_{\mathsf{u}} A \text{ and } B \Leftrightarrow st \vDash_{\mathsf{u}} B \text{ and } A}}{\vdash A \text{ and } B = B \text{ and } A}$$

where *subtree* is:

$$\dfrac{\dfrac{\dfrac{}{\neg st \vDash_{\mathsf{u}} A,\ st \vDash_{\mathsf{u}} A,\ st \vDash_{\mathsf{u}} B \vdash st \vDash_{\mathsf{u}} B}\quad \dfrac{}{\neg st \vDash_{\mathsf{u}} A,\ st \vDash_{\mathsf{u}} A,\ st \vDash_{\mathsf{t}} B \vdash st \vDash_{\mathsf{u}} B}\quad \dfrac{}{\neg st \vDash_{\mathsf{u}} A,\ st \vDash_{\mathsf{t}} A,\ st \vDash_{\mathsf{u}} B \vdash st \vDash_{\mathsf{u}} B}}{st \vDash_{\mathsf{u}} A \text{ and } B,\ \neg st \vDash_{\mathsf{u}} A \vdash st \vDash_{\mathsf{u}} B}\quad \cdots \quad \dfrac{}{st \vDash_{\mathsf{u}} A \text{ and } B \vdash st \vDash_{\mathsf{u}} A}}{st \vDash_{\mathsf{u}} A \text{ and } B \vdash st \vDash_{\mathsf{f}} \mathtt{def}(B) \text{ and } \mathtt{def}(A)}\quad \dfrac{}{st \vDash_{\mathsf{u}} A \text{ and } B \vdash st \vDash_{\mathsf{f}} \mathtt{def}(A) \text{ and not } A}$$

$$st \vDash_{\mathsf{u}} A \text{ and } B \vdash st \vDash_{\mathsf{u}} B \text{ and } A$$

Table 6 A Example OCL Derivation

4.5.3 Completeness

Theorem 2. *LTC is complete for propositional HOL-OCL.*

Proof. We transform an LTC proof goal into a conjunction of goals (i.e. Horn-clauses) in an equivalent normal form: first, we eliminate all negative judgments by applying the identities (5(a)) and disjunction introduction and elimination (4). Throughout these goals, we expand `or`, `implies`, `xor` by their definitions. With respect to `if _ then _ else _ endif`, we apply a particular instance of the trichotomy-rule (Tab. 3(c)):

$$\frac{st \vDash_{\mathsf{t}} P \quad \not\perp \quad st \vDash_{\mathsf{t}} P\ Y \quad st \vDash_{\mathsf{t}} P\ Z \quad \mathrm{cp}(P)}{st \vDash_{\mathsf{t}} P(\mathtt{if}\ X\ \mathtt{then}\ Y\ \mathtt{else}\ Z\ \mathtt{endif})}$$

together with its variant as an elimination rule in order to eliminate all occurrences of `if _ then _ else _ endif`. Both normal forms can be computed in finitely many steps, are logically equivalent, and unique.

In the resulting sequence Horn-clauses with positive judgments, the core rules of Sec. 4.5.1 can now subsequently applied until after finitely many steps all judgments are reduced to variable or constant formulae. We formally prove that the resulting Horn-clause sequence is again equivalent by a number of lemmas that prove the "$\Leftarrow$"-direction of the equivalence of each logical rules of Sec. 4.5.1. Thus the original goal is only satisfiable if the resulting sequence of Horn-clauses is closeable, i.e. one of the close-rules can be applied. □

4.5.4 Handling Quantifiers. In the following, we discuss an extension of the propositional OCL fragment to a language with bounded quantifiers introduced for collections. For brevity, we will concentrate on the quantifiers on `Set`. Recall that in *HOL-OCL* we defined `Integer.allInstances'()` to be the infinite set of integers (instead of $\not\perp\!\!\!\perp_{\mathscr{O}}$ as prescribed by the standard); thus, quantifications over such sets reach the full power of arithmetics.

First, we present some universal equalities of the universal quantifiers, which also satisfy the usual context passing rules (Tab. 7(c)). With respect to strictness rules, the universal quantifier (and its dual the existential quantifier) follows the usual scheme:

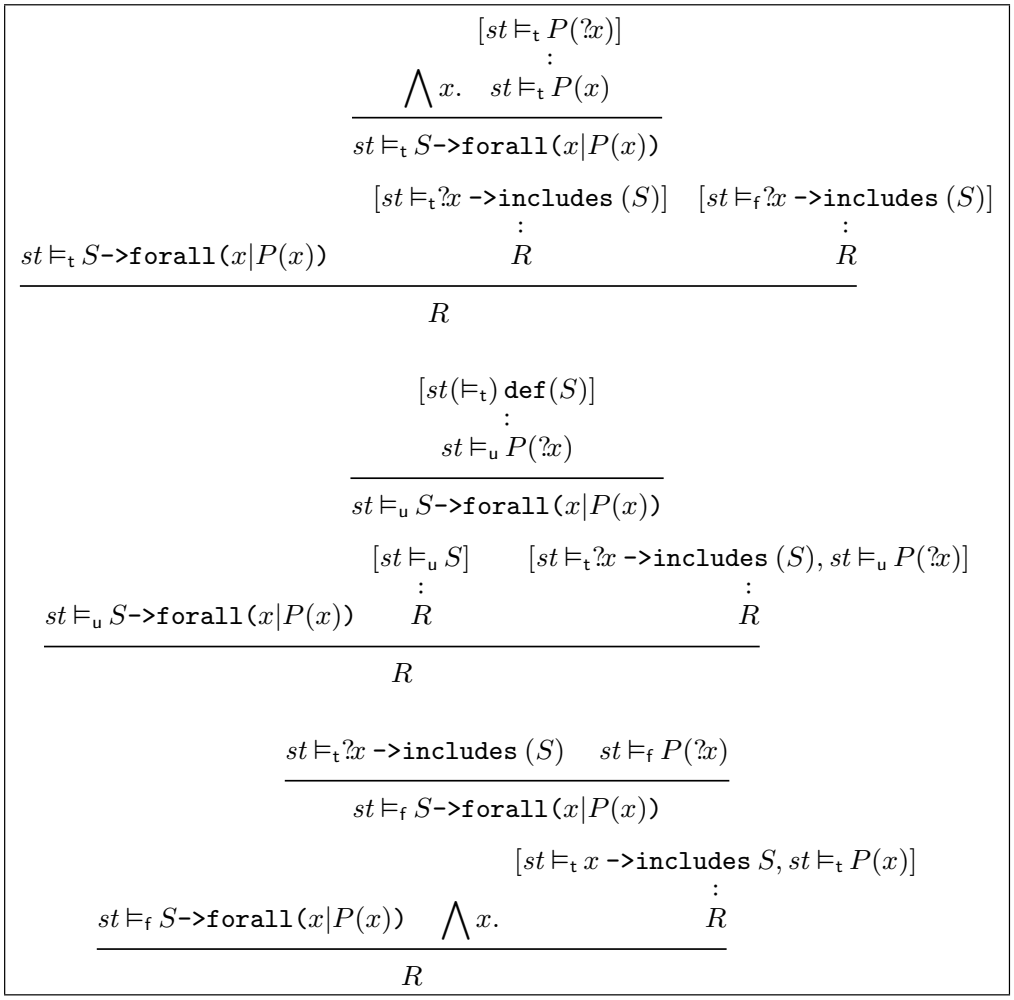
$$\begin{aligned} \not\perp\!\!\!\perp_{\mathscr{O}}\texttt{->forall}(x|Px) &= \not\perp\!\!\!\perp_{\mathscr{O}} \\ \not\perp\!\!\!\perp_{\mathscr{O}}\texttt{->exists}(x|Px) &= \not\perp\!\!\!\perp_{\mathscr{O}} \\ X\texttt{->forall}(\not\perp\!\!\!\perp_{\mathscr{O}}) &= \not\perp\!\!\!\perp_{\mathscr{O}} \end{aligned}$$

The distributivities of SKL can be extended to the quantifiers, see Tab. 7(b). Note that if `x->includes(S)` is valid, we know that `x` must be defined. This is a characteristic property of smashed sets that yields the following property:

$$st \vDash_{\mathsf{t}} x\ \texttt{->includes}\ (S) \Rightarrow st \vDash_{\mathsf{t}} \texttt{def}(x)$$

Following the usual "scheme of six rules" as for the logical connectives, we present the rules for the bounded universal quantifier. These rules are simply variations of standard quantifier rules in HOL, but subsume also definedness-reasoning. We start with the "usual" introduction and elimination rules for valid judgments. Note that we use meta-quantifier and meta-variables to represent Skolem terms and terms for witnesses; respectively (constructed during the proof at need via unification and resolution), see Tab. 7(a).

The following introduction and elimination rules capture the essence for undefined quantifiers: if the set S is defined (implying that each element in it is defined), then there must be an instance



(a) Skolem

$$\texttt{def}(X) = \texttt{true} \Rightarrow X\texttt{->forall(true)} = \texttt{true}$$
$$\texttt{def}(X) = \texttt{true} \Rightarrow X\texttt{->forall(false)} = \texttt{false}$$
$$X\texttt{->forall}(x|P(x)\ \texttt{and}\ Q(x)) = X\texttt{->forall}(x|P(x))\ \texttt{and}\ X\texttt{->forall}(x|Q(x))$$

(b) Distributivities

$$\frac{\mathrm{cp}(P) \quad \bigwedge x.\, \mathrm{cp}(P'x)}{\mathrm{cp}(\lambda X.(P\ X)\texttt{->forall}(\lambda x.(P'\ x\ X)))}$$

(c) Context Passing for Quantifiers

Table 7 Extensions of LTC: Quantifiers

of the quantifier body P that is undefined. On the other hand, from an undefined quantifier we have a case split for undefined S or for witnesses of undefined $P(?x)$: The rules for the existential quantifiers follow easily from the definition and rules above and are omitted here.

4.5.5 Completeness

Definition 4. *The* predicative fragment of OCL *is the inductively defined set of OCL-formulae that is built over variables (or applications of variables to bound variables), the propositional fragment, and the logical quantifiers (of OCL-Sets). Thus, the predicative fragment is an extension of the propositional fragment of OCL.*

The rules of Sec. 4.5.4 are incomplete for predicative OCL in the presented form, since any universal premise can be instantiated only once. However, for this class of theorems called *obvious theorems*, a concept attributed to Martin Davis and described by Rudnicki [47].

A complete version for predicative OCL can be obtained by adding contraction rules such as the standard HOL-rule:

$$\frac{\forall x.Px \qquad \begin{array}{c}[P?x, \forall x.Px] \\ \vdots \\ R\end{array}}{R}$$

Unfortunately, this type of rules can be applied infinitively many times. This reflects the fact that the propositional fragment (as an extension of first-order logic) is semi-decidable. A proof of completeness for a predicative LTC calculus with contraction rules may be adopted from [28].

With respect to full OCL including arithmetics and collection types, the question of completeness has to be answered negatively since the underlying theories are well-known to be too strong.

4.6 Lifting Theorems from HOL to the HOL-OCL Level

Since all operations in the OCL-library are defined canonically by a combination of smashing, strictification and lifting operators, i.e. since the definitions have a specific format, it is possible to derive automatically many rules from generic theorems for them. This applies to context passing rules (see Tab. 3(e)), strictness rules, definedness propagation etc. For example, some generic theorems have the form:

$$lift_2(strict(\lambda x.\, strict(f\ x)))\, \not\perp\!\!\!\perp\, X = \not\perp\!\!\!\perp$$
$$lift_2(strict(\lambda x.\, strict(f\ x)))\ X \not\perp\!\!\!\perp = \not\perp\!\!\!\perp$$
$$\mathtt{def}(lift_2(strict(\lambda x.\, strict(\lambda y. \lfloor f\ x\ y \rfloor)))\ X\ Y)) = \mathtt{def}(X)\ \mathtt{and}\ \mathtt{def}(Y)$$

From there, automatic derivations yield reduction rules like $x + \bot\!\!\!\bot = \bot\!\!\!\bot$ which are added to the rewriting rules of GEC and LEC, enabling them to reason over terms.

Moreover, we developed techniques to lift algebraic properties such as $X \cup Y = Y \cup X$ of library operators from the meta-level to OCL automatically. This covers a wide range of "folklore theorems" needed in a theorem proving environment. For details of this technique which is also required on combinator-style semantics, we refer to [12].

4.7 An Implementation in Isabelle

In this section, we present some details of our implementation using Isabelle. Isabelle's proof engine is centered around Horn-clauses: a *proof state* contains a list of Horn-clauses called *goals*. The proof engine proceeds by applying *tactics* which represent logical transformations.Some tactics like atac eliminate goals, because a conclusion follows from an assumption. A proof is finished if no goals remain.

In principle, UEC and LEC are both ideal for the rewriting engine of Isabelle encapsulated in the tactic simp_tac. In particular, the rules of LEC and GEC can be directly processed by the Isabelle rewriter. Datatype related reasoning is also amenable to this technique.

With respect to the OCL-equalities, the form of the substitutivity rule is unusual for the generic rewrite engine due to the cp(P) constraint that has to be checked at each rewrite redex. We implemented two proof procedures: a canonizer ocl_hyp_subst_tac and a (slow) rewriter ocl_simp_tac. The canonizer eliminates in a goal

$$[\![A_1(x); \ldots; st \vDash_{\mathsf{t}} x \ \mathit{rel}\ t; \ldots; A_n(x)]\!] \Longrightarrow A_{n+1}(x)$$

a variable x and replaces it with t

$$[\![A_1(t); \ldots; A_n(t)]\!] \Longrightarrow A_{n+1}(t)$$

where $\mathit{rel} \in \{\triangleq, \doteq\}$, or implicit HOL equalities as occurring in $st \vDash_{\mathsf{t}} x$, $st \vDash_{\mathsf{f}} x$ or $st \vDash_{\mathsf{u}} x$ (where $t \in \{\mathtt{true}, \mathtt{false}, \bot\!\!\!\bot\}$). The canonizer allows for the propagation of local knowledge throughout a whole proof state and paves the way for the ground reductions in UEC and therefore substantial reductions of its size. The OCL rewrite procedure can process general judgments of the form $st \vDash_{\mathsf{t}} t \ \mathit{rel}\ t'$ and applies them from left to right; an integration into the standard rewriter is highly desirable and may be possible in future versions of Isabelle.

Finally, we provided a setup of Isabelle's generic tableau prover blast_tac [44] with LTC. We applied a number of improvements:

1. Own LTC-rules were provided for all other OCL connectives such as `xor`, `implies` or `if _ then _ else _ endif`. Thus, some of the blowup of the proof size produced by the simplistic unfolding strategy (as used in proof 4.5.3) can be avoided.
2. ocl_hyp_subst_tac is performed interleaved with the tableau process whenever possible, followed by a simplification using strictness rules.
3. Instead of translating away negative judgments by applying the identities (5(a)), we chose to keep some of them following an idea of Hähnle [23].
4. We introduced the usual contraction rules for universal quantifier elimination in order to enable Isabelle to use its heuristic search procedures for non-obvious theorems.

Since both procedures are integrated into the standard tactic auto, they are used in the Isabelle/*HOL-OCL* environment as default.

5 Applications

In the previous sections, we described a formal, machine-checked semantics for OCL and derived sound and complete calculi , which are partly suited for automated reasoning over OCL specifications. Now we present our vision how to take practical advantage of these results.

In practice, OCL has been most widely used for the capture of requirements in object-oriented software analysis and for run-time testing of software components, e.g. [8]. Therefore, the most likely application scenarios for automated reasoning for the moment are:

1. analysis and simplification of specifications, and
2. optimization of run-time testing code.

When capturing the requirements for a larger software system, the problem arises how to detect potential inconsistencies, contradictions or redundancies in larger numbers of invariants or method specifications. For example, a specification may contain undefined subexpressions; many of them like subpath-expressions may be unavoidable, but it may be useful to inform the user of them. Contradictions may easily arise when post-conditions conflict with invariants; automated proof attempts of their conjunctions may reveal this. Redundancies may be detected by partitioning formulae (e.g. by splitting it along the topmost conjoints) and finding parts that are implied by others.

Since OCL's semantic has been geared towards executability, a number of tools have been developed (mostly based on [17]) to generate code from invariants and method specifications in order to check

a system component for violations at runtime. While this approach is not really an analytic method, it allows at least for an a posteriori analysis of crashes in large systems. Since the generated check-code is running at each entry or exit of a method, for example, there is a practical need for an optimization, be it by elimination of redundancies, be it by replacing general operators in particular contexts by more efficient ones (e.g. standard `and` by strict `and`) or by program transformation rules like let-unfold or let-introduction rules for common subexpressions, `if-then` simplifications and variants thereof.

In the previous sections, we also laid the groundwork for more advanced, future application scenarios, involving either challenges in computational complexity or interactive theorem proving. These are:

1. systematic test-case generation, and
2. transition from more abstract to more concrete models.

Systematic test-case generation is characterized by an algorithm that enumerates all test-cases for a specification or a program according to a preconceived *test adequacy criterion* [59]. Such a criterion could be *partition coverage*, *range-bound coverage* or *control path coverage*: a test is adequate meaning "a program is tested enough", if a test-case from each partition of the specification has been successfully passed by a program, or if for each control path in a program, there is a test-case. In itself, systematic test-case generation is similar to abstraction techniques leading to finite sub-models used for model-checking. We will present this scenario for partition analysis in the next section.

The transition from abstract to concrete system models, in particular concrete code, via a defined and well-understood relation is known as a *refinement* in the literature. So far, the OCL standard is very restrictive here: it offers only a notion of *satisfaction* [56, A.34] of a method implementation to an OCL method specification. Satisfaction in this sense requires an implementation (seen as a function F) to be just a subrelation of R (in the sense of Sec. 3.5). Thus, a program *must* diverge whenever the specification is false or undefined — and may not behave arbitrarily in this case, as in conventional operation refinement as in [50], where only the restriction of F to the domain of R must be included in R. We will discuss ways to integrate more liberal refinement notions into *HOL-OCL* in Sec. 5.2. Three-valued logics inherently offer a number of advantages for testing and refinement. More precisely, instead of interpreting the system specification as transition relation $set(state(\mathscr{U}_\alpha) \times state(\mathscr{U}_\alpha))$, we use the richer structure $V_\alpha(\text{Boolean}) = state(\mathscr{U}_\alpha) \times state(\mathscr{U}_\alpha) \to \text{Boolean}$ (recall that Boolean is three-valued; c.f. Sec. 3.3.3). As already mentioned, this paves the way for characterizing a state transition as *possible*,

impossible or *unknown*. Instead of identifying the impossible and unknown case in order to return to the well-traced road of two-valued logics (as suggested by the OCL standard), we argue that this richer structure has many advantages for the envisaged applications of OCL.

For example, in test harnesses, it is natural to implement recursive predicates occurring in pre-conditions and post-conditions by "bounded recursion up-to test depth m". Operationally, this can be understood as a counter of recursion calls that lead to $\not\perp_{\mathscr{L}}$ when exceeded; denotationally, this means that $f^m(\perp)$ is used to approximate $\mathrm{fix}(f)$ where f is the body functional of the predicate, cf. [57]. In a two-valued setting, a test must simply be aborted whenever the test depth was exceeded, in a three valued setting the remaining information can still be systematically exploited and leading to a valid test. Thus, it is possible to scale the ratio between test depth and efficiency in a three valued setting.

In refinement, for example, the distinction between impossible and unknown system transitions helps to avoid confusions [26]. In particular, if a pre-condition is false, the standard implies that a system should not be able to make a transition; in many cases, however, it is desirable to leave it up to the implementation how the system behaves in this case. Our approach gives more expressiveness: the operators $\texttt{toFalse}(X)$ and $\texttt{toUndef}(X)$ can be defined inside OCL; while the former maps `false` and $\not\perp_{\mathscr{L}}$ to `false`, the latter maps them to $\not\perp_{\mathscr{L}}$:

$$\texttt{toFalse}(X) = \texttt{if}\ \texttt{def}(X)\ \texttt{then}\ X\ \texttt{else}\ \texttt{false}\ \texttt{endif}$$
$$\texttt{toUndef}(X) = \texttt{if}\ X\ \texttt{then}\ X\ \texttt{else}\ \not\perp_{\mathscr{L}}\ \texttt{endif}$$

Thus, the specification may explicitly state if there is freedom for the implementor or not.

5.1 Automatic Test-Case Generation

We now show, by a small example, that *HOL-OCL* can be used for automatically partitioning the input domain. The adequacy criterion underlying this specification-based method following [16] is described as follows: for any method, and for each disjunction in the *disjunctive normal form* (DNF) of the method specification, there must be a test-case with generated input and results.

A prominent example for automatic test-case generation is the triangle problem [40]: given three integers representing the lengths of the sides of a triangle, a small algorithm has to check, whether these integers describe an equilateral, isosceles,scalene triangle, or no triangle at all. Assume a class Triangle with the operations isTri() (test

if the input describes a triangle) and triangle() (classify the valid triangle), further assume a enumeration type `TriType` that enumerates the possible result: `Equilateral`, `Isosceles`, `Scalene`, or `Invalid`:

```
context Triangle::isTri(s0, s1, s2:Integer):Boolean
pre:  (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result = (s2 < (s0 + s1)) and (s0 < (s1 + s2))
               and (s1 < (s0 + s2))

context Triangle::triangle(s0, s1, s2:Integer):TriType
pre:  (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result = if (isTri(s0,s1,s2)) then
                   if (s0 = s1) then
                       if (s1 = s2) then Equilateral
                       else Isosceles endif
                   else if (s1 = s2) then Isosceles
                        else if (s0 = s2) then Isosceles
                             else Scalene endif
                endif endif else Invalid endif
```

In *HOL-OCL* this leads to the following method specification[5]:

$$
\begin{aligned}
\text{triangle\_spec} \equiv \lambda res\ s_1\ s_2\ s_3.res \triangleq\ & \\
& (\texttt{if}\ \text{isTri}(s_1, s_2, s_3)\ \texttt{then} \\
& \quad \texttt{if}\ s_0 \triangleq s_1\ \texttt{then}\ \texttt{if}\ s_1 \triangleq s_2 \\
& \qquad \texttt{then}\ \text{equilateral}\ \texttt{else}\ \text{isosceles}\ \texttt{endif} \\
& \qquad \texttt{else if}\ s_1 \triangleq s_2\ \texttt{then}\ \text{isosceles}\ \texttt{else} \\
& \qquad\quad \texttt{if}\ s_0 \triangleq s_2\ \texttt{then}\ \text{isosceles}\ \texttt{else}\ \text{scalene} \\
& \qquad \texttt{endif endif endif} \\
& \quad \texttt{else}\ \text{invalid}\ \texttt{endif})
\end{aligned}
$$

For the actual test-case generation, we define triangle, which selects via Hilbert's epsilon operator (ε) an eligible implementation:

$$\text{triangle}\ s_0\ s_1\ s_2 \equiv \varepsilon res. \vDash_{\mathsf{t}} \text{triangle\_spec}\ res\ s_0\ s_1\ s_2$$

In our setting, Dick-Faivre's method leads to the following main steps:

1. Eliminate logical operators except `and`, `or`, and `not`.
2. Convert the formula into DNF.
3. Eliminate unsatisfiable disjunctions.
4. Select the actual set of test-cases.

[5] In the following, we omit the specification of isTri() for space reasons.

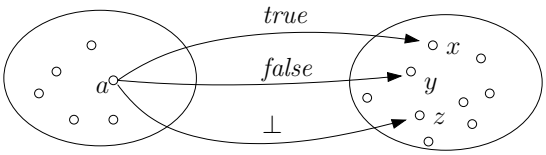


Figure 3 OCL specifications as labeled relations

HOL-OCL automatically calculates the DNF representation of the triangle-specification. The simplification process makes heavily use of *congruence rewriting*, which based on its deeper knowledge of the used datatypes (exploiting, e.g. $\neg \vDash \text{isosceles} \triangleq \text{invalid}$) eliminates many unsatisfiable disjunctions caused by conflicting constraints. At the end, only six cases are left, respectively one for invalid inputs and one for equilateral triangles, and three for isosceles triangles.

1. $(res \triangleq \text{invalid})$ **and not** $\text{isTri}(s_0, s_1, s_2)$
2. $(res \triangleq \text{equilateral})$ **and** $\text{isTri}(s_0, s_1, s_2)$ **and** $(s_0 \triangleq s_1)$ **and** $(s_1 \triangleq s_2)$
3. $(res \triangleq \text{isosceles})$ **and** $\text{isTri}(s_0, s_1, s_2)$ **and** $(s_0 \triangleq s_1)$ **and** $(s_1 \mathrel{\triangleq\!\!\!\!/} s_2)$
4. $(res \triangleq \text{isosceles})$ **and** $\text{isTri}(s_0, s_1, s_2)$ **and** $(s_0 \triangleq s_2)$ **and** $(s_0 \mathrel{\triangleq\!\!\!\!/} s_1)$
5. $(res \triangleq \text{isosceles})$ **and** $\text{isTri}(s_0, s_1, s_2)$ **and** $(s_1 \triangleq s_2)$ **and** $(s_0 \mathrel{\triangleq\!\!\!\!/} s_1)$
6. $(res \triangleq \text{scalene})$ **and** $\text{isTri}(s_0, s_1, s_2)$ **and** $(s_0 \mathrel{\triangleq\!\!\!\!/} s_1)$ **and** $(s_0 \mathrel{\triangleq\!\!\!\!/} s_2)$ **and** $(s_1 \mathrel{\triangleq\!\!\!\!/} s_2)$

An exhaustive test has to select at least one concrete test input from each of these six sub-domains and should also exploit boundary cases like minimal or maximum Integers of the underlying implementation.

5.2 Refining OCL Specifications

For a discussion of refinement notions in a three-valued setting (cf. Fig. 3), we need an extension of notions such as *domain* and *image*:

$$\begin{aligned} \text{dom}_X\, S &= \{\sigma | \exists \sigma'.(\sigma, \sigma') \models_X S\} \\ a(\!|S|\!)_X &= \{\sigma' | (a, \sigma') \models_X S\} \end{aligned}$$

Note that these definitions, as well as the following ones, are stated in HOL. Thus, we take advantage of having both HOL and OCL together in our environment *HOL-OCL*.

On this basis, we can define a first refinement notion, called *weak satisfaction*, which ignores the difference between undefined and impossible system transitions:

$$S \sqsubseteq_w T = \forall a \in \text{dom}_t(S).a(\!|T|\!)_t \subseteq a(\!|S|\!)_t$$

This means, for any input of S, and any possible result r, there must be a system transition; and any possible system transition in the

implementation must be also possible in the specification. A notion of *strong satisfaction* that takes explicitly impossible transitions into account:

$$\begin{aligned} S \sqsubseteq_s T = & \forall a \in \mathrm{dom}_t\, S. a(\!|T|\!)_t \subseteq a(\!|S|\!)_t \\ & \wedge \forall a \in \mathrm{dom}_f\, T - \mathrm{dom}_t\, S. a(\!|S|\!)_f \subseteq a(\!|T|\!)_f \end{aligned}$$

Both satisfaction notions presented above can be compared to operation refinements in the sense of [50, p. 135]. The following example will shed some light on the difference of these two notions:

```
context A::spec(a, b:Integer):Integer
post: result >= a div b

context A::prog(a, b:Integer):Integer
post: result = if b = 0 then 0 else a div b endif
```

Let us state that the method specification relation *spec* of method `spec` is refined by *prog* (interpreting `prog`). This can be done by *spec* `self` a b `result` $\sqsubseteq_X$ *prog* `self` a b `result` where $X \in \{s, f\}$. Four cases can be distinguished:

1. `b = 0`
2. `b <> 0`, `result < a div b`
3. `b <> 0`, `result > a div b`
4. `b <> 0`, `result = a div b`

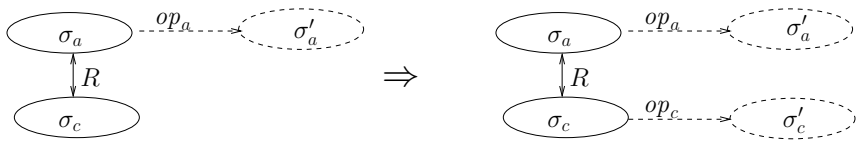
In case 1, *spec* `self` a `0` `result` $= \not\perp$ holds and any transition will be unknown, i.e. there is no element in dom_t(*spec* `self` a b `result`). In case 2, both *spec* and *prog* yield `false`, any transition is impossible and dom_t(*spec* `self` a b `result`) is again empty. In case 3, the *spec* yields `true` and *prog* `false`; thus, any transition is possible in *spec*, and $a(\!|$*prog* `self` a b `result`$|\!)_t = \{\}$ and $a(\!|$*spec* `self` a b `result`$|\!)_t = \{x \mid \mathrm{True}\}$. For the last case 4, both *spec* and *prog* are `true` and both sets of reachable states were equal. Summing up, the refinement holds for both weak and strong satisfaction. Note that the argument would be analogously if `a` and `b` would be attributes of `self` and not parameters of the methods.

We only briefly sketch how to prove the satisfaction relation. The key observation for the proof in HOL-OCL is the following lemma:

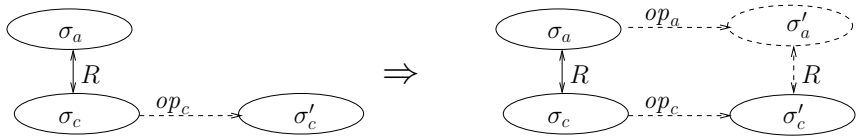
$$x(\!|S|\!)_X \subseteq y(\!|T|\!)_X = \forall z.(x, z) \models_X S \rightarrow (y, z) \models_X T$$

After unfolding a proof goal *spec* s a r $\sqsubseteq_w$ *prog* s a r according to the above definitions, a proof state remains that is amenable to LTS.

A principle limit of the satisfaction notions discussed above is a consequence of the fact that the underlying states of *spec* and *prog*



(a) Proof Obligation I



(b) Proof Obligation II

Figure 4 Data Refinement

are the same. Thus, it is impossible to relate methods stemming from different class diagrams with another, since the underlying data universes are incomparable. Thus, it is impossible to handle data structure changes naturally occurring the transition from specification to implementation. This limit can be overcome by data refinements similar to [50, p. 138]. The key concept of data refinement is the abstraction relation $R :: \text{set}(\mathit{state}(\mathscr{U}_\alpha) \times \mathit{state}(\mathscr{U}_\beta))$ that relates abstract states σ_a to concrete states σ_c underlying methods that may occur in different classes or even class diagrams possible. A (weak) data refinement $S \sqsubseteq_s^R T \equiv \mathit{po}_1(S, R, T) \wedge \mathit{po}_2(S, R, T)$ comes in two parts which turn into proof obligations when stated as proof goals. They are best explained with a diagram, such as see 4(a). The first condition po_1 means that whenever the abstract operation S *can* make a transition, the concrete operation T can make a transition too. More formally, this is covered by:

$$\mathit{po}_1(S, R, T) \equiv \forall \sigma_a \in \text{dom}_t(S).\forall \sigma_c.(\sigma_a, \sigma_c) \in R \rightarrow \sigma_c \in \text{dom}_t(T)$$

The second condition po_2 is presented in diagram 4(b). It states that whenever the concrete operation can make a step to a new system state σ'_c, then the abstract operation may be able to reach a state σ'_a that is in the abstraction relation to σ'_c. This condition is presented formally as follows:

$$\mathit{po}_2(S, R, T) \equiv \forall \sigma_a \in \text{dom}_t(S), \sigma_c.(\sigma_a, \sigma_c) \in R \wedge (\sigma_c, \sigma'_c) \vDash_{\mathsf{t}} T \rightarrow \exists \sigma'_a.(\sigma_a, \sigma'_a) \vDash_{\mathsf{t}} S \wedge (\sigma_a, \sigma'_a) \in R$$

A proof of data refinements proceeds by unfolding the above definitions and solving the resulting principal proof obligations in LTS.

Of course, other refinement concepts may be transferred to OCL along the lines presented above; e.g. *backward simulation* [58]. Further, it is possible to link method specifications to implementations in concrete code of a programming language like Java. Technically, this is an integration of a denotational semantics with a Hoare logic (cf. [57]), see also the corresponding formal proofs in Isabelle; meanwhile, it is feasible to combine *HOL-OCL* with a Hoare logic for Nano-Java [43]. Of course, such a combination will inherently be programming language specific. An in-depth discussion of these alternatives, however, is out of the scope of this paper.

6 Conclusion

6.1 Achievements

We presented a formal, machine-checked semantics of OCL in form of a conservative embedding into Isabelle/HOL that strives for compliance with the OCL standard. On the basis of this embedding, we derived several calculi and proof techniques for OCL. Since deriving means that we proved all rules with an interactive theorem proving assistant, we can guarantee both the consistency of the semantics as well as the soundness of the calculi. Our semantics is organized in a modular way, allowing extension such as temporal aspects or general recursion. The former may be used to give a formal semantics also to the other UML diagrams such as statecharts or sequence diagrams, while the latter may provide the basis to the development of powerful and executable libraries *within* OCL.

We developed automatic proof support for the derived calculi UEC, LEC, and LTC. In particular, the calculi led to rewriting and tableau based decision procedures for certain fragments of OCL. Based on a three-valued logic, many doubts have been expressed that OCL can be used for efficient deduction in OCL. While clearly a price has to be paid here, our experiments seem to indicate that in a predominantly strict language, there is a sufficiently high potential for optimizations such that effective reasoning in this language is actually feasible with recent technology. Thus, we provided the basis for deduction based OCL tools.

Finally, we demonstrated the potential for application of such deduction based tools for OCL. We adopted a classical data-refinement notion to three-valued OCL and showed how this can be used to relate specifications to implementations, even if based on different data

universes. In another application scenario, we used our automated deduction techniques for normal form computations of OCL specifications in order to yield automated test-case generation.

Thus, we believe that we have provided a solid basis for turning object oriented modeling in the UML into a true formal method.

6.2 Related Work

Our work is related to four areas: formal tool support for OCL, formal semantics for OCL, proof support for three-valued logics, and the embedding of object-oriented languages into theorem provers.

6.2.1 Formal OCL Tool Support. The group of tools providing OCL support beyond type checking can be roughly divided into three groups: runtime checking of OCL constraints (e.g. [17,8]), using OCL for model simulation and validation (e.g. [46]) and proof environments. The latter group contains only the integrated KeY tool [2]; which, as the many tools claiming to support OCL, is *not* compliant to the OCL specification, but based on a two-valued version of dynamic first-order logic combined with a fragment of Java. Moreover, it does not attempt to build up the theory of OCL by definitional axioms and thus has not formally investigated the issue of consistency. However, KeY provides an easy-to-use integration into a CASE tool, which is missing in *HOL-OCL* at the moment.

6.2.2 Proof Support for Three-valued Logics. The construction of specialized (semi-)decision procedures for many-valued logics such as SKL has been investigated before, in particular based on semantic tableaux methods [28,4]. Actually, the development of the tableaux-based proof-procedure in Isabelle has been deeply influenced by the leanTAP [4], which in itself is just the "bare bones" of its ancestor 3TAP based on SKL. However, one of our design goals is to provide a suitably abstract calculi that can be processed in a *generic* (Isabelle-like) prover engine.

6.2.3 Formal OCL Semantics. Previous semantic definitions of OCL are based on "mathematical notation" in the style of "naïve set theory", which is in our view quite inadequate to cover subtle subjects of object-orientation, e.g. inheritance. Moreover, the development of proof calculi and automated deduction for OCL has not been in the focus of interest so far. In [32], an operational semantics together with

a formal type system for OCL 1.4 [41] was presented. The only semantics for OCL 2.0 is given in a semi-formal way in the OCL standard [56] (based on [45]). In contrast to the mentioned work, our version of OCL can admit infinite states which turns `allInstances` into an infinite universal quantifier and, as we strongly suggest, adding least-fixpoint semantics for recursive methods.

6.2.4 Embedding of Object-oriented Languages Using a shallow embedding for an object-oriented language is still a challenge. While the basic concepts in our approach of representing subtyping by the subsumption relation on polymorphic types are not new [48,36], we included concepts such as undefinedness, mutual recursion between object instances, dynamic types, recursive method invocation and extensible class hierarchies.

6.3 Future Work

6.3.1 Improving Proof Support. While our existing proof procedures for OCL are quite satisfactory, more work has to be done in order to increase efficiency and to cover larger fragments of the language (including automated procedures for arithmetics, for example). Combining techniques for multivalued logics [22] and optimized data structures (e.g. decision diagrams) should yield efficient proof support for highly complex proofs usually occurring in test-case generations.

6.3.2 Tool Integration. Aiming for the broader acceptance of formal methods, and with a tool like *HOL-OCL* we understand OCL as such, a powerful integration into accepted CASE tools has to be provided. This includes the building of powerful, easy to use, front-ends which have to build on a improved, highly automated, proof support.

6.3.3 Applications. Beside larger case studies, e.g. in the area of secure and safe system development, we see a great potential for a formal refinements calculus for OCL. Such a refinement calculus would one allow to use *HOL-OCL* in a consistent way over several stages of a formally supported software development cycle and is in our opinion a cornerstone for applying formal methods successfully.

Also, combining *HOL-OCL* with other theorem prover projects such as μJava [39] opens a interesting field. Such combinations of formal semantics for specification and programming languages can pave the way for an integrated formal reasoning over specifications and code.

References

1. ArgoUML, July 2003. `http://argouml.tigris.org`.
2. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. Technical Report 2000/4, University of Karlsruhe, Department of Computer Science, 2000.
3. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, Orlando, 1986.
4. B. Beckert and J. Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.
5. E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: The notion and the issues. *Trans. on Knowledge and Data Engineering*, 4(3):223–237, 1992.
6. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP*, volume A-10, pages 129–156, Nijmegen, 1992. Elsevier.
7. M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, O. Slotosch, F. Regensburger, and K. Stølen. The requirement and design specification language Spectrum, an informal introduction (V 1.0), part 1 & 2. Technical Report TUM-I9312, Technische Universität München, 1993.
8. A. D. Brucker and B. Wolff. Testing distributed component bases systems using UML/OCL. In K. Bauknecht, W. Brauer, and T. Mück, editors, *Informatik 2001*, Tagungsband der GI/ÖCG Jahrestagung, pages 608–614. 2001.
9. A. D. Brucker and B. Wolff. HOL-OCL: Experiences, consequences and design choices. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*. Springer-Verlag, Dresden, 2002.
10. A. D. Brucker and B. Wolff. A note on design decisions of a formalization of the OCL. Technical Report 168, Albert-Ludwigs-Universität Freiburg, 2002.
11. A. D. Brucker and B. Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In C. Muñoz, S. Tahar, and V. Carreño, editors, *TPHOLs*, number 2410 in LNCS, pages 99–114. Springer-Verlag, 2002.
12. A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In H. Geuvers and F. Wiedijk, editors, *Types 2002*, number 2460 in LNCS. Springer-Verlag, Nijmegen, 2003.
13. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
14. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam Manifesto on OCL. Technical Report TUM-I9925, Technische Univerität München, 1999.
15. J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer-Verlag, London, 2001.
16. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *FME 92*, volume 670 of *LNCS*, pages 268–284. Springer, 1993.
17. OCL Compiler Suite, 2003. `http://dresden-ocl.sourceforge.net/`.
18. S. Flake and W. Mueller. An OCL extension for real-time constraints. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 150–171. Springer, 2002.

19. D. M. Gabbay. *Labelled Deductive Systems*. Number 33 in Oxford Logic Guides. Oxford University Press, 1997.
20. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
21. R. Hähnle. Towards an efficient tableau proof procedure for multiple-valued logics. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *Selected Papers from Computer Science Logic, CSL'90, Heidelberg*, volume 533 of *LNCS*, pages 248–260. Springer-Verlag, 1991.
22. R. Hähnle. *Automated Deduction in Multiple-valued Logics*. Oxford University Press, 1994.
23. R. Hähnle. Efficient deduction in many-valued logics. In *Proc. International Symposium on Multiple-Valued Logics, ISMVL'94, Boston/MA, USA*, pages 240–249. IEEE CS Press, Los Alamitos, 1994.
24. R. Hähnle. Tableaux for many-valued logics. In M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 529–580. Kluwer, Dordrecht, 1999.
25. S. Helke and T. Santen. Mechanized analysis of behavioral conformance in the Eiffel base libraries. *LNCS*, 2021:20–42, 2001.
26. R. Hennicker, H. Hussmann, and M. Bidoit. On the precise meaning of OCL constraints. In T. Clark and J.Warmer, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *LNCS*, pages 69–84. Springer-Verlag, 2002.
27. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1990.
28. M. Kerber and M. Kohlhase. A mechanization of strong Kleene logic for partial functions. In A. Bundy, editor, *Proceedings of CADE*, pages 371–385, Nancy, France, 1994. Springer Verlag. LNAI 814.
29. M. Kerber and M. Kohlhase. A tableau calculus for partial functions. *Collegium Logicum – Annals of the Kurt-Gödel-Society*, **2**:21–49, 1996.
30. C. Kobryn. Will UML 2.0 be agile or awkward? *CACM*, 45(1):107–110, 2002.
31. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
32. L. Mandel and M. V. Cengarle. A formal semantics for OCL 1.4. In C. K. M. Gogolla, editor, *UML 2001*, volume 2185 of *LNCS*. Springer, 2001.
33. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME 97*, volume 1313 of *LNCS*, pages 82–101. Springer-Verlag, 1997.
34. P. D. Mosses. *Denotational Semantics*, chapter 11. In van Leeuwen [53], 1990.
35. O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
36. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *TPHOLs*, volume 1479 of *LNCS*, pages 349–366. Springer, 1998.
37. T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
38. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
39. T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

40. N. D. North. Automatic test generation for the triangle problem. Technical Report DITC 161/90, National Physical Laboratory, Teddington, UK, 1990.
41. *Object Constraint Language Specification*, chapter 6. In Object Management Group [42], 2001.
42. OMG Unified Modeling Language Specification, 2001.
43. D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *FME 02*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.
44. L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.
45. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In T.-W. Ling, S. Ram, and M. L. Lee, editors, *17th Int. Conf. Conceptual Modeling*, pages 449–464. Springer, LNCS 1507, 1998.
46. M. Richters and M. Gogolla. OCL - syntax, semantics and tools. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.
47. P. Rudnicki. Obvious inferences. *Journal of Automated Reasoning*, 3(4):383–394, 1987.
48. T. Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification*. PhD thesis, Technical University Berlin, 1999.
49. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, 2001.
50. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 1992.
51. H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.
52. M. Utting and K. Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, pages 344–367. Springer-Verlag, 1993. LNCS 669.
53. J. van Leeuwen, editor. *Handbook of Theoretical Computer Science - Volume B: Formal Models and Semantics*. Elsevier, Amsterdam, 1990.
54. L. Viganò. *Labelled Non-Classical Logics*. Kluwer Academic Publishers, Dordrecht, 2000.
55. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Inc., Reading, MA, USA, 2003.
56. J. Warmer, A. Kleppe, T. Clark, A. Ivner, J. Högström, M. Gogolla, M. Richters, H. Hussmann, S. Zschaler, S. Johnston, D. S. Frankel, and C. Bock. Response to the UML 2.0 OCL RfP. Technical report, 2002.
57. G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993.
58. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996.
59. H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

Contents

1 Introduction 1
2 Preliminaries 4
2.1 A Guided Tour Through UML/OCL 4
2.2 The Syntax of OCL. 6
2.3 Formal and Technical Background of HOL in Isabelle . 6
2.3.1 The Meta-Language HOL 6
2.3.2 The Logical Framework Isabelle 8
3 A Formal Semantics of UML/OCL in Isabelle/HOL 9
3.1 Preliminaries: Lift Combinators 11
3.2 Preliminaries: Undefinedness and Strictness Combinators 12
3.3 Extensible Object Models and Path Expressions . . . 13
3.3.1 Managing Holes in Universes 13
3.3.2 Outlining the Coding Scheme: Types and Constructors 15
3.3.3 System States and System Transitions 15
3.3.4 Outlining the Coding Scheme: Accessors 16
3.3.5 Encoding the Running Example 16
3.3.6 Implementation Details 17
3.4 Semantics for OCL Expressions 18
3.4.1 System State Access Operations 18
3.4.2 Equalities 18
3.4.3 Logical Operators 19
3.4.4 Expressions: Standard Operations of the Library 20
3.5 Operation Specifications vs. Method Invocation 22
3.6 Possible Extensions 24
3.6.1 Alternative Logical Connectives 24
3.6.2 Method Invocation and General Recursion. . . 25
3.6.3 Alternative Quantifiers. 26
4 A Proof Calculus for OCL 27
4.1 Validity and Judgments 27
4.2 A Universal Equational Calculus for OCL 28
4.2.1 A Further Proof Principle of LEC: Trichotomy. 31
4.2.2 Completeness of UEC 31
4.3 A Local Equational Calculus for OCL (LEC) 32
4.4 Reasoning over OCL-Equalities 34
4.5 A Local Tableaux Calculus for OCL (LTC) 34
4.5.1 A Natural Deduction Tableaux Calculus for OCL. 36
4.5.2 An Example Derivation. 37
4.5.3 Completeness 38
4.5.4 Handling Quantifiers. 39

4.5.5 Completeness 41
4.6 Lifting Theorems from HOL to the *HOL-OCL* Level . . . 41
4.7 An Implementation in Isabelle 42
5 Applications 43
5.1 Automatic Test-Case Generation 45
5.2 Refining OCL Specifications 47
6 Conclusion 50
6.1 Achievements 50
6.2 Related Work 51
6.2.1 Formal OCL Tool Support. 51
6.2.2 Proof Support for Three-valued Logics. 51
6.2.3 Formal OCL Semantics. 51
6.2.4 Embedding of Object-oriented Languages . . . 52
6.3 Future Work 52
6.3.1 Improving Proof Support. 52
6.3.2 Tool Integration. 52
6.3.3 Applications. 52
7 Style Guide 58
8 Misc. Stuff 58
9 OCL Grammar 58

List of Tables

1 Formal Grammar of OCL (fragment) 7
2 Truth tables for the OCL operations `def`, `not`, `and`, and `or`. 20
3 The Propositional Universal Equational Calculus (UEC) 29
4 The Local Equational Calculus LEC 33
5 The Core of LTC 35
6 A Example OCL Derivation 38
7 Extensions of LTC: Quantifiers 40

List of Figures

1 Modeling a simple banking scenario with UML 5
2 Extending Class Hierarchies and Universes with Holes 14
3 OCL specifications as labeled relations 47
4 Data Refinement 49

List of FIXME's

7 Style Guide

8 Misc. Stuff

$$sumCase(f, g, p) = \begin{cases} f(x) & \text{if } p = Inr(x) \\ g(x) & \text{if } p = Inl(x) \end{cases}$$

9 OCL Grammar

$$\begin{aligned}
\mathit{classifierContextDecl} &::= \texttt{context}\ \mathit{pathName}\ \mathit{invDecl} \\
\mathit{invDecl} &::= [\mathit{invDecl}]\ \texttt{inv}\ [\mathit{simpleName}]\ :\ \mathit{expr} \\
\mathit{operationContextDecl} &::= \texttt{context}\ \mathit{operation}\ \mathit{prePostDecl} \\
\mathit{prePostDecl} &::= [\mathit{prePostDecl}]\ \texttt{pre}\ [\mathit{simpleName}]\ :\ \mathit{expr} \\
&\quad \mid [\mathit{prePostDecl}]\ \texttt{post}\ [\mathit{simpleName}]\ :\ \mathit{expr} \\
\mathit{operation} &::= [\mathit{pathName}\ ::]\ \mathit{simpleName}\ (\ [\mathit{varDeclList}]\ )\ [:\ \mathit{type}] \\
\mathit{varDeclList} &::= [\mathit{varDeclList}\ ,]\ \mathit{varDecl} \\
\mathit{varDecl} &::= \mathit{simpleName}\ [:\ \mathit{type}]\ [=\ \mathit{expr}] \\
\mathit{type} &::= \mathit{pathName} \mid \mathit{collKind}\ (\ \mathit{type}\ ) \\
\mathit{expr} &::= \mathit{literalExp} \mid \mathit{pathName}\ [@\texttt{pre}] \\
&\quad \mid \mathit{expr}.\mathit{simpleName}\ [@\texttt{pre}] \mid \mathit{expr}->\mathit{simpleName} \\
&\quad \mid \mathit{expr}([\{\mathit{expr},\}\mathit{expr}]) \mid \mathit{expr}(\mathit{varDecl}|\mathit{expr}) \\
&\quad \mid \mathit{expr}(\mathit{expr}[:\mathit{type}][=\mathit{expr}],\mathit{varDecl}|\mathit{expr}) \\
&\quad \mid \mathit{expr}[\{\mathit{expr},\}\mathit{expr}][@\texttt{pre}] \\
&\quad \mid \mathit{expr}->\texttt{iterate}(\mathit{varDecl}[;\mathit{varDecl}]|\mathit{expr}) \\
&\quad \mid \mathit{prefixOperator}\ \mathit{expr} \mid \mathit{expr}\ \mathit{infixOperator}\ \mathit{expr} \\
&\quad \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr}\ \texttt{endif} \\
&\quad \mid \texttt{let}\ \mathit{varDeclList}\ \texttt{in}\ \mathit{expr} \\
\mathit{infixOperator} &::= * \mid / \mid \texttt{div} \mid \texttt{mod} \mid + \mid - \mid < \mid > \\
&\quad \mid <= \mid >= \mid = \mid <> \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} \mid \texttt{implies} \\
\mathit{prefixOperator} &::= - \mid \texttt{not} \\
\mathit{literalExp} &::= \mathit{collLiteralExp} \mid \mathit{primitiveLiteralExp} \\
\mathit{collLiteralExp} &::= \mathit{collKind}\{\{\mathit{collLiteralPart},\}\mathit{collLiteralPart}\} \\
\mathit{collKind} &::= \texttt{Set} \mid \texttt{Bag} \mid \texttt{Sequence} \mid \texttt{Collection} \mid \texttt{OrderedSet} \\
\mathit{collLiteralPart} &::= \mathit{expr} \mid \mathit{expr}..\mathit{expr} \\
\mathit{primitiveLiteralExp} &::= \texttt{Integer} \mid \texttt{Real} \mid \texttt{String} \mid \texttt{true} \mid \texttt{false} \mid \texttt{OclUndefined} \\
\mathit{pathName} &::= [\mathit{pathName}::]\mathit{simpleName} \\
\mathit{simpleName} &::= SIMPLE_NAME
\end{aligned}$$

Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL

Nicole Rauch [1]

Universität Kaiserslautern, Germany

Burkhart Wolff

Albert-Ludwigs-Universität Freiburg, Germany

Abstract

We present a formal model of the Java two's-complement integral arithmetics. The model directly formalizes the arithmetic operations as given in the Java Language Specification (JLS). The algebraic properties of these definitions are derived. Underspecifications and ambiguities in the JLS are pointed out and clarified. The theory is formally analyzed in Isabelle/HOL, that is, machine-checked proofs for the ring properties and divisor/remainder theorems etc. are provided. This work is suited to build the framework for machine-supported reasoning over arithmetic formulae in the context of Java source-code verification.

Key words: Java, Java Card, formal semantics, formal methods, tools, theorem proving, integer arithmetic.

1 Introduction

Admittedly, modelling numbers in a theorem prover is not really a "sexy subject" at first sight. Numbers are fundamental, well-studied and well-understood, and everyone is used to them since school-mathematics. Basic theories for the naturals, the integers and real numbers are available in all major theorem proving systems (e.g. [11,26,21]), so why care?

However, numbers as specified in a concrete processor or in a concrete programming language semantics are oriented towards an efficient implementation on a machine. They are finite datatypes and typically based on bit-fiddling definitions. Nevertheless, they often possess a surprisingly rich theory (ring properties, for example) that also comprises a number of highly non-standard and tricky laws with non-intuitive and subtle preconditions.

[1] Partially funded by IST VerifiCard (IST-2000-26328)

In the context of program verification tools (such as the B tool [2], KIV [3], LOOP [5], and Jive [20], which directly motivated this work), efficient numerical programs, e.g. square roots, trigonometric functions, fast Fourier transformation or efficient cryptographic algorithms represent a particular challenge. Fortunately, theorem proving technology has matured to a degree that the analysis of realistic machine number specifications for widely-used programming languages such as Java or C now is a routine task [13].
With respect to the formalization of integers, we distinguish two approaches:

(1) the *partial approach*: the arithmetic operations $+ - * / \%$ are only defined on an interval of (mathematical) integers, and left undefined whenever the result of the operation is outside the interval (c.f. [4], which is mainly geared towards this approach).

(2) the *wrap-around approach*: integers are defined on $[-2^{n-1} .. 2^{n-1} - 1]$, where in case of overflow the results of the arithmetic operations are mapped back into this interval through modulo calculations. These numbers can be equivalently realized by bitstrings of length n in the widely-used two's-complement representation system [10].

While in the formal methods community there is a widespread reluctance to integrate machine number models and therefore a tendency towards either (infinite) mathematical integers or the first approach ("either remain fully formal but focus on a smaller or simpler language [...]; or remain with the real language, but give up trying to achieve full formality." [23]), we strongly argue in favour of the second approach for the following reasons:

(1) In a wrap-around implementation, certain properties like "Maxint + 1 = Minint" hold. This has the consequence that crucial algebraic properties such as the associativity law "a + (b + c) = (a + b) + c" hold in the wrap-around approach, but not in the partial approach. The wrap-around approach is therefore more suited for automated reasoning.

(2) Simply using the mathematical operators on a subset of the mathematical integers does not handle surprising definitions of operators appropriately. E.g. in Java the result of an integer division is always rounded towards zero, and thus the corresponding modulo operation can return negative values. This is unusual in mathematics. Therefore, this naïve approach does not only disregard overflows and underflows but also disregards unconventionally defined operators.

(3) The Java type `int` is defined in terms of wrap-around in the Java Language Specification [12], so why should a programmer who strictly complies to it in an efficient program be punished by the lack of formal analysis tools?

(4) Many parts of the JLS have been analyzed formally — so why not the part concerning number representations? There are also definitions and claimed properties that should be checked; and there are also possible inconsistencies or underspecifications as in all other informal specifications.

As technical framework for our analysis we use Isabelle/HOL and the Isar proof documentation package, whose output is directly used throughout this paper (for lack of space, however, we will not present any proofs here. The complete documentation will be found in a forthcoming technical report). Isabelle [21] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church's *higher-order logic* (HOL) [11], a classical logic with equality enriched by total polymorphic higher-order functions. In HOL, induction schemes can be expressed inside the logic, as well as (total) functional programs. Isabelle's methodology for safely building up large specifications is the decomposition of problems into *conservative extensions*. A conservative extension introduces new constants (by *constant definitions*) and types (by *type definitions*) only via axioms of a particular, machine-checked form; a proof that conservative extensions preserve consistency can be found in [11]. Among others, the HOL library provides conservative theories for the logical type *bool*, for the numbers such as *int* and for bitstrings *bin*.

1.1 Related Work

The formalization of IEEE floating point arithmetics has attracted the interest of researchers for some time [8,1], e.g. leading to concrete, industry strength verification technologies used in Intel's IA64 architecture [13].

In hardware verification, it is a routine task to verify two's complement number operations and their implementations on the gate level. Usually, variants of *binary decision diagrams* are used to represent functions over bit words canonically; thus, if a trusted function representation is identical to one generated from a highly complex implementation, the latter is verified. Meanwhile, addition, multiplication and restricted forms of operations involving division and remainder have been developed [15]. Unfortunately, it is known that one variant particularly suited for one operation is inherently intractable for another, etc. Moreover, general division and remainder functions have been proven to be intractable by word-level decision diagrams (WLDD) [24]. For all these reasons, the approach is unsuited to investigate the *theory* of two's complement numbers: for example, the theorem JavaInt-div-mod (see Section 4.2), which involves a mixture of all four operations, can only be proven up to a length of 9 bits, even with leading edge technology WLDD packages [2] .

Amazingly, *formalized theories* of the two's complement number have only been considered recently; i.e. Fox formalized 32-bit words and the ARM processor for HOL [9], and Bondyfalat developed a (quite rudimentary) bit words theory with division in the AOC project [6]. In the context of Java and the JLS, Jacobs [16] presented a fragment of the theory of integral types. This work (like ours) applies to Java Card as well since the models of the four smaller integral types (excluding long) of Java and Java Card are identical

[2] Thanks to Marc Herbstritt [14] to check this for us!

[25, § 2.2.3.1]. However, although our work is in spirit and scope very similar to [16], there are also significant differences:

- We use standard integer intervals as reference model for the arithmetic operations as well as two's-complement bitstrings for the bitshift and the bitwise AND, OR, XOR operations (which have not been covered by [16] anyway). Where required, we prove lemmas that show the isomorphy between these two representations.
- While [16] just presents the normal behavior of arithmetic expressions, we also cover the exceptional behavior for expressions like "x / 0" by adding a second theory layer with so-called strictness principles (see Sect. 6).

The Java Virtual Machine (JVM) [19] has been extensively modelled in the Project Bali [22]. However, the arithmetic operations in this JVM model are based on mathematical integers. Since our work is based on the same system, namely Isabelle2002, our model of a two's-complement integer datatype could replace the mathematical integers in this JVM theory.

1.2 Outline of this Paper

Section 2 introduces the core conservative definitions and the addition and multiplication, Section 3 presents the division and remainder theory and Section 4 gives the bitwise operations. Sections 2, 3 and 4 examine the *normal behavior*, while Section 5 describes the introduction of *exceptional behavior* into our arithmetic theory leading to operations which can deal with exceptions that may occur during calculations.

2 Formalizing the Normal Behavior Java Integers

The formalization of Java integers models the primitive Java type int as closely as possible. The programming language Java comes with a quite extensive language specification [12] which tries to be accurate and detailed. Nonetheless, there are several white spots in the Java integer specification which are pointed out in this paper. The language Java itself is platform-independent. The bit length of the data type int is fixed in a machine-independent way. This simplifies the modelling task. The JLS states about the integral types:

Java Language Specification [12], §4.2

> "The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing Unicode characters. [...] The values of the integral types are integers in the following ranges: [...] For int, from −2147483648 to 2147483647, inclusive"

The Java int type and its range are formalized in Isabelle/HOL [21] this way:

constdefs
BitLength :: nat BitLength ≡ 32

MinInt-int :: int	MinInt-int ≡ − (2 ^ (BitLength − 1))
MaxInt-int :: int	MaxInt-int ≡ 2^(BitLength − 1) − 1

Now we can introduce a new type for the desired integer range:
typedef JavaInt = {i. MinInt-int ≤ i ∧ i ≤ MaxInt-int}

This construct is the Isabelle/HOL shortcut for a type definition which defines the new type JavaInt isomorphic to the set of integers between MinInt-int and MaxInt-int. The isomorphism is established through the automatically provided (total) functions Abs-JavaInt :: int⇒JavaInt and Rep-JavaInt :: JavaInt⇒int and the two axioms y : {i. MinInt-int ≤ i ∧ i ≤ MaxInt-int} ⟹ Rep-JavaInt (Abs-JavaInt y) = y and Abs-JavaInt (Rep-JavaInt x) = x. Abs-JavaInt yields an arbitrary value if the argument is outside of the defining interval of JavaInt.
We define MinInt and MaxInt to be elements of the new type JavaInt:

constdefs

MinInt :: JavaInt	MinInt ≡ Abs-JavaInt MinInt-int
MaxInt :: JavaInt	MaxInt ≡ Abs-JavaInt MaxInt-int

In Java, calculations are only performed on values of the types int and long. Values of the three smaller integral types are widened first:

Java Language Specification [12], §4.2.2

> "If an integer operator other than a shift operator has at least one operand of type long, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type long. If the other operand is not long, it is first widened (§5.1.4) to type long by numeric promotion (§5.6). Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type int. If either operand is not an int, it is first widened to type int by numeric promotion. The built-in integer operators do not indicate overflow or underflow in any way."

This paper describes the formalization of the Java type int, therefore conversions between the different numerical types are not in the focus of this work. The integer types byte and short can easily be added as all calculations are performed on the type int anyways, so the only operations that need to be implemented are the widening to int, and the cast operations from int to byte and short, respectively. The Java type long can be added equally easily as our theory uses the bit length as a parameter, so one only need to change the definition of the bit length (see above) to gain a full theory for the Java type long, and again only the widening operations need to be added. Therefore, we only conentrate on the Java type int in the following.

Our model of Java int covers all side-effect-free operators. This excludes the operators ++ and −−, both in pre- and postfix notation. These operators return the value of the variable they are applied to while modifying the value stored in that variable independently from returning the value. We do not treat assignment of any kind either as it represents a side-effect as well. This also disallows combined operators like `a += b` etc. which are a shortcut for `a = a + b`. This is in line with usual specification languages, e.g. JML [17],

which also allows only side-effect-free operators in specifications. From a logical point of view, this makes sense as the specification is usually regarded as a set of predicates. In usual logics, predicates are side-effect-free. Thus, expressions with side-effects must be treated differently, either by special Hoare rules or by program transformation.

In our model, all operators are defined in Isabelle/HOL, and their properties as described in the JLS are proven, which ensures the validity of the definitions in our model. In the following, we quote the definitions from the JLS and present the Isabelle/HOL definitions and lemmas.

Our standard approach of defining the arithmetic operators on JavaInt is to convert the operands from JavaInt to Isabelle int, to apply the corresponding Isabelle int operation, and to convert the result back to JavaInt. The first conversion is performed by the representation function Rep-JavaInt (see above). The inverse conversion is performed by the function Int-to-JavaInt:

Int-to-JavaInt :: int ⇒ JavaInt

Int-to-JavaInt (x::int) ≡ Abs-JavaInt(
((x + (−MinInt-int)) mod (2 ∗ (−MinInt-int))) + MinInt-int)

This function first adds (−MinInt) to the argument and then performs a modulo calculation by $2 * (-\text{MinInt})$ which maps the value into the interval $[0 \,..\, 2*(-\text{MinInt})-1]$ (which is equivalent to only regarding the lowest 32 bits), and finally subtracts the value that was initially added. This definition is identical to the function Abs-JavaInt on arguments which are already in JavaInt. Larger or smaller values are mapped to JavaInt values, extending the domain to int.

This standard approach is not followed for operators that are explicitly defined on the bit representation of the arguments. Our approach differs from the approach used by Jacobs [16] who exclusively uses bit representations for the integer representation as well as the operator definitions.

2.1 Unary Operators

This section gives the formalizations of the unary operators +, − and the bitwise complement operator ∼. The unary plus operator on int is equivalent to the identity function. This is not very challenging, thus we do not elaborate on this operator. In the JLS, the unary minus operator is defined in relation to the binary minus operator described below.

Java Language Specification [12], §15.15.4

"At run time, the value of the unary minus expression is the arithmetic negation of the promoted value of the operand. For integer values, negation is the same as subtraction from zero.(1)
[...] negation of the maximum negative int or long results in that same maximum negative number.(2)
[...] For all integer values x, −x equals (∼x)+1.(3)"

The unary minus operator is formalized as

uminus-def : − (x::JavaInt) ≡ Int-to-JavaInt (− Rep-JavaInt x)

We prove the three properties described in the JLS:

(1) **lemma** uminus-property: $0 - x = -$ (x::JavaInt)

(2) **lemma** uminus-MinInt: $-$ MinInt = MinInt

(3) **lemma** uminus-bitcomplement: $(\sim x) + 1 = -x$

The bitwise complement operator is defined by unary and binary minus:

Java Language Specification [12], §15.15.5

"At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, $\sim$x equals $(-x)-1$."

This is formalized in Isabelle/HOL as follows:

constdefs

JavaInt-bitcomplement :: JavaInt ⇒ JavaInt

JavaInt-bitcomplement (x::JavaInt) ≡ $(-x) - (1$::JavaInt)

We use the notations $\sim$ and JavaInt-bitcomplement interchangeably.

3 Additive and Multiplicative Operators

3.1 Additive Operators

This section formalizes the binary + operator and the binary $-$ operator.

Java Language Specification [12], §15.18.2

"The binary + operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The binary - operator performs subtraction, producing the difference of two numeric operands.(1) [...]
Addition is a commutative operation if the operand expressions have no side effects. Integer addition is associative when the operands are all of the same type(2) [...]
If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format.(3)
If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.(4)
For both integer and floating-point subtraction, it is always the case that a$-$b produces the same result as a+$(-$b$)$.(5)"

(1) These two operators are defined in the standard way described above. We only give the definition of the binary + operator:
defs (**overloaded**)
add-def : x + y ≡ Int-to-JavaInt (Rep-JavaInt x + Rep-JavaInt y)

(2) This behavior is captured by the two lemmas
lemma JavaInt-add-commute: x + y = y + (x::JavaInt)
lemma JavaInt-add-assoc: x + y + z = x+(y+z::JavaInt)

(3) This requirement is already fulfilled by the definition.

(4) This specification can be expressed as
lemma JavaInt-add-overflow-sign :
(c = a + b ∧ MaxInt-int < Rep-JavaInt a + Rep-JavaInt b) ⟶ (c < 0)

This is a good example of how inexact several parts of the Java Language Specification are. If indeed only overflow, i.e. regarding two operands whose sum is larger than MaxInt, is meant here, then why pose such a complicated question? "the sign of the mathematical sum of the two operand values" will always be positive in this case, so why talk about "the sign of the result is not the same"? It would be much clearer to state "the sign of the result is always negative". But what if the authors also wanted to describe underflow, i.e. negative overflow, which is sometimes also referred to as "overflow"? In §4.2.2 the JLS states "The built-in integer operators do not indicate overflow or underflow in any way." Thus, the term "underflow" is known to the authors and is used in the JLS. Why do they not use it in the context quoted above? This would also explain the complicated phrasing of the above formulation.
To clarify these matters, we add the lemma
lemma JavaInt-add-underflow-sign :
$(c = a + b \wedge \text{Rep-JavaInt } a + \text{Rep-JavaInt } b < \text{MinInt-int}) \longrightarrow (0 \leq c)$

(5) This has been formalized as
lemma diff-uminus: $a - b = a + (-b::\mathsf{JavaInt})$

3.2 Multiplication Operator

The multiplication operator is described and formalized as follows:

Java Language Specification [12], §15.17.1

> "The binary * operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. [...] integer multiplication is associative when the operands are all of the same type"

defs (**overloaded**)
times-def : $x * y \equiv$ Int-to-JavaInt (Rep-JavaInt x $*$ Rep-JavaInt y)
The commutativity and associativity are proven by the lemmas
lemma JavaInt-times-commute: $(x::\mathsf{JavaInt}) * y = y * x$
lemma JavaInt-times-assoc: $(x::\mathsf{JavaInt}) * y * z = x * (y * z)$

Java Language Specification [12], §15.17.1

> "If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values."

This is again implicitly fulfilled by our standard modelling.

4 Division and Remainder Operators

4.1 Division Operator

In Java, the division operator produces the first surprise if compared to the mathematical definition of division, which is also used in Isabelle/HOL:

Java Language Specification [12], §15.17.2

"The binary / operator performs division, producing the quotient of its operands. [...] Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d \times q| \leq |n|$; moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.(1)
There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is -1, then integer overflow occurs and the result is equal to the dividend.(2)
Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an ArithmeticException is thrown.(3)"

This definition points out a major difference between the definition of division in Isabelle/HOL and Java. If the signs of dividend and divisor are different, the results differ by one because Java rounds towards 0 whereas Isabelle/HOL floors the result. Thus, the naïve approach of modelling Java integers by partialization of the corresponding operations of a theorem prover gives the wrong results in these cases.
We model the division by performing case distinctions:

defs (**overloaded**)
div-def : (x::JavaInt) div y ≡
if ((0 ≤ x ∧ y < 0) ∨ (x < 0 ∧ 0 < y)) then
Int-to-JavaInt (Rep-JavaInt x div Rep-JavaInt y) + 1
else
Int-to-JavaInt(Rep-JavaInt x div Rep-JavaInt y)

The properties mentioned in the language report are formalized as follows:

(1) **lemma** quotient-sign-plus :
abs d ≤ abs n ∧ neg (Rep-JavaInt n) = neg (Rep-JavaInt d)
⟹ 0 < (n div d)
lemma quotient-sign-minus :
abs d ≤ abs n ∧ neg (Rep-JavaInt n) ≠ neg (Rep-JavaInt d)
⟹ (n div d) < 0
The predicate "neg" holds iff the value of its argument is less than zero.

(2) **lemma** JavaInt-div-minusone : MinInt div −1 = MinInt

(3) is not modelled by the theory presented in this section because this theory does not introduce a bottom element for integers in order to treat exceptional cases. Our model returns 0 in this case. Exceptions are handled by the next theory layer (see Sect. 6) which adds a bottom element to JavaInt and lifts all operations in order to treat exceptions appropriately.

Again, the division operation is underspecified. The language report does not describe in (2) the sign of the resulting value if the magnitude of the dividend is less than the magnitude of the divisor.

4.2 Remainder Operator

The remainder operator is closely related to the division operator. Thus, it does not conform to standard mathematical definitions either.

Java Language Specification [12], §15.17.3

"The binary % operator is said to yield the remainder of its operands from an implied division [...] The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that (a/b)*b+(a%b) is equal to a.(1)
This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0).(2)
It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative,(3)
and can be positive only if the dividend is positive;(4)
moreover, the magnitude of the result is always less than the magnitude of the divisor.(5)
If the value of the divisor for an integer remainder operator is 0, then an ArithmeticException is thrown.(6)

Examples:		
	5%3 produces 2	(note that 5/3 produces 1)
	5%(-3) produces 2	(note that 5/(-3) produces -1)
	(-5)%3 produces -2	(note that (-5)/3 produces -1)
	(-5)%(-3) produces -2	(note that (-5)/(-3) produces 1)(7)"

When formalizing the remainder operator, we have to keep in mind the formalization of the division operator and the required equality (1). Therefore, the remainder operator mod is formalized as follows:

mod-def : (x::JavaInt) mod y ≡
  if (0 ≤ x ∧ y < 0) ∨ (x < 0 ∧ 0 < y) then
    Int-to-JavaInt(Rep-JavaInt x mod Rep-JavaInt y) − y
  else
    Int-to-JavaInt(Rep-JavaInt x mod Rep-JavaInt y)

The formulations in the JLS give rise to the following lemmas:

(1) **lemma** JavaInt-div-mod : ((a::JavaInt) div b) ∗ b + (a mod b) = a

(2) **lemma** MinInt-mod-minusone: MinInt mod −1 = 0
lemma MinInt-minusone-div-mod-eq :
(MinInt div −1) ∗ (−1) + (MinInt mod −1) = MinInt

(3) **lemma** neg-mod-sign-ineq : ((a::JavaInt) < 0) ⟹ ((a mod b) < 0)

(4) **lemma** pos-mod-sign-ineq : (0 < (a::JavaInt)) ⟹ (0 < (a mod b))

(5) **lemma** JavaInt-mod-less : abs ((a::JavaInt) mod b) < abs b

(6) See the discussion for div above.

(7) **lemma** div-mod-example1 : (5::JavaInt) mod 3 = 2 etc.

Again, it is not clear in the JLS what happens if the dividend equals 0.

Java is not the only language whose definitions of div and mod do not resemble the mathematical definitions. The languages Fortran, Pascal and Ada

define division in the same way as Java, and Fortran's MOD and Ada's REM operators are modelled in the same way as Java's % operator. Goldberg [10, p. H-12] regrets this disagreement among programming languages and suggests the mathematical definition, some of whose advantages he points out.

5 Formalization With Bitstring Representation

5.1 Shift Operators

The shift operators are not properly described in the JLS (§15.19) either. It is especially unclear what happens if the right-hand-side operand of the shift operators is negative. Due to the space limitations of this paper, we have to refrain from presenting the full formalization of the shift operators here.

5.2 Relational Operators

As the relational operators (described in JLS §§15.20, 15.21) do not offer many surprises, we abstain from presenting their formalization here.

5.3 Integer Bitwise Operators &, ^, and |

This section formalizes the bitwise AND, OR, and exclusive OR operators.

Java Language Specification [12], §15.22, 15.22.1

> "The bitwise operators [...] include the AND operator &, exclusive OR operator ^, and inclusive OR operator |.(1) [...]
> Each operator is commutative if the operand expressions have no side effects. Each operator is associative.(2) [...]
> For &, the result value is the bitwise AND of the operand values. For ^, the result value is the bitwise exclusive OR of the operand values. For |, the result value is the bitwise inclusive OR of the operand values. For example, the result of the expression 0xff00 & 0xf0f0 is 0xf000. The result of 0xff00 ^ 0xf0f0 is 0x0ff0. The result of 0xff00 | 0xf0f0 is 0xfff0.(3)"

(1) These bitwise operators are formalized as follows:

constdefs

JavaInt-bitand :: [JavaInt,JavaInt] ⇒ JavaInt
x & y ≡ number-of (zip-bin (op &::[bool,bool]⇒bool)
(bin-of x) (bin-of y))

where bin-of transforms a JavaInt into its bitstring representation, zip-bin merges two bitstrings into one by applying a function (which is passed as the first argument) to each bit pair in turn, and number-of turns the resulting bitstring back into a JavaInt. The other two bit operators are defined accordingly.

(2) The commutativity and associativity of the three operators is proven by six lemmas, of which we present two here:

lemma bitand-commute: a & b = b & a
lemma bitand-assoc: (a & b) & c = a & (b & c)

(3) We verify the results of the examples by proving the three lemmas
lemma bitand-example : 65280 & 61680 = 61440
lemma bitxor-example : 65280 ^ 61680 = 4080
lemma bitor-example : 65280 | 61680 = 65520
In these lemmas we transformed the hexadecimal values into decimal values because Isabelle is currently not able to read hex values.

5.4 Further Features of the Model

The model of Java integers presented above forms a ring. This could easily be proved by using Isabelle/HOL's Ring theory which only requires standard algebraic properties like associativity, commutativity and distributivity to be proven. The Ring theory makes dozens of ring theorems available for use in proofs. Our model also forms a linear ordering. To achieve this property, reflexivity, transitivity, antisymmetry and the fact that the $\leq$ operator imposes a total ordering had to be proven. This allows us to make use of Isabelle/HOL's linorder theory. We get a two's-complement representation by redefining (using our standard wrapper) the conversion function number-of-def which is already provided for int. This representation is used for those operators that are defined bitwise.

Altogether, the existing Isabelle theories make it relatively easy to achieve standard number-theoretic properties for types that are defined as a subset of the Isabelle/HOL integers.

5.5 Empirical Data: The Size of our Specification and Proofs

The formalization presented in the preceding sections consists of five theory files, the size of which is as follows:

Filename	Lines	Filename	Lines
JavaIntegersDef.thy	180	JavaIntegersAdd.thy	225
JavaIntegersTimes.thy	190	JavaIntegersDiv.thy	1210
JavaIntegersBit.thy	350		

It took about one week to specify the definitions and lemmas presented here and about six to eight weeks to prove them, but the proof work was mainly performed by one of the authors (NR) who at the same time learned to use Isabelle, so an expert would be able to achieve these results much faster.

6 Formalizing the Exceptional Behavior Java Integers

The Java Language Specification introduces the concept of *exception* in expressions and statements of the language:

Java Language Specification [12], §11.3, §11.3.1

"The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception [...]
when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated."

Thus, exceptions have two aspects in Java:

- they change the control flow of a program,
- they are a particular kind of side-effect (i.e. an exception object is created), and they prevent program parts from having side-effects.

While we deliberately neglect the latter aspect in our model (which can be handled in a Hoare Calculus on full Java, for example, when integrating our expression language into the statement language), we have to cope with the former aspect since it turns out to have dramatic consequences for the rules over Java expressions (these effects have not been made precise in the JLS).

So far, our normal behavior model is a completely denotational model; each expression is assigned a value by our semantic definitions. We maintain this denotational view, with the consequence that we have to introduce *exceptional values* that are assigned to expressions that "may [not] appear to have been evaluated". In the language fragment we are considering, only one kind of exception may occur:

Java Language Specification [12], §4.2.2

"The only numeric operators that can throw an exception (§11) are the integer divide operator / (§15.17.2) and the integer remainder operator % (§15.17.3), which throw an ArithmeticException if the right-hand operand is zero."

In order to achieve a clean separation of concerns, we apply the technique developed in [7]. Conceptually, a theory morphism is used to convert a normal behavior model into a model enriched by exceptional behavior. Technically, the effect is achived by redefining all operators such as $+,-,*$ etc. using "semantical wrapper functions" and the normal behavior definitions given in the previous chapters. Two types of theory morphisms can be distinguished: One for a *one-exception world*, the other for a *multiple-exception world*. While the former is fully adequate for the arithmetic language fragment we are discussing throughout this paper, the latter is the basis for future extensions by e.g. array access constructs which may raise *out-of-bounds exceptions*. In the following, we therefore present the former in more detail and only outline the latter.

6.1 The One-Exception Theory Morphism

We begin with the introduction of a type constructor that disjointly adds to a type α a failure element such as $\perp$ (see e.g. [27], where the following construction is also called "lifting"). We declare a type class *bot* for all types

containing a failure element $\bot$ and define as *semantical combinator*, i.e. as "wrapper function" of this theory morphism, the combinator *strictify* that turns a function into its *strict extension* wrt. the failure elements:

$$\begin{array}{ll} \text{strictify} & :: ((\alpha{::}\text{bot}) \Rightarrow (\beta{::}\text{bot})) \Rightarrow \alpha \Rightarrow \beta \\ \text{strictify f x} & \equiv \text{if x}{=}\bot \text{ then } \bot \text{ else f x} \end{array}$$

Moreover, we introduce the definedness predicate DEF :: α::bot $\Rightarrow$ bool by DEF x $\equiv$ (x $\neq \bot$). Now we introduce a concrete type constructor that lifts any type α into the type class bot:

$$\textbf{datatype} \quad \text{up}(\alpha) = \lfloor\_\rfloor\ \alpha \ \mid\ \bot$$

In the sequel, we write $t_\bot$ instead of up(t). We define the inverse to the constructor $\lfloor\_\rfloor$ as $\lceil\_\rceil$. Based on this infrastructure, we can now define the type JAVAINT that includes a failure element:

$$\textbf{types}\ \mathsf{JAVAINT} = \mathsf{JavaInt}_\bot$$

Furthermore, we can now define the operations on this enriched type; e.g. we convert the JavaInt unary minus operator into the related JAVAINT operator:

constdefs

$$\begin{array}{ll} \text{uminus} & :: \ \mathsf{JAVAINT} \Rightarrow \mathsf{JAVAINT} \\ \text{uminus} & \equiv \ \text{strictify}(\lfloor\_\rfloor \ \circ \text{ uminus} \circ \lceil\_\rceil) \end{array}$$

As a canonical example for binary functions, we define the binary addition operator by (note that Isabelle supports overloading):

$$\begin{array}{l} \textbf{op} + :\ [\mathsf{JAVAINT},\mathsf{JAVAINT}] \Rightarrow \mathsf{JAVAINT} \\ \textbf{op} + \equiv \text{strictify}(\lambda\ \text{X. strictify}(\lambda\ \text{Y. } \lfloor\lceil \text{X} \rceil + \lceil \text{Y} \rceil\rfloor)) \end{array}$$

All binary arithmetic operators that are strict extensions like $-$ or $*$ are constructed analogously; the equality and the logical operators like the strict logical AND & follow this scheme as well. For the division and modulo operators / and %, we add case distinctions whether the divisor is zero (yielding $\bot$). Java's non-strict logical AND && is defined in our framework by explicit case distinctions for $\bot$.

This adds new rules like X + $\bot$= $\bot$ and $\bot$ + X = $\bot$. But what happens with the properties established for the normal behavior semantics? They can also be lifted, and this process can even be automated (see [7] for details). Thus, the commutativity and associativity laws for normal behavior, e.g. (X:: JavaInt) + Y = Y + X, can be lifted to (X:: JAVAINT) + Y = Y + X by generic proof procedure establishing the case distinctions for failures. However, this works smoothly only if all variables occur on both sides of the equation; variables only occurring on one side have to be restricted to be defined. Consequently, the lifted version of the division theorem looks as follows:

$$[\![\ \text{DEF Y; Y} \neq 0\ ]\!] \Longrightarrow ((\text{X}{::}\ \mathsf{JAVAINT})\ /\ \text{Y}) * \text{Y} + (\text{X}\ \%\ \text{Y}) = \text{X}$$

6.2 The Multiple-Exception Theory Morphism

The picture changes a little if the semantics of more general expressions are to be modelled, including e.g. array access which can possibly lead to out-of-bounds exceptions. Such a change of the model can be achieved by exchanging the theory morphism, leaving the normal behavior model unchanged.

It suffices to present the differences to the previous theory morphism here. Instead of the class bot we introduce the class exn requiring a family of undefined values $\bot_e$. The according type constructor is defined as:

$$\textbf{datatype} \quad \mathrm{up}(\alpha) = \lfloor\_\rfloor\ \alpha \ \mid\ \bot \text{ exception}$$

and analogously to $\lceil\_\rceil$ we define exn-of$(\bot_e) = e$ as the inverse of the constructor $\bot$; exn-of is defined by an arbitrary but fixed HOL-value *arbitrary* for exn-of$(\lfloor_\rfloor) = arbitrary$. Definedness is $\mathrm{DEF}(x) = (\forall e.x \neq \bot_e)$.

The definition of operators is analogous to the previous section for the canonical cases; and the resulting lifting as well. Note, however, that the lifting of the commutativity laws fails and has to be restricted to the following:

$$[\![\ \mathrm{DEF\ X} = \mathrm{DEF\ Y} \wedge \text{exn-of X} = \text{exn-of Y}\ ]\!] \Longrightarrow (\mathrm{X}::\mathsf{JAVAINT}) + \mathrm{Y} = \mathrm{Y} + \mathrm{X}$$

These restrictions caused by the lifting reflect the fact that commutativity does not hold in a multi-exception world; if the left expression does not raise the same exception as the right, the expression order cannot be changed.

Hence, our proposed technique to use a theory morphism not only leads to a clear separation of concerns in the semantic description of Java, but also leads to the systematic introduction of the side-conditions of arithmetic laws in Java that are easily overlooked.

7 Conclusions and Future Work

In this paper we presented a formalization of Java's two's-complement integral types in Isabelle/HOL. Our formalization includes both normal and exceptional behavior. Such a formalization is a necessary prerequisite for the verification of efficient arithmetic Java programs such as encryption algorithms, in particular in tools like Jive [20] that generate verification conditions over arithmetic formulae from such programs.

Our formalization of the normal behavior is based on a direct analysis of the Java Language Specification [12] and led to the discovery of several underspecifications and ambiguities (see 3.1 (4), 4.1, 4.2, 5.1). These underspecifications are highly undesirable since even compliant Java compilers may interpret the same program differently, leading to unportable code. In the future, we strongly suggest to supplement informal language definitions by machine-checked specifications like the one we present in this paper as a part of the normative basis of a programming language.

We applied the technique of mechanized theory morphisms (developed in [7]) to our Java arithmetic model in order to achieve a clear separation of concerns between normal and exceptional behavior. Moreover, we showed that the concrete exceptional model can be exchanged — while controlling the exact side-conditions that are imposed by a concrete exceptional model. For the future, this leaves the option to use a lifting to the *exception state monad* [18] mapping the type JAVAINT to state $\Rightarrow$(JavaInt$_\perp$,state) in order to give semantics to expressions with side-effects like `i++ + i`.

Of course, more rules can be added to our theory in order to allow effective automatic computing of large (ground) expressions — this has not been in the focus of our interest so far. With respect to proof automation in JavaInt, it is an interesting question whether arithmetic decision procedures of most recent Isabelle versions (based on Cooper's algorithm for Presburger Arithmetic) can be used to decide analogous formulas based on machine arithmetic. While an *adoption* of these procedures to Java arithmetic seems impossible (this would require cancellation rules such as $(a \leq b) = (k \times a \leq k \times b)$ for nonnegative k which do not hold in Java), it is possible to retranslate JavaInt formulas to standard integer formulas; remainder sub-expressions can be replaced via $P(a \bmod b) = \exists m.\ 0 \leq m < a \wedge (a - m) \mid b \wedge P(m)$, such that finally a Presburger formula results. Since a translation leads to an exponential blow-up in the number of quantifiers (a critical feature for Cooper's algorithm), it remains to be investigated to what extent this approach is feasible in practice.

References

[1] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Int. Conf. on Computer Aided Design*. IEEE Computer Society, 1995.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.

[3] M. Balser et al. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*, LNCS 1783, 2000.

[4] B. Beckert and S. Schlager. Integer arithmetic in the specification and verification of Java programs. In *FM-TOOLS*, 2002.

[5] J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS01*, LNCS 2031, 2001.

[6] Dider Bondyfalat. Long integer division in Coq (algorithm divide and conquer). http://www-sop.inria.fr/lemme/Didier.Bondyfalat/DIV/.

[7] A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In *Types for Proof and Programs*, LNCS, 2003.

[8] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *Higher Order Logic Theorem Proving and its Applications*, 1995.

[9] A. C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. TR 512, University of Cambridge, 2001.

[10] D. Goldberg. Computer arithmetic. In *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.

[11] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification – Second Edition*. Addison-Wesley, 2000.

[13] J. Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 1690, 1999.

[14] Marc Herbstritt. e-mail communication, May 2003. Chair of Computer Architecture, Uni Freiburg.

[15] S. Höreth and R. Drechsler. Formal verification of word-level specifications. In *IEEE Design, Automation and Test in Europe (DATE)*, 1999.

[16] Bart Jacobs. Java's integral types in PVS. *Submitted*, 2003.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999.

[18] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL'95: Principles of Programming Languages*, 1995.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[20] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS00*, LNCS 276, 2000.

[21] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer, 1994.

[22] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, TU München, 1998.

[23] D. Sannella and A. Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Computing Surveys*, 31(3es), 1999.

[24] C. Scholl, B. Becker, and T. Weis. On WLCDs and the complexity of word-level decision diagrams — a lower bound for division. *Formal Methods in System Design*, 20(3), 2002.

[25] Sun Microsystems, Inc. *Java Card^{TM} 2.1.1 Specifications – Release Notes*, 2000.

[26] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, 2002.

[27] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

Part II

Selected Papers: Special Deduction for Method Support

Correct and User-Friendly Implementations of Transformation Systems[1]

Kolyang[†], T. Santen[‡], B. Wolff[†]

[†]Universität Bremen , FB3
P.O. Box 330440
D-28334 Bremen
{bu,kol}@informatik.uni-bremen.de

[‡]GMD FIRST Berlin
Rudower Chaussee 5
D-12489 Berlin
santen@first.gmd.de

Abstract. We present an approach to integrate several existing tools and methods to a technical framework for correctly developing and executing program transformations. The resulting systems enable program derivations in a user-friendly way.
We illustrate the approach by proving and implementing the transformation Global Search on the basis of the tactical theorem prover Isabelle. A graphical user-interface based on the X-Window toolkit Tk provides user friendly access to the underlying machinery.

1 Introduction

Development by transformation is a prominent approach in formal program development (CIP [Bau+85], PROSPECTRA [HK 93], KIDS [Smi 90]). Many case studies have proven its feasibility and demonstrated how much more abstract and user-oriented developments could be achieved than using usual post-verification approaches (fundamental for systems like PVS [OSR 93]). One recent case study is [KW 95]; and a prominent one is [SPW 95] where a strategic transportation scheduling algorithm is developed which is 200 times faster than the ones in practical use today. Unfortunately, implementations of transformation systems tend to be complicated and insecure. The correctness issue of transformation rules is usually not treated at the implementation level of existing systems.

In contrast to this, there is a family of "tactical theorem provers" in the tradition of LCF available with systems like *HOL* [GM 93] and *Isabelle* [Pau 94a], that are both well-designed and powerful. Coming with an open system-design going back to Milner, they allow for user-programmed extensions in a logically sound way. But there is recent prominent criticism that these provers, because of their "academic (ivory tower) origins", have "historically placed more emphasis on logical foundations and less on usability" [Gor 95]. This is clearly one of the reasons for the small acceptance of these provers in industry up to now.

In this paper, we demonstrate a technique to combine these two approaches. It results in a *simple* implementation design, in proven correct transformations which are easy to extend and to modify, and in a graphical user-interface that allows developers to profit from the abstraction of the transformational approach. We claim that our

---

[1] This work has been supported by the BMBF projects **UniForM** [Kri+95] and **ESPRESS**.

technique is applicable in a fairly wide range of problems, simply by modifying and extending our prototype implementation.

Our work integrates three existing and well documented public domain tools — the tactical theorem prover Isabelle based on *Standard ML* [HMM 86] and the X-Window toolkit *Tk* [Ous 94]. As object-language, we chose higher-order logic (HOL) which is one instantiation of the generic system Isabelle with an object logic and is delivered with the standard package. A subset of HOL formulas can easily be translated into functional programs (e.g. ML).

The basic idea of our approach is to separate the *logical core* of a transformation from the pragmatics of its application. As a *synthesis theorem* it can be proven correct independently in the logics of the object language, while the *tactical sugar*, which often highly system dependent, is concerned with the concrete application in a development context, i.e. the construction of suitable substitutions, "hard-wired" quantifier eliminations and standard simplifications, together with user interaction to control this process. The distinction between synthesis theorem and tactical sugar establishes an important separation of concerns.

We illustrate our approach by the transformation *Global Search* [Smi 87] that converts a non-constructive specification into a constructive one. This complex transformation has the character of a "design tactic" [Smi 90] or "design method" [HK 93]. Other transformations like *Divide-and-Conquer* or *Split of Postcondition* and elementary transformations like *recursion removal* or *fusion* also fit into our framework.

We proceed as follows: after introducing the idea of synthesis theorems and a brief presentation of Isabelle, we present Global Search as a synthesis theorem and prove it correct within Isabelle/HOL. We sketch several possibilities of tactical programs for our synthesis theorem. The resulting system is embedded into a user interface. Lastly, a small application example demonstrates the use of the resulting prototype system.

2 Transformations as Synthesis Theorems

2.1 The Concept

The core of our presentation is a general scheme of synthesis theorems, i.e. of logical formulas. The automatic construction of substitutions and other deduction-technical machinery, for short — the tactical sugar — is discussed in section 3.5.

Our intuition of "performing a program transformation" motivates several key notions (cf. [HK 93]). A transformation is composed of an *input pattern I* which is matched against an application context of a specification. This pattern is designed to be as general as possible and at the same time to be best supportable by automatic matching procedures (which belong to the tactical sugar). From the result of matching *I* against the specification at a specific position, an instance of the *output pattern O* is constructed automatically. All side conditions that can not be treated by automatic procedures and require theorem proving are collected in the *applicability condition V*. Usually, the output pattern contains function symbols that are introduced by the application of the transformation. They represent the design decisions of the whole transformation step, i.e. auxiliary functions whose definitions have to be provided by the user. These items are called the *parameters* $P_1 \ldots P_n$ of the transformation.

On the logical side, these items can be organised as a conditional equation:

$$\forall P_1 .. P_n .\, V \Rightarrow I \rightsquigarrow O$$

where $\leadsto$ is a transitive binary operator that typically stands for

- logical equality or equivalence in case of symmetric transformations or
- the implication $\Leftarrow$ in case of classical refinement (the input pattern has to follow from the output; in algebraic jargon: the model class of the output specification is included in the model class of the input specification) or
- the Scott-definedness ordering $\sqsubseteq$ in case of "robust implementations" using object-logics like LCF (see [Pau 94a]).

This scheme is strong enough to capture a large variety of transformations — from "Filter Fusion" [BM 93] to "Split of Postcondition" [HK 93]. These have been presented in [KW 95]. The synthesis theorem for Global Search is discussed later.

The separation into synthesis theorem and tactical sugar has the following consequence for the soundness of a transformation: The logical core can be proven *within* the logic in which it is represented. This can be done by showing that the synthesis theorem follows from the basic axioms of the logic — or, in other words, the synthesis theorem follows from a *conservative extension of the core logic* (see below).

2.2 Introduction to Isabelle

Isabelle is a *generic* theorem prover that supports a family of logics, e.g. first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), the Logic of Computable Functions (LCF), and others. We only use its set-up for higher-order logic (HOL). Isabelle supports natural deduction style. Its principal inference techniques are resolution (based on higher-order unification) and term-rewriting. Isabelle provides syntax for hierarchical theories (containing signatures and axioms).

As an example, let us create the theory List0 from the theory HOL that contains the basic rules of the logic. All input in the form of UNIX files or user input will be denoted with this FONT — enriched by the usual mathematical notation for $\forall$, $\exists$,... instead of ASCII-transcriptions.[2] We define the unary type constructor list, its constructors ([], #) and the concatenation @:

```
List0 = HOL +
      types     list        1
      arities   list        :: (term)term
      consts    "[]"        :: "α list"                    ("[]")
                "#"         :: "[α, α list] → α list"      (infixr 70)
                "@"         :: "[α list, α list] → α list" (infixl 60)
rules
      app_mt          "[]@m = m"
      app_cons        "(a#n)@m = a#(n@m)"
end
```

Here, list belongs to the type universe term of HOL and accepts types from term. This construct is a tribute to the genericity of Isabelle. "→" is the function space constructor, and the brackets denote curried functions: [α, α list] → α list is equivalent to α→α list→α list. The equality "=" stems from HOL, while "≡" is

---

2 We do not distinguish quantifications and implications at the different logical levels throughout this paper; see [Pau 94a].

used to denote the definitional equality. The comments ("[]") and (infixl 70) are pragmas setting up the parsing and pretty-printing machinery of Isabelle.

2.3 The Logic HOL

In this section, we will give a short overview of the concepts and the syntax. Our object-language HOL goes back to [Chu 40]; a more recent presentation is [And 86]. In the formal methods community, it has achieved some acceptance, especially in hardware-verification. HOL is a classical logic with equality. It is based on total functions denoted by λ-abstractions like "$\lambda x.x$". Function application is denoted by $f\ a$. Although its type discipline incorporates polymorphism with type-classes (as in Haskell), in this paper we only use Milner-Polymorphism (as in ML).

Logical rules of HOL like:

$$\frac{P \quad Q}{P \wedge Q}\ (\text{conjI}) \qquad \frac{P \wedge Q}{P}\ (\text{conjunct1})$$

will be represented in Isabelle by

```
[| ?P ; ?Q |] ⟹ ?P ∧ ?Q        ?P ∧ ?Q ⟹ ?P
```

where variables prefixed by a question mark are called *meta-variables*. Their exact meaning in the deduction process is discussed later.

2.4 Proving in Isabelle

Isabelle as a system is a set of function definitions in the ML-environment (or: "data-base"). They represent a collection of function- and data type definitions. Most notable are the three mutually dependent data types: tactic, thm and (internal) proof_state.

Isabelle supports two styles of theorem proving: forward proof and backward proof.

Backward proving in Isabelle. The general scheme of a backward proof consists of three steps:

(1) The initialisation of the internal "proof-state" with the formula to be proven (the "goal"). It is done by the operation:

```
goal: theory -> string -> thm list
```

where the string contains the textual representation of the formula to be type-checked and proved within theory.

(2) A refinement of the proof-state. It is performed with the operation:

```
by : tactic -> unit
```

This refinement can be seen as a transformation of the proof-state by means of tactics and already proven theorems. Two of these are the following pivotal *built-in* tactics

```
atac: int -> tactic        rtac: thm -> int -> tactic
```

The integer parameters specify the subgoal to which the tactic is applied. They encapsulate the Isabelle meta-inferences *assumption* (basically "A" implies "A" modulo unification) and *resolution* (essentially "A ⇒ B" and "B ⇒ C" implies "A ⇒ C" modulo unification)

(3) The extraction of a theorem produced out of a proof-state with no subgoals:

```
result: unit -> thm;
```

returns a value that can be bound to an arbitrary ML-identifier.

By composition of these operations on the proof-state, large proof-scripts can be organised in the *.ML files that are executed automatically when loading a theory.

Forward proving in Isabelle. Forward reasoning mimics the classical way of constructing proof trees. The combination of two rules, say conjI and conjunct1 given above, can be done by using the resolution combinator (involving unification):

```
RS : thm * thm -> thm
```

in the form:

```
conjI RS conjunct1
```

which is evaluated by Isabelle to the derived rule:

```
[| ?P; ?Q |] ⟹ ?P
```

As simple example for higher-order unification, we consider the specialisation rule:

```
∀ x. ?P x ⟹ ?P ?x                                    (spec)
```

With this HOL rule it is possible via forward proving in a theorem, say ∀ y. y = a, to eliminate the quantification and to replace the bound variable y by the *meta-variable* ?x. The resulting formula would be ?x = a. We can interpret the meta-variable ?x as a "hole" in the formula that can be filled later by substitutions (usually produced as a consequence of unification inside atac and rtac). This possibility of "postponing substitutions" and of transforming theorems in a programmed, but logically sound way, is important for our approach.

Isabelle provides substantially more machinery, especially for people who want to set-up their own logic or who yearn for a higher degree of automatisation. However, with the subset presented here, extended by some variants, it is already possible to perform substantial proofs and to describe the relevant operations in this paper.

2.5 Conservative Extension in HOL

The introduction of new axioms ("rules" in case of List0) while building a new theory is an extremely dangerous method, since the resulting theory may easily be inconsistent. Hence it is necessary to recall that there is a number of syntactic schemes for specification-extension that maintain the consistency of the extended one. (For a more formal and very readable account on "conservative extensions schemes" the reader is referred to [GM 93]). Some schemes are:

- the *constant definition* "$c \equiv t$" or "$c\ x \equiv t\ x$" of a fresh constant symbol c by a closed expression t not containing c,
- the *type definition* (a set of axioms stating an isomorphism between a non-empty subset of a base-type and the new type to be defined),
- a set of equations forming a *primitive recursive scheme* over a fresh constant symbol f,
- a set of equations forming a *well-founded recursive scheme* over a fresh symbol f.

The basic idea of these extension schemes is to avoid general recursion. Instead, they introduce axioms only in a controlled way. The desired properties have to be derived from these. Building up large theories by methodically using conservative extensions used to be a quite tedious enterprise, but recent advances in the Isabelle implementation have substantially improved the support for this approach [Pau 94b].

3 Global Search Transformation

We use the approach sketched in section 2 to implement a transformation based on the theory of *Global Search* algorithms that has been developed at Kestrel Institute and implemented in the *Kestrel Interactive Development System* (KIDS) [Smi 87, Smi 90]. After presenting the basic idea of global search, we show how the theory can be formalised in Isabelle/HOL. We prove a synthesis theorem under the resulting theory, and finally provide a tactic program (the sugar) converting the synthesis theorem into an executable transformation.

3.1 The Algorithm Design Theory *Global Search*

The global search theory characterises a large class of algorithmic problems that are solvable by search or optimisation algorithms. It covers problems typically solved, e.g., by backtracking, branch-and-bound, or simplex algorithms.

For the purpose of this paper, we closely stick to the notation used in [Smi 87]. There, algorithms are described by input / output predicates. A *problem specification* is a quadruple $P = \langle D,R,I,O\rangle$ where D is the input domain and R is the output range of the function f to synthesise. The predicate I describes the admissible inputs, and O describes the input / output behaviour of f. Hence, f is a solution to P if

$$\forall\, x{:}D.\;\; I(x) \wedge y = f(x) \Rightarrow O(x, f(x))$$

A *design theory* extends a problem specification by additional functions. It states sufficient properties of these functions to formulate a schematic algorithm that solves the problem correctly. The basic idea of global search is to associate inputs x with *search spaces* that initially contain all solutions z with $O(x,z)$. Search is then performed by splitting search spaces into "smaller" ones until solutions are directly extractable. This idea is captured in the design theory of Figure 3.1.1.

sorts D, R, R'

operations

$I : D \to \text{Bool}$	$Satisfies : R \times R' \to \text{Bool}$
$O : D \times R \to \text{Bool}$	$Split : D \times R' \times R' \to \text{Bool}$
$I' : D \times R' \to \text{Bool}$	$Extract : R \times R' \to \text{Bool}$
$r'_0 : D \to R'$	$< : R' \times R' \to \text{Bool}$

axioms

GS0 $I(x) \Rightarrow I'(x,(r'_0(x))$

GS1 $I(x) \wedge I'(x,r') \wedge Split(x,r',s') \Rightarrow I'(x,s') \wedge s' < r'$

GS2 $I(x) \wedge O(x,z) \Rightarrow Satisfies(z,r'_0(x))$

GS3 $I(x) \wedge I'(x,r') \Rightarrow Satisfies(z,r') = (\exists\, s'.\; Split^*(x,r',s') \wedge Extract(z,s'))$

GS5 $<$ is a well-founded ordering on R'

Figure 3.1.1: Global Search Theory.[3]

The sort R' is the type of search space descriptors, I' defines legal descriptors. For an input x, r'_0 and *Split* describe the search tree for solutions z with $O(x,z)$: its root is

[3] In [Smi 87], axiom GS4 deals with necessary filters, which we do not consider in this paper. Still, we call our last axiom GS5 to stay consistent with the literature.

$r'_0(x)$, the initial search space; a descendant relation on nodes is given by *Split*: *Split*(x,r',s') is true if s' is a (direct) subspace of r' for an input x. *Split** is defined by

$$Split^*(x,r',s') = (\exists\, k{:}\mathbb{N}.\ Split^k(x,r',s'))$$
$$Split^0(x,r',s') = (r' = s')$$
$$Split^{k+1}(x,r',s') = (\exists\, t'.\ Split(x,r',t') \wedge Split^k(x,t',s'))$$

The possible solutions that can be extracted from a node r' are *Extract*(z,r').

Axioms GS0 and GS1 ensure that all considered search spaces are legal. Axiom GS1 additionally ensures that search spaces can be split only finitely often, i.e. that the search tree has a finite depth. GS2 requires the initial search space to contain all feasible solutions.

By axiom GS3, *Satisfies*(z,r') describes the solutions z contained in a search space r' that can be found with finite effort: there must exist a finite path in the search tree from r' to a search space s' from which z can be extracted.

Under the global search theory, we can use *Split* and *Extract* to get an algorithm schema satisfying the problem specification $\langle D,R,I,O\rangle$. Following [Smi 87], we express the algorithm by input / output predicates.

$$I(x) \wedge I'(x,r') \Rightarrow Fgs(x,r',z) = (Extract(z,r') \wedge O(x,z) \vee (\exists\, s'.\ Split(x,r',s') \wedge Fgs(x,s',z)))$$
$$I(x) \Rightarrow F(x,z) = Fgs(x,r'_0(x),z)$$

F computes a solution z for some admissible input x by searching in the initial search space $r'_0(x)$. Searching is performed by the auxiliary function *Fgs*: if z is not directly extractable from r' this search space is split and its subspaces are searched.

The global search theory described above is relatively simple. More refined ones incorporate filters to prune search spaces. The most elaborate one stated in [SPW 95] uses a refinement relation on search spaces and cutting constraints to profoundly exploit the problem domain and synthesise highly efficient search algorithms.

3.2 Formalisation in Isabelle/HOL

How do we know that a particular application of Global Search is correct, i.e. how can we be sure that we get a correct implementation of our problem specification when we instantiate the abstract global search algorithm on the basis of particular I', r'_0, *Satisfies*, *Split*, *Extract* and < defined in our problem domain? There are two reasons why correctness might be spoiled: we may make a mistake in the particular application, e.g. choosing components that do not fulfil the global search axioms, or, more fundamentally, the implementation of the transformation may be faulty, i.e. the actually implemented transformation may be unsound. It is not in question here that "something" like the theory presented in section 3.1 describes a mathematically sound transformation. But it is a long way from a paper-and-pencil proven "idea" of a transformation to its actual implementation and application. We must make sure that the transition from idea to implementation is traceable and based on well-understood principles, and that it leads to a *soundly implemented* transformation.

In contrast to the KIDS system which is not based on a general logical framework but implements transformations like Global Search directly, we have chosen higher-order logic as implemented in Isabelle. Implementing the transformation in our system first of all means formalising the description of global search given in

section 3.1 in a Isabelle/HOL theory. The Isabelle theory is sketched in the following. It is based on the HOL theories of natural numbers and sets.

```
GlobalSearch = Nat + Set +
consts
    ...
    GS3 :: "[δ → bool,[δ, ρ'] → bool,
             [δ, ρ', ρ'] → bool,
             [ρ, ρ'] → bool,[ρ, ρ'] → bool] → bool"
    ...
defs
...
REC_def "REC Fgs I I' Extract Out Split ≡
                ∀ x r z . I x ∧ I' x r ⟹
                Fgs x r z = (Extract z r ∧ Out x z ∨
                             (∃ s'.Split x r s' ∧ Fgs x s' z))"

GS3_def "GS3 I I' Split Satisfies Extract ≡
                ∀ x z r'. I x ∧ I' x r' ⟹
                  Satisfies z r' = (∃ s'. rep_s Split x r' s' ∧ Extract z s')"
...
GSA_def "GSTHEORY I Out I' r0 Split Extract subspace Satisfies ≡
                GS0 I I' r0                          ∧
                GS1 I I' Split subspace              ∧
                GS2 I Out Satisfies r0               ∧
                GS3 I I' Split Satisfies Extract     ∧
                GS5 subspace"
```

Figure 3.2.1: Isabelle theory of Global Search.

Using the definitional equality ≡, we define higher-order predicates GS1 through GS5 for the global search axioms. Their conjunction GSTHEORY gives us a predicate that represents the global search axiomatization. We chose to formalise the parameter sorts *D*, *R* and *R'* of Figure 3.1.1 by making GS1 through GS5 polymorphic and use the type variables δ, ρ and ρ', respectively. In this way, we need not explicitly instantiate parameter sorts when applying the Global Search transformation: Isabelle's type inference system will find suitable sorts for us. The predicates rep_n and rep_s are defined as primitive recursors on Nat, which construct $Split^k$ and $Split^*$ from a given *Split*. REC provides an abbreviation of the characteristic equation for *Fgs*.

Building the theory in this way ensures consistency since each axiom is formalised as a conservative constant definition. We indicate this by the keyword defs, and the system checks for conservativity of these axioms as sketched in section 2.5.

3.3 The Global Search Synthesis Theorem

The following synthesis theorem for global search is based on theory GlobalSearch:

```
        ∀ I' r0 Split Extract subspace Satisfies.
          GSTHEORY I Out I' r0 Split Extract subspace Satisfies ⟹
            (I x ⟹ F x z = Out x z)
(GS)        =
            (I x ⟹ (∃ Fgs. REC Fgs I I' Extract Out Split ∧ F x z = Fgs x (r0 x) z ))
```

Assuming a global search theory, (GS) relates the problem specification to the schematic search algorithm. The function Fgs we get when composing I, I', Extract, Out, and Split according to REC finds exactly the solutions z that are specified by Out if the search starts with the initial search space r0(x) for some legal input x.

Note that (GS) has the form of synthesis theorems introduced in section 2.1. The components of the problem specification are free variables, while the bound variables I' through Satisfies serve as parameters to the transformation obtained from (GS).

What are axioms in the theory of Figure 3.1.1 appears as the premise (GSTHEORY ...) in the implication of (GS). Therefore, an inconsistency in Figure 3.1.1 can not affect the "global" consistency of the Isabelle theory we are working in. If the theory of global search algorithms were inconsistent, this would only affect the applicability of the global search transformation: the synthesis theorem would trivially hold but the corresponding transformation could never be applied.

3.4 Mechanical Proof of the Synthesis Theorem

Figure 3.4.1 shows the structure of the proof of (GS) that we have carried out in Isabelle. To keep the picture readable, we omit most of the functions' parameters.

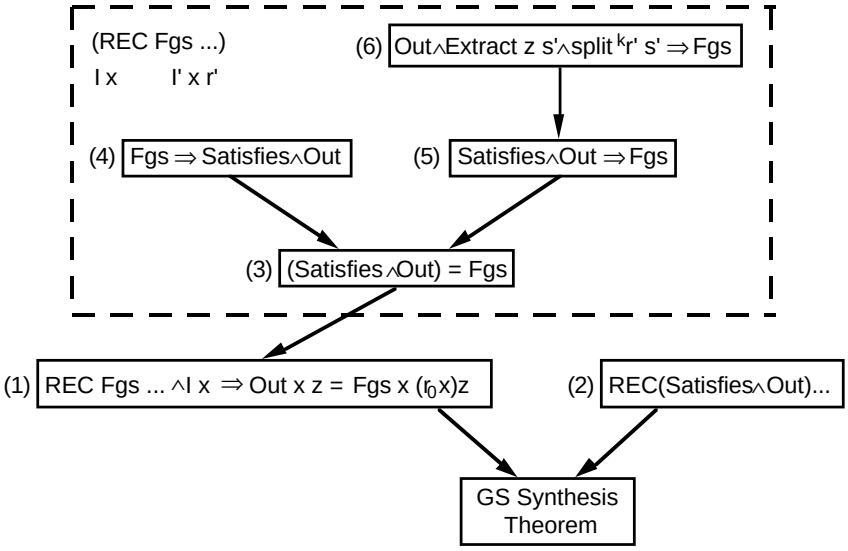


Figure 3.4.1: Proof Structure

The proof proceeds backwards in a goal-directed fashion. The first steps are to apply the introduction rules for universal quantification and implication and exhibit the equality

```
   (I x ⟹ F x z = Out x z)
=
   (I x ⟹ (∃ Fgs. REC Fgs I I' Extract Out Split ∧ F x z = Fgs x (r0 x) z ))
```

We prove this equality by mutual implication. The "right-to-left" direction is the hard one, which after some simplification reduces to Lemma (1):

```
(1)   I x ∧ REC Fgs I I' Extract Out Split ⟹ Out x z = Fgs x (r0 x) z
```

The central proof-idea is to find a closed form for the recursively defined Fgs.

(3) REC Fgs ... ∧ ... ⇒ (Satisfies z r' ∧ Out x z) = Fgs x r' z

This lemma says that any function Fgs satisfying the recursive equation REC behaves like the conjunction of Satisfies and Out. By GS3, Satisfies z r' means that we only need finitely many applications of Split to find a subspace of r' where we can extract z from. On the other hand, Fgs x r' z is defined by recursively splitting r' and extracting solutions z that additionally fulfil Out x z — the latter condition being the only intuitive difference between the two predicates. Once we have proved (3), we can use GS2 to specialise it and show (1).

With Lemma (3) in mind, it is easy to prove the "left-to-right" direction of (GS). Here, we basically have to show that there exists a function Fgs fulfilling REC. From (3) we know that *if* a function Fgs satisfies REC then it behaves like the conjunction of Satisfies and Out. So we use this conjunction — suitably abstracted — as a witness for the existential quantification and show that it indeed satisfies REC.

The proof of (3) does the real work. Here, we generally assume that Fgs satisfies REC, and that the input x and search space r' are admissible, which is indicated by the dashed frame in Figure 3.4.1. Again, we prove the equality by mutual implication and reduce (3) to (4) and (5). Both can be interpreted computationally. Lemma (4) deals with termination and correctness of solutions produced by Fgs.

(4) Fgs x r' z ⇒ (Satisfies z r' ∧ Out x z)

We not only have to show that all output z produced by Fgs is a feasible solution, i.e. Out x z holds, but also that it can be extracted from the input search space r' by finitely many Split's, i.e. Satisfies z r' holds. Here it is crucial that Split produces a decreasing chain of search spaces with respect to a well-founded ordering (cf. GS1 and GS5). Only this requirement allows us to interpret REC as a definition of a recursive function. Otherwise predicates that are true on cycles of Split's where Extract is false would satisfy REC. GS5 allows us to use a theory of well-founded sets that comes with Isabelle/HOL: we prove (4) by well-founded induction on r'.

Lemma (5) deals with completeness of the set of solutions produced by Fgs: all feasible solutions are indeed found by Fgs.

(5) (Satisfies z r' ∧ Out x z) ⇒ Fgs x r' z

The proof idea for (5) is induction on the length of search paths, i.e. the number *k* of Split's leading to the search space from which the solution z can directly be extracted. Lemma (6) formally captures this idea. It is gained from (5) by unfolding the definitions of Satisfies and rep_s, i.e. $Split^*$.

Global search is an example of a non-trivial transformation. The entire proof script for (GS) consists of about 140 tactics' applications. Isabelle under Standard ML of New Jersey takes about 60 CPU seconds to execute it on a Sun Sparc 5 workstation. We needed several attempts to develop the global search theory and the proof of the synthesis theorem in Isabelle. The first version of the theory was non-conservative and explicitly introduced parameter sorts *D*, *R* and *R'*. We then abolished these sorts and used polymorphism. The next stage in the theory development was to come to the conservative theory sketched in Figure 3.2.1.

The proofs had to be adapted to each rephrasal of the theory. The structure of the proofs also changed several times due to new proof ideas — the latest being to intro-

duce Lemma (3) – and due to changes in the formulation of the synthesis theorem: the first version only was an implication from the algorithm schema to the problem specification. In this version, we also left out the precondition I x ∧ I' x r in the definition of REC. This formulation of the theorem was still correct but its premises would have been too strong to be practically useful. Only after we introduced Lemma (3) and tried to prove equality instead of implication, we became aware of the missing precondition.

Despite of all these changes to the theory, it was relatively easy to adapt the proof scripts. Simple "replay until failure" was usually sufficient to find the points were changes had to be made, and these were mostly local ones like inserting a tactic to re-establish some syntactic structure, that the next tactic depended on.

3.5 Tactical Sugar for Global Search

Global Search is used in algorithm construction by providing a mapping from GlobalSearch to an extension of the concrete problem theory such that the global search axioms are theorems under the extended problem theory. We can apply the same mapping to the schematic algorithm and get a solution for our problem, i.e. we have transformed the non-constructive problem specification into a constructive form. This algorithm is usually inefficient and has to be optimised by further transformations.

In [Smi 90, SPW 95], elaborate techniques to find a global search algorithm for a given problem specification are described. They are based on a library of global search theories that basically describe the structures of search trees for various data structures.

While it is certainly possible to implement these techniques in our framework, we focus on the description of the basics of our approach and restrain ourselves to much simpler tactical sugar: the proven synthesis theorem is used to define an ML function

```
fun GLOBAL_SEARCH : nat * string list → tactic
```

that takes the subgoal number and the list of parameters to produce a tactic. This function successively removes the universal quantification via forward proof and rule spec (see section 2.4). Similarly, the implication and the equality are converted by application of the modus ponens and substitutivity rule. These operations convert the synthesis theorem into the following version:

```
[| GSTHEORY ?I ?Out ?I' ?r0 ?Split ?Extract ?subspace ?Satisfies ;
  ?I ?x  ⇒ (∃ Fgs. REC Fgs ?I ?I' ?Extract ?Out ?Split ∧
                   ?F ?x ?z = Fgs ?x (?r0 ?x) ?z) |]
⇒ (?I ?x ⇒ ?F ?x ?z = ?Out ?x ?z)
```

Furthermore, GLOBAL_SEARCH successively substitutes the parameters (after parsing and typechecking the string list) into the meta-variables. Finally, the result is applied via rtac to a particular subgoal — this will set the remaining variables (?F, ?x and ?z) and complete the mapping to the problem theory.

There are different versions of tactical sugar conceivable — one could leave more parameters uninstantiated or massage the conclusion into a different syntactical form using more forward resolution steps. More complex tactics based on the synthesis theorem could employ the substitution rule for equality of higher-order logic. We would not break up the equality in (GS) but apply it to a subterm of a possibly complex goal. The choice how to come to the parameters of the global search theory would remain the same as before.

Note that the process of transforming the synthesis theorem into a logical rule which is applied by some ML function is "safe". The transformations used there are all based on proven correct rules and the primitive theorem manipulating functions of Isabelle, so nothing incorrect can happen — assuming the core of Isabelle is sound.

Proving the correctness of tactical sugar is not necessary in the sense that it simply controls the application of basic axioms and lemmas. This control may lead to a dead end and *fail*, or it might prove something that we did not want to prove, but it can never produce a proof for something that would not be provable in the theorem prover without this tactical program. Of course, the *implementation* for "apply an axiom" which represents the atoms of our tactical control programs might be incorrect or the basic rules of the logic might be unsound. But proving the correctness of "apply an axiom" in the absolute sense would require a formalisation and verification of the core theorem prover. As a consequence, this "meta-encoding" can not fully solve the problem since it raises the same problems of correctness on the meta-level.

4 YATS — the System

YATS can be regarded as a step towards an IFDSE (Integrated Formal Development Support Environment) following the philosophy of [BH 95] or [Kri+95]. Such systems support many stages of the formal development, from initial functional specifications, through design specifications and refinement. More elaborated systems will also provide a support for specification animation, version-management etc.

We believe that a high-quality graphical user interface (GUI) plays a key role for both the acceptance and productivity of an IFDSE. To date, there is no common agreement on what could be a good design of a GUI for a theorem prover or an IFDSE (we admit that we have not found the definite answer either), although there are recent remarkable efforts in this direction.

Our GUI should be completely independent from Isabelle and as independent as possible from our system environment and our hardware-platform. In the past, the development of many systems (PROSPECTRA, for example) has been trapped by their complex and monolithic design. An answer to this dilemma of monolithic designs can be a *heterogeneous* one with few complementary components that are systems in their own right. This way it is possible to integrate work of independent research groups. The main task of an heterogeneous design is to provide suitably abstract and flexible interfaces in order to enable an easy and stable integration of new versions of its components.

For our GUI, we chose the toolkit *Tk* [Ous 94] to achieve this goal. Although we wish to profit from Tk as a highly portable interface to X-Windows, we do not believe that the command-language Tcl on top of Tk should be used for larger software developments. The major reason is that Tcl supports only one data-structure — text — in a way similar to LISP and its lists. Tcl is an untyped language without data structures and lacks higher modularisation concepts.

For this reason, we implemented an SML interface for Tk, called *sml_tk*. It is a component in its own right and provides a toolkit for standard windows, e.g. a substitution window, that can be reused by research groups working on, e.g., another theorem prover interface. Based on sml_tk, the GUI itself is implemented as an SML functor, called *isawin*, which is parametrised by a list of tactical-sugar functions. The system YATS is an instantiation of this functor with a list containing the function

GLOBAL_SEARCH (see section 3.5). According to the instantiation, isawin automatically produces new interface components and new dialogues with the user — the implementor of a transformation has only to provide its proof and its tactical sugar.

The following diagram gives a short overview over the stack of main components (the size of the blocks roughly corresponds to their implementation size):

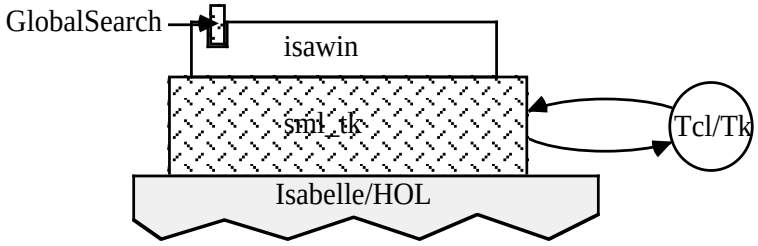


Figure 4.0.1: The System architecture

Note that the formal proof of Global Search and its integration into the system contributes only to a very small part to the whole system. This justifies our claim that the design allows the implementors to concentrate on the real intellectual problem of designing and verifying new transformations. Moreover, the instantiation of isawin with a new transformation is a question of a few seconds. Even if a very different state of technology (hardware and software) has to be taken into consideration, in comparison to PROSPECTRA, for example, where a complete recompilation was necessary that took 2 hours [GL 93], this is quite remarkable.

4.1 The interface sml_tk

Our interface evolved from an imperative version of a purely functional, monad-style encapsulation of Tk in Gofer [VTS 95]. It has a more functional flavour than the interface available for caml/light [PR 95]. It is characterised by the following features:

- abstract data-types for options, configurations, packing information
- abstract data-type for graphical objects, called *widgets*
- events on the interface (mouse clicks, key strokes, etc) are mapped to SML functions associated to widgets via *bindings*.
- a toolkit for defining a problem specific set of window types.

To give a flavour of programming in sml_tk, let us have a brief look at a fragment of the essential tree-like data type for widgets used to describe the content of windows:

```
datatype  Widget =
        Frame of WidId * Widget list * Pack list * Configure list * Binding list
      | Label of WidId * Pack list * Configure list * Binding list
      | Entry of WidId * Pack list * Configure list * Binding list
      ...

type Window  = (WinId * Title * (Widget)list * Action);
```

The following fragment is taken from the description of a small standard-window of the toolkit:

```
fun input enteraction =
      let fun mrs () = let val nm = selectTextAll "e1"
                       in  enteraction nm ();closeWindow "enter" end
      in  Entry("e1",[],[Width 20], [Bind("<Return>",mrs)]) end;
```

The function input yields an Entry-Widget [Ous 94], that represents a graphical field allowing to enter a string. It has the name e1 and a width of 20 characters. Associated to e1, there is a function mrs that is evaluated whenever the event <Return> happens. mrs selects the inserted text, passes it to the parameter function enteraction and closes the surrounding window called enter.

4.2 The GUI of YATS

The main window of YATS (as well as any other instance of isawin) consists of two major components: An edit-window (where the editing facilities like Cut-Copy-Paste are already provided by Tk without writing any additional line of code in the interface) and a prover-window to which the transformation facilities (a result of the instantiation of isawin) and the operations controlling Isabelle are associated. It is possible to browse theories and ML files in suitable subwindows with their associated axioms and theorems (see Figure 4.2.1 below). The user-interface for the prover part is not in the focus of this paper.

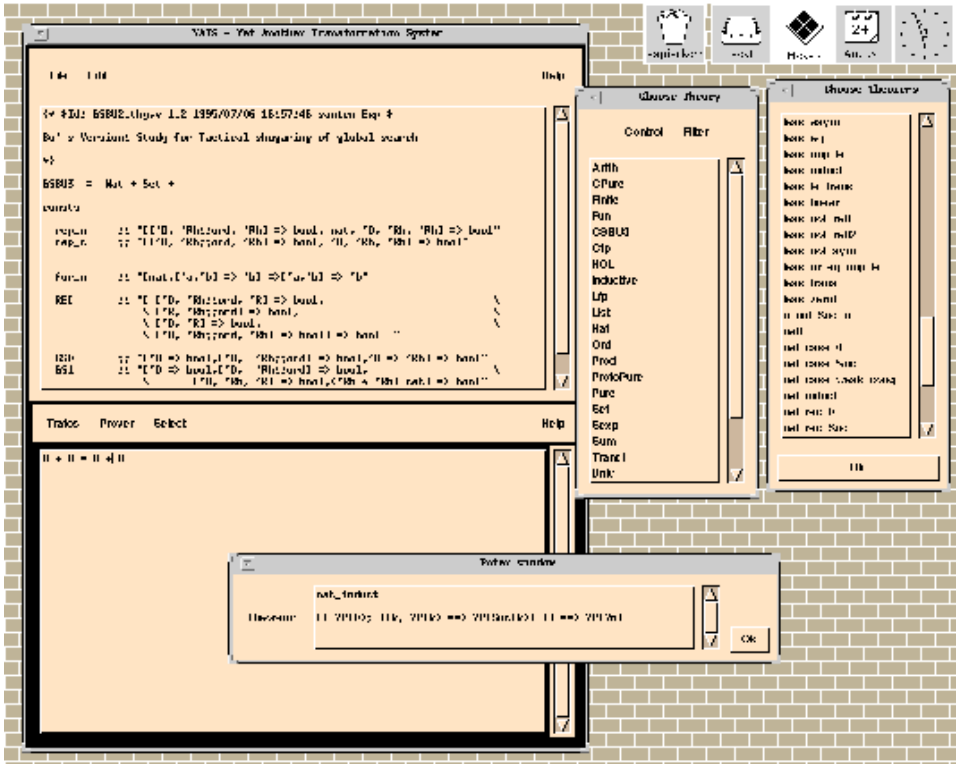


Figure 4.2.1: Screenshot of YATS (Overview)

Usually, by double-click on an arbitrary widget of the interface, the user can get more information, and by triple-clicks suitable operations on the activated unit are executed. For example, when pointing to a subgoal in the prover-state-widget, a double-click will inform the user on what transformation or Isabelle-command will be executed (due to settings and other information inferred by the system), while a triple-click performs this command.

4.3 An Application Example

In this section we use our system to develop a global search algorithm that enumerates all maps from a finite set *U* to a finite set *V*. We take the global search theory for this algorithm, *gs_finite_mappings*, from [Smi 87]. In KIDS, abstract and simple theories like *gs_finite_mappings* are used to describe search patterns on particular data structures. To develop a search algorithm for a particular problem with KIDS means to *specialise* such a "pattern theory" to the given problem specification (see [Smi 90]). Since the specialisation procedure as well as the pattern theories are hard-wired into KIDS, their correctness can not be guaranteed within the system. The development of "pattern theories" can not rely on specialisation but must use different tactical sugar like we have implemented in YATS.

The problem specification for *gs_finite_mappings* is based on a library theory of finite maps:

$F \mapsto$ fin_maps

$\delta \mapsto \alpha$ set $\times \beta$ set $\rho \mapsto (\alpha, \beta)$ Fmap

$I \mapsto \lambda (U,V).$ Fin $U \wedge$ Fin V Out $\mapsto \lambda (U,V)\ N.\ N \in$ Map(U,V)

We wish to synthesise a function called fin_maps that transforms pairs of sets over types α and β to finite maps whose domains are sets over α and whose ranges are sets over β, i.e. members of (α, β) Fmap. For finite input sets U and V the function must return a map N with domain U and a range in V. The predicate Map is defined by

$$Map(U,V) \equiv \{\ M \mid dom\ M = U \wedge \forall\ b \in\ dom\ M.\ \ M\ \hat{}\ b \in\ V\ \}$$

Note that we need an explicit operation ^ to apply maps because their type (α, β) Fmap is different form the HOL function type.

To develop an algorithm for the problem, we chose the theory of finite maps as the logical context (just by activating it via a mouse-click) and enter the (slightly massaged) specification as a goal into YATS as shown in Figure 4.3.1.

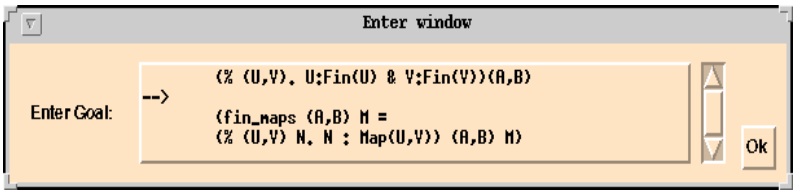


Figure 4.3.1: Entering the transformation goal

We now apply the transformation Globalsearch of section 3.5. First of all, this makes Isabelle match the goal with the transformation rule and set up the substitution of meta-variables and type variables for the problem specification. The next step is the creative part of the transformation: we have to provide the parameters that establish a global search theory. To date, as Figure 4.3.2 shows, this is done by explicitly entering a substitution for the parameters.

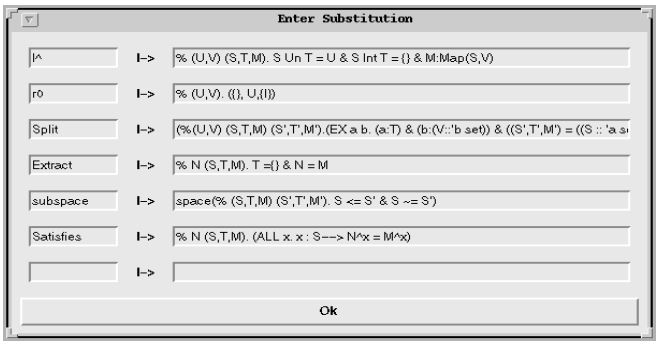


Figure 4.3.2: Entering parameter substitution

In mathematical notation, the substitution looks as follows:

$$
\begin{array}{ll}
I' & \mapsto \lambda\ (U,V)\ (S,T,M).\ S \cup T = U \wedge S \cap T = \emptyset \wedge M \in \mathsf{Map}(S,V) \\
r'_0 & \mapsto \lambda\ (U,V).\ (\emptyset, U, \{\!|\,|\!\}) \\
\mathsf{Split} & \mapsto \lambda\ (U,V)\ (S,T,M)\ (S',T',M').\ (\exists\ a\ b.\ a \in T \wedge b \in V \\
 & \qquad\qquad \wedge (S',T',M') = (S \cup \{a\}, T - \{a\}, M \oplus \{\!|a \mapsto b|\!\})) \\
\mathsf{Extract} & \mapsto \lambda\ N\ (S,T,M).\ T = \emptyset \wedge N = M \\
\mathsf{subspace} & \mapsto \mathsf{space}(\lambda\ (S,T,M)\ (S',T',M').\ S \subset S') \\
\mathsf{Satisfies} & \mapsto \lambda\ N\ (S,T,M).\ (\forall\ x \in S.\ N\ \hat{}\ x = M\ \hat{}\ x)
\end{array}
$$

The crucial part here is to find a representation of search spaces and a suitable Split based on that representation. Our search idea is to successively extend a partial map until its domain encompasses all of U. Therefore, we model search spaces by tri–ples (S,T,M) where S and T partition U and M is a map to V with domain S. Split–ting a search space means to extend M by a new pair mapping a not yet used member a of U to some arbitrary value b of V. The operation $\oplus$ overwrites the first map on the domain of the second, and $\{\!|a \mapsto b|\!\}$ maps a exactly to b. The subspace relation on search spaces is induced by the strict subset relation on S, which we for–malise using the library function space that converts an ordering function into its graph. Given the transformation above, Isabelle computes the type of search spaces auto–matically.

$\rho' \mapsto \alpha\ \mathsf{set} \times \beta\ \mathsf{set} \times (\alpha, \beta)\ \mathsf{Fmap}$

After type checking the parameter substitution, YATS responds with the following proof state, containing the proof obligations and the synthesised "program":

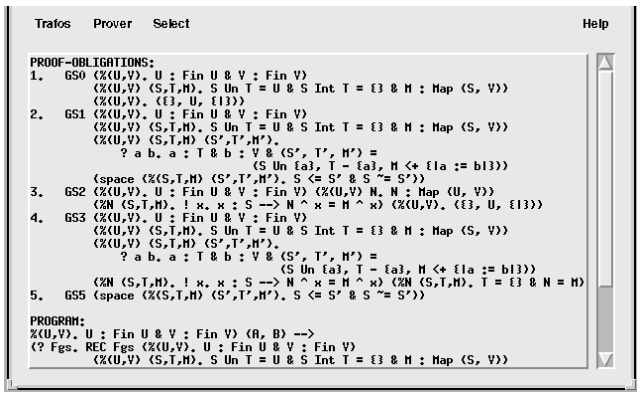


Figure 4.3.3: The resulting proof state

Now, we may use Isabelle to verify the proof obligations. At an arbitrary point of the development, we may decide to "freeze" the complete proof state i.e. convert it into an Isabelle theory containing the proof state as the implication: "if the (remaining) proof obligations hold, then the program is equivalent to the specification". Later, we can reload the frozen proof state and resume the development.

Although a small example, fin_maps demonstrates the virtues of top-down development via stepwise refinement in a transformational setting. Simply by indicating to the system which transformation shall be applied to a specification, YATS can check if the transformation is applicable, and systematically leads the user to the necessary design decisions and proof obligations.

5 Discussion

We have shown how to implement transformation systems in a systematic way that clearly separates the *soundness* issues of transformations, the *pragmatics* of their application, and their *presentation* to developers at the user interface.

For several reasons, representing the logical content of transformations by synthesis theorems highly increases the users' confidence in single transformation steps and thereby in the correctness of software they develop with the implemented system.

- Several attempts to construct a new transformation are usually needed until it is indeed expressed in a correct and useful form. Since *practically useful* transformations must capture large and complex design steps, they are difficult to conceive and implement by human developers. Here, a mechanical proof of the synthesis theorem may be useful to increase confidence in the soundness of the transformation. Moreover, a proof can be useful to *find* the final shape of a transformation.
- Separating transformations into logical core and tactical sugar clearly identifies the parts of the implementation that guarantee correctness of the resulting software. Program errors in the tactical sugar parts of the system or the user interface can in no way lead to logically incorrect results of transformations.
- It is often possible to formalise transformation concepts like "Divide-and-Conquer" in different synthesis theorems. With our approach, it is easy to relate, to specialise and to combine synthesis theorems to form new transformations.
- It is conceivable to extend our system dynamically by proving the synthesis-theorems which are sugared in a standardised way automatically.
- Different specification formalisms like CSP and Z have been represented in HOL. The representation of transformation rules as synthesis theorems provides a means to study rules in combined multi-formalism environments.

The tactical theorem prover is the core of our system. It is not only used to prove synthesis theorems but also, based on its meta-variables and deduction facilities, to implement transformation applications. We have shown that this can be done with little effort. Moreover, in [HSZ 95], we have demonstrated a systematic approach to build tactical sugar.

The user-interface description and command language Tcl/Tk has seen a tremendous success in the recent years. From our experience, the X-Window toolkit Tk seems to offer "the right abstraction" for building user interfaces. However, the design of our parameterized interface and our productivity to code it profited substantially from the embedding of Tk into SML with its typing and modularisation concepts.

5.1 Related Work

The transformational approach to program development has a long tradition. Starting from the Munich CIP project [Bau+85], many studies have stressed the importance of the approach. During the PROSPECTRA project [HK 93], a system has been developed that enabled the formalisation of transformation rules and their use during the software development process.

In KIDS [Smi 91], programs are developed by transforming *problem specifications* to programs. First, high-level transformations such as global search are used to come from the problem specification to a (inefficient) program. This program is then optimised by low-level program transformations like finite differencing or case distinction.

While the research in the context of KIDS has contributed much in the area of mathematically describing complex transformations and tactical sugar for their successful application, a shortcoming of the implemented system is that there is no easy way to convince oneself of the soundness of the implemented transformations. Our work focuses on this aspect and may thus be regarded complementary to the KIDS work.

Kreitz [Kre 93] gives mechanical proofs of global search theories in a constructive type theory, namely Nuprl [Con 86]. He aims at capturing even the pragmatics of transformation applications in a logical framework and attempts to extract synthesis tactics from the computational content of constructive proofs. In our approach, formal logic is used only to treat the soundness of transformations. This admits varying degrees of sophistication of tactical sugar: we can easily have different tactics for the same synthesis theorem. To achieve the same effect, Kreitz would have to provide different *proofs* of the theorem, each encoding a different approach to its application.

Basin's work [Bas 94] represents an approach to logic program synthesis also implemented in Isabelle. It is based on the Whelk logic that has been proposed as foundation. The rules of Whelk are derived in Isabelle. This work focuses on foundational issues rather than on a practical system implementation.

The recent formation of new workshops show a growing interest in the design and implementation of graphical user interfaces for both theorem provers and IFDSEs. A notable implementation is the TkHOLWorkbench currently developed in Cambridge [Sym 95], another one is [CO 95] for Isabelle. Although these interfaces currently are clearly superior to ours it is predominantly implemented in Tcl and not in ML. For this reason, we believe that our approach offers a higher potential of growth and reusability for similar systems.

5.2 Future Work

Our implementation to date is a prototype to illustrate the approach. To make it practically usable, two improvements have to be made: we need to extend the library of transformations and we need to incorporate a standard specification language which is used in practice.

Incorporating more standard transformations from the literature is easy. The standardised form of synthesis theorems even allows us to develop a set of "meta-transformations" — as ML functions — yielding transformations from synthesis theorems automatically. Meta-transformations might implement different ways to deal with application conditions and parameters. As we have mentioned in section 3.5, verificational approaches supplying the parameters directly and leaving application conditions as subgoals to verify are as well as possible as constructive approaches that — auto-

matically or interactively — synthesise parameters while proving application conditions. Different techniques to match an input pattern against a goal are possible.

As for the use of a standard specification language, research is going on to implement support for, e.g., Z in Isabelle. Proving suitable synthesis theorems in the logical representation of such a specification language makes our approach immediately applicable to that language. This work is partly an objective of the project UniForM [Kri+95].

Acknowledgement. We would like to thank Maritta Heisel for many discussions on synthesis theorems, and two anonymous referees for very extensive and constructive comments.

References

[And 86] Andrews, P.B., *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[Bas 94] Basin, D., Isawhelk: Whelk interpreted in Isabelle. Abstract accepted at the *11th International Conference on Logic Programming* (ICLP 94). Full version available via http://www.mpi-sb.mpg.de/guide/staff/basin/-pubs/iclp11.ps.Z.

[Bau+85] Bauer et al. (The CIP Language Group): *The Munich Project CIP. Volume I: The wide spectrum language CIP-L*. LNCS 183. 1985.

[BH 95] Bowen, J. P., Hinchey, M. J., Seven more Myths of Formal Methods: Dispelling Industrial Prejudices, in *FME'94: Industrial Benefit of Formal Methods*, proc. 2nd Int. Symposium of Formal Methods Europe, LNCS 873, Springer Verlag 1994, pp. 105-117.

[BM 93] Bird, R., de Moor, O.: Solving Optimisation Problems with Catamorphisms, in Proc. *Second Conference on the Mathematics of Program Construction*, LNCS 669, Springer Verlag 1993, pp. 49-56.

[Chu 40] Church, A., A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940, pp. 56-68.

[CO 95] Cant, A., Ozohls, M.A. XIsabelle: A graphical User Interface to the Isabelle Theorem Prover. ftp://ftp.cl.cam.ac.uk/ml/XIsabelle-2.0.tar.gz

[Con 86] Constable, R. S. et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[GL 93] Gersdorf, B., Liu, J. personnal communication.

[GM 93] Gordon, M.J.C., Melham, T.M.: *Introduction to HOL: a theorem proving environment for higher order logics*, Cambridge University Press, 1993.

[Gor 95] Gordon. M: Notes on PVS from a HOL perspective. Available via Internet http://www.cl.cam.ac.uk/users/mjcg/PVS.ps.gz, 1995.

[HK 93] Hoffmann, B., Krieg-Brückner, B. (eds.): *PROgram Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System.* LNCS 680, Springer-Verlag 1993.

[HMM 86] Harper, R., MacQueen, R., Milner, R: Standard ML. Technical Report ECS-LFCS-86-2. 1986.

[HSZ 95] Heisel, M., Santen, T., Zimmermann, D.: Tool Support for Formal Software Development: A Generic Architecture, in *Software Engineering — ESEC '95*, LNCS 989, Springer Verlag, 1995, pp. 272-293.

[Kre 93] Kreitz, C.: Meta-Synthesis. Deriving Programs that Develop Programs. Technische Hochschule Darmstadt, 1993.

[Kri+95] Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A.: Uniform Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995.

[KW 95] Kolyang, Wolff, B.: Development by Refinement Revisited: Lessons learnt from a case study. Proc. *Softwaretechnik '95*. Software-Technik Trends, Gesellschaft für Informatik, 1995, pp. 55-66 .

[OSR 93] Owre, S., Shankar, N., Rushby, J.M.: The PVS Specification Language (Beta Release). Comp. Sci. Lab., SRI International, Menlo Park, 1993.

[Ous 94] Ousterhout, J.K.: *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[Pau 94a] Paulson, L. C.: *Isabelle - A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994.

[Pau 94b] Paulson, L. C.: A fixedpoint approach to implementing (co)inductive definitions, in Alan Bundy (ed), *12th International Conference on Automated Deduction*, LNAI 814, 1994, Springer Verlag, pp. 148-161.

[PR 95] Pessaux, F., Rouaix, F.: The Caml/Tk interface, Projet Cristal, INRIA Rocquencourt, July 1995 ftp://ftp.inria.fr/lang/../INRIA/Projects/cristal/caml-light/camltk.dvi.tar.gz

[Smi 87] Smith, D. R.: Structure and Design of Global Search Algorithms, Technical Report Kes.U.87.12, Kestrel Institute, Palo Alto, 1987.

[Smi 90] Smith, D. R.: KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering* Special Issue on Formal Methods in Software Engineering, 16(9), 1990, pp. 1024–1043.

[SPW 95] Smith, D. R., Parra, E. A., Westfold, S. J.: Synthesis of High-Performance Transportation Schedulers, Technical Report, Kestrel Institute, Palo Alto, 1995.

[Sym 95] Syme, D.: A New Interface for HOL – Ideas, Issues and Implementation in *Higher Order Logic: Theorem Proving and its Applications*, LNCS 971, Springer Verlag 1995. pp 325-339.

[VTS 95] Vullinghs,T., Tuijnman, D., Schulte, W., Lightweight GUI's for functional programming. PLILP 95, Utrecht, The Netherlands, 1995.

TAS — A Generic Window Inference System

Christoph Lüth[1] and Burkhart Wolff[2]

[1] FB 3 — Mathematik und Informatik, Universität Bremen
cxl@informatik.uni-bremen.de
[2] Institut für Informatik, Albert-Ludwigs-Universität Freiburg
wolff@informatik.uni-freiburg.de

Abstract. This paper presents work on technology for transformational proof and program development, as used by window inference calculi and transformation systems. The calculi are characterised by a certain class of theorems in the underlying logic. Our transformation system TAS compiles these rules to concrete deduction support, complete with a graphical user interface with command-language-free user interaction by gestures like drag&drop and proof-by-pointing, and a development management for transformational proofs. It is *generic* in the sense that it is completely independent of the particular window inference or transformational calculus, and can be instantiated to many different ones; three such instantiations are presented in the paper.

1 Introduction

Tools supporting formal program development should present proofs and program developments in the form in which they are most easily understood by the user, and should not require the user to adapt to the particular form of presentation as implemented by the system. Here, a serious clash of cultures prevails which hampers the wider usage of formal methods: theorem provers employ presentations stemming from their roots in symbolic logic (e.g. Isabelle uses natural deduction), whereas engineers are more likely to be used to proofs by transformation as in calculus. As a way out of this dilemma, a number of systems have been developed to support *transformational development*. However, many of these systems such as CIP [3], KIDS [21] or PROSPECTRA [12] suffered from a lack of proof support and proven correctness. On the other hand, a variety of calculi have been developed which allow formal proof in a transformational way and are proven correct [8–10, 28, 11, 2], some even with a graphical user interface [14, 6]. However, what has been lacking is a systematic, generic and reusable way to obtain a user-friendly tool implementing transformational reasoning, with an open system architecture capable of coping with the fast changes in technology in user interfaces, theorem provers and formal methods. Reusability of components is crucial, since we hope that the considerable task of developing appropriate GUIs for formal method tools can be shared with other research groups.

In [15], we have proposed an open architecture to build graphical user interfaces for theorem provers in a functional language; here, we instantiate this

architecture with a generic transformation system which implements transformational calculi (geared towards refinement proofs) on top of an LCF-like prover. By *generic*, we mean that the system takes a high-level characterisation of a refinement calculus and returns a user-friendly, formally correct transformation or window inference system. The system can be used for various object logics and formal methods (a property for which Isabelle is particularly well suited as a basis). The instantiation of the system is very straightforward once the formal method (including the refinement relation) has been encoded. Various aspects of this overall task have been addressed before, such as logical engines, window-inference packages and prototypical GUIs. In contrast, TAS is an *integrated* solution, bringing existing approaches into one technical framework, and filling missing links like a generic pretty-printer producing markups in mathematical text.

This paper is structured as follows: in Sect. 2 we give an introduction to window inference, surveying previous work and presenting the basic concepts. We explain how the formulation of the basic concepts in terms of ML theorems leads to the implementation of TAS. We demonstrate the versatility of our approach in Sects. 3, 4 and 5 by showing examples of classical transformational program development, for process-oriented refinement proofs and for data-oriented refinement proofs. Sect. 6 finishes with conclusions and an outlook.

2 A Generic Scheme of Window Inference

Window inference [18], structured calculational proof [8, 1, 2] and transformational hierarchical reasoning [11] are closely related formalisations of proof by transformation. In this paper, we will use the format of [1], although we will refer to it as window inference.

2.1 An Introduction to Window Inference

As motivating example, consider the proof for $\vdash (A \wedge B \Rightarrow C) \Rightarrow (B \wedge A \Rightarrow C)$. In natural deduction, a proof would look like (in the notation of [27]; we assume that the reader is roughly familiar with derivations like this):

$$\dfrac{\dfrac{\dfrac{\dfrac{[B \wedge A]^1}{A}\wedge E \quad \dfrac{[B \wedge A]^1}{B}\wedge E}{A \wedge B}\wedge I \qquad [A \wedge B \Rightarrow C]^2}{C}\Rightarrow E}{\dfrac{B \wedge A \Rightarrow C}{(A \wedge B \Rightarrow C) \Rightarrow (B \wedge A \Rightarrow C)}\Rightarrow I_2}\Rightarrow I_1 \tag{1}$$

The following equivalent calculational proof is far more compact. We start with $B \wedge A \Rightarrow C$. In the first step, we open a *subwindow* on the sub-expression $B \wedge A$, denoted by the markers. We then transform the sub-window and obtain

the desired result for the whole expression:

$$
\begin{array}{ll}
 & \llcorner B \wedge A \lrcorner \Rightarrow C \\
\Leftarrow & \quad \{\text{focus on } B \wedge A\} \\
 & \quad \bullet\ B \wedge A \\
 & \quad = \{\wedge \text{ is commutative}\} \\
 & \quad\ \ A \wedge B \\
 & \cdot\ \ulcorner A \wedge B \urcorner \Rightarrow C
\end{array}
\tag{2}
$$

The proof profits from the fact that we can replace equivalent subexpressions. This is formalised by *window rules* [11]. In this case the rule has the form

$$
\frac{\Gamma \vdash A = B}{\Gamma \vdash E[A] \Rightarrow E[B]} \tag{3}
$$

where the second-order variable E stands for the unchanged *context*, while the subterm A (the *focus* of the transformation) is replaced by the transformation.

Comparing this proof with the natural deduction proof, we see that in the latter we have to decompose the context by applying one rule per operator, whereas the calculational proof employs second-order matching to achieve the same effect directly. Although in this format, which goes back to Dijkstra and Scholten [8], proofs tend to be shorter and more abstract, there are known counterexamples such as proof by contradiction.

In Grundy's work [11], window inference proofs are presented in terms of natural deduction proofs. By showing every natural deduction proof can be constructed using window inference rules, completeness of window inference for first-order logic is shown. This allows the implementation of window inference in a theorem prover. A similar technique underlies our implementation: the system constructs Isabelle proofs from window inference proofs.

As was shown in [11, 1], window inference proofs are not restricted to first-order logic or standard proof refinement, i.e. calculational proofs based on the implication and equality. It is natural to admit a family $\{R_i\}_{i \in I}$ of reflexive and transitive binary relations that enjoy a generalised form of monotonicity (in the form of (3) above).

Extending the framework of window inference in these directions allows to profit from its intuitive conciseness not only in high-school mathematics and traditional calculus, which deals with manipulating equations, but also in formal systems development, where the refinement of specifications is often the central notion. However, adequate user interface support is needed if we want to exploit this intuitive conciseness; the user interaction to set a focus on a subterm should be little more than marking the subterm with the mouse (point&click), otherwise the whole beneficial effect would be lost again.

2.2 The Concepts

Just as equality is at the heart of algebra, at the heart of window inference there is a family of binary preorders (reflexive and transitive relations) $\{\sqsubseteq_i\}_{i \in I}$. These

preorders are called the *refinement relations*. Practically relevant examples of refinement relations in formal system development are impliedness $S \Leftarrow P$ (used for algebraic model inclusion, see Sect. 3), process refinement $S \sqsubseteq_{FD} P$ (the process P is more defined and more deterministic than the process S, see Sect. 4), set inclusion (see Sect. 5), or arithmetic orderings for numerical approximations [29]. An example for an infinite family of refinement relations in HOL is the Scott-definedness ordering for higher-order function spaces (where the indexing set I is given by the types):

$$f \sqsubseteq_{(\alpha\to\beta)\times(\alpha\to\beta)\to Bool} g \equiv \forall x.\, f\, x \sqsubseteq_{\beta\times\beta\to Bool} g\, x \tag{4}$$

The refinement relations have to satisfy a number of properties, given as a number of theorems. Firstly, we require reflexivity and transitivity for all $i \in I$:

$$a \sqsubseteq_i a \qquad [\text{Refl}_i]$$
$$a \sqsubseteq_i b \wedge b \sqsubseteq_i c \Rightarrow a \sqsubseteq_i c \qquad [\text{Trans}_i]$$

The refinement relations can be ordered. We say $\sqsubseteq_i$ is *weaker* than $\sqsubseteq_j$ if $\sqsubseteq_i$ is a subset of $\sqsubseteq_j$, i.e. if $a \sqsubseteq_i b$ implies $a \sqsubseteq_j b$:

$$a \sqsubseteq_i b \Rightarrow a \sqsubseteq_j b \qquad [\text{Weak}_{i,j}]$$

The ordering is optional; in a given instantiation, the refinement relations may not be related at all. However, because of reflexivity, equality is weaker than any other relation, i.e. for all $i \in I$, the following is a derived theorem:[1]

$$a = b \Rightarrow a \sqsubseteq_i b \tag{5}$$

The main device of window inferencing are the window rules shown in the previous section:

$$(A \Rightarrow a \sqsubseteq_i b) \Rightarrow F\, a \sqsubseteq_j F\, b \qquad [\text{Mono}^F_{i,j}]$$

Here, F can either be a meta-variable[2], or a constant-head expression, i.e. a term of the form $\lambda y_1 \ldots y_m.c x_1 \ldots x_n$ with c a constant. Note how there are different refinement relations in the premise and conclusion of the rule. Using a family of rules instead of one monotonicity rule has two advantages: firstly, it allows us to handle, on a case by case basis, instantiations where the refinement relations are not congruences, and secondly, by allowing an additional assumption A in the monotonicity rules, we get more assumptions when refining inside a context. These *contextual assumptions* are crucial, many proofs depend on them.[3]

[1] In order to keep our transformation system independent of the object logic being used, we do not include any equality per default, as different object logics may have different equalities.

[2] In Isabelle, meta-variables are variables in the meta-logic, which are subject to unification. Users of other theorem provers can think of them just as variables.

[3] They already featured in the pioneering CIP-S system [3] in 1984.

Dependencies between refinement relations can be more complicated than the restricted form of weakening rules [$\text{Weak}_{i,j}$] above may be able to express; for example, (4) cannot be expressed by a weakening rule in either direction because of the outermost quantor on the right side. For this reason, there is a further need for *refinement conversions*, i.e. tactical procedures that attempt to rewrite one refinement proof goal into another.

To finish off the picture, we consider transformation rules. A transformation rule is given by a *logical core theorem* of the form

$$A \Rightarrow (I \sqsubseteq_j O) \tag{6}$$

where A is the *application condition*, I the *input pattern* and O the *output pattern*. In other words, transformation rules are theorems the conclusion of which is a refinement relation.

2.3 Parameters

The *parameters* for a transformation rule given by core theorem schema (6) are meta-variables occuring in the output pattern O but not in the input pattern I. After applying the transformation, a parameter occurs as a free meta-variable in the proof state. This is not always useful, hence parameters enjoy special support. In particular, in transformational program development (see Sect. 3) we have rather complex transformations with a lot of parameters and their instantiation is an important design decision. As a simple example, consider the theorem

$$t \Leftrightarrow \texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ t$$

which as a transformation rule from the left to the right introduces a case distinction on b. This is not very helpful unless we supply a concrete value for b which helps us to further develop t in the two different branches of the conditional expression under the respective assumption that b holds, or does not.

TAS supports parameters by when applying a transformation checking whether it contains parameters, and if so querying for their instantiation. It further allows parameter instantiations to be stored, edited and reused. This avoids having to retype instantiations, which can get quite lengthy, and makes TAS suitable for transformational program development as well as calculational proof.

2.4 The `Trafos` package

The `Trafos` package implements the basic window inferencing operations as Isabelle tactics, such as:

- opening and closing subwindows,
- applying transformations,
- searching for applicable transformations,
- and starting and concluding developments.

In general, our implementation follows Staples' approach [23], for example in the use of the transitivity rules to translate the forward chaining of transformation steps into backwards proofs on top of Isabelle's goal package, or the reflexivity rules to close subwindows or conclude developments.The distinctive features of our implementation are the subterm and search functionalities, so we concentrate on these in the following.

In order to open a subwindow or apply a transformation at a particular subterm, `Trafos` implements an abstract datatype `path` and operations `apply_trafo`, `open_sub` taking such a path (and a transformation) as arguments. To allow direct manipulation by point&click, we extend Isabelle's powerful syntax and pretty-printing machinery by *annotations* [15]. Annotations are markup sequences containing a textual representation of the path, which are attached to the terms. They do not print in the user interface, but instead generate a binding which invokes the respective operations with the corresponding path as argument. In general, users do not need to modify their theories to use the subterm selection facilities, they can be used as they are, including user-defined pretty-printing.[4]

The operations `apply_trafo` and `open_sub` analyse the context, and for each operation making up the context, the most specific $[\mathrm{Mono}_i^F]$ rule is selected, and a proof step is generated. In order to speed up this selection, the monotonicity rules are indexed by their head symbol, so we can discard rules which cannot possibly unify; still, the application of the selected rules may fail, so a tactic is constructed which tries to apply any combination of possibly fitting rules, starting with the most specific.

Further, for each refinement relation $\sqsubseteq_i$, we try to find a rule $[\mathrm{Mono}_{i,i}^F]$ where F is just a meta-variable and the condition A is void — this rule would state that $\sqsubseteq_i$ is a congruence. If we can find such a rule, we can use it to handle, in one step, large parts of the context consisting of operations for which no more specific rule can be found. If no such congruence rule can be found, we do not construct a step-by-step proof but instead use Isabelle's efficient rewriter, the simplifier, with the appropriate rules to break down larger contexts in one step.

As an example why the more specific rules are applied first, consider the expression $E = x + (\mathtt{if}\ x = 0\ \mathtt{then}\ u + x\ \mathtt{else}\ v + x)$. If we want to simplify $u + x$, then we can do so under the assumption that $x = 0$, and we have $x + 0 \Rightarrow u + x = u$ because of the theorem

$$(B \Rightarrow x = y) \Rightarrow (\mathtt{if}\ B\ \mathtt{then}\ x\ \mathtt{else}\ z = \mathtt{if}\ B\ \mathtt{then}\ y\ \mathtt{else}\ z) \qquad [\mathrm{Mono}_{=}^{\mathtt{If}}]$$

But if we had just used the congruence rule for equality $x = y \Rightarrow f\,x = f\,y$ we would have lost the contextual assumption $x = 0$ in the refinement of the if-branch of the conditional.

When looking for applicable transformations, performance becomes an issue, and there is an inherent trade-off between the speed and accuracy of the search. In principle, we have to go through all theorems in Isabelle's database and check

[4] Except if Isabelle's freely programmable so-called *print translations* are used (which is rarely the case). In this case, there are facilities to aid in programming markup-generation analogously to these print-translations.

whether they can be considered as transformation rule, and if so if the input pattern of the rule matches. Many theorems can be excluded straight away since their conclusion is not a refinement. For the rest, we can either superficially check whether they might fit, which is much faster but bears the risk of returning rules which actually do not fit, or we can construct and apply the relevant tactic. We let users decide (by setting a search option) whether they want fast or accurate search. Another speed-up heuristic is to be able to specify that rules are only collected from certain theories (called *active theories*). Finally, users can exclude expanding rules (where the left-hand side is only a variable), because most (but not all) of the time these are not really helpful. In this way, users can guide the search for applicable transformations by selecting appropriate heuristics.

When instantiating the functor `Trafos`, the preprocessing of the monotonicity rules as described above takes place (calculation of the simplifier sets, head constants etc.) Further, some consistency checks are carried out (e.g. that there are transitivity and reflexivity rules for all refinement relations).

2.5 Genericity by Functors

In Standard ML (SML), modules are called *structures*. *Signatures* are module types, describing the interface, and *functors* are parameterised modules, mapping structures to structures. Since in LCF provers theorems are elements of an abstract SML datatype, we can describe the properties of a window inference calculus as described in Sect. 2.2 above using SML's module language, and implement TAS a functor, taking a structure containing the necessary theorems, and returning a transformation or window inferencing system complete with graphical user interface built on top of this:

```
functor TAS(TrfThy: TRAFOTHY) = ...
```

The signature `TRAFOTHY` specifies a structure which contains all the theorems of Sect. 2.2. Abstracted a little (by omitting some parameters for special tactical support), it reads as follows:

```
signature TRAFOTHY =
    sig val topthy   : string
        val refl     : thm list
        val trans    : thm list
        val weak     : thm list
        val mono     : thm list
        val ref_conv : (string* (int-> tactic)) list
        ...
    end
```

To instantiate TAS, we need to provide a theory (named `topthy`) which encodes the formal method of our choice and where our refinement lives, theorems describing the transitivity, reflexivity and monotonicity of the refinement relation(s), and a list of refinement conversions, which consist of a name, and a tactic

when when applied to a particular subgoal converts the subgoal into another refinement relation.

When applying this functor by supplying appropriate arguments, we obtain a structure which implements a window inferencing system, complete with a graphical user interface. The graphical user interface abstracts from the command line interface of most LCF provers (where functions and values are referred to by names) by implementing a *notepad*, on which objects (theorems, theories, etc.) can be manipulated by drag&drop. It provides a *construction area* where the current on-going proof is displayed, and which has a *focus* to open subwindows, apply transformations to subterms or search the theorem database for applicable transformations. We can navigate the *history* (going backwards and forwards), and display the history concisely, or in detail through an active display, which allows us to show and hide subdevelopments. Further, the user interface provides an active *object management* (keeping track of changes to external objects like theories), and a *session management* which allows to save the system state and return to it later. All of these features are available for *any* instance of TAS, and require no additional implementation; and this is what we mean by calling TAS *generic*.

The implementation of `TAS` consists of two components: a kernel transformation system, which is the package `Trafos` as described in Sect. 2.4, and a graphical user interface on top of this. We can write this simplified as

```
functor TAS(TrfThy : TRAFOTHY) = GenGUI(Trafos(TrfThy : TRAFOTHY))
```

The graphical user interface is implemented by the functor `GenGUI`, and is independent of `Trafos` and Isabelle. For a detailed description, we refer to [15], but in a nutshell, the graphical user interface is implemented entirely in SML, using a typed functional encapsulation of Tcl/Tk called `sml_tk`. Most of the GUI features mentioned above (such as the notepad, and the history, object and session management) are implemented at this more general level.

The division of the implementation into a kernel system and a generic graphical user interface has two major advantages: firstly, the GUI is reusable for similar applications (for example, we have used it to implement a GUI `IsaWin` to Isabelle itself); and secondly, it allows us to run the transformation system without the graphical user interface, e.g. as a scripting engine to check proofs.

3 Design Transformations in Classical Program Transformation

In the design of algorithms, certain schemata can be identified [7]. When such a schema is formalised as a theorem in the form of (6), we call the resulting transformation rule a *design transformation*. Examples include *divide and conquer* [20], *global search* [22] or *branch and bound*. Recall from Sect. 2.2 that transformation rules are represented by a logical core theorem with an input pattern and an output pattern. Characteristically, design transformations have as input pattern a *specification*, and as output pattern a *program*.

Here, a specification is given by a pre- and a postcondition, i.e. a function $f : X \to Y$ is specified by an implication $Pre(x) \longrightarrow Post(x, f(x))$, where $Pre : X \to Bool, Post : X \times Y \to Bool$. A program is given by a recursive scheme, such as well-founded recursion; the proof of the logical core theorem must accordingly be based on the corresponding induction principles, i.e. here well-founded induction. Thus, a function $f : X \to Y$ can be given as

$$\texttt{let fun } f(x) = E \texttt{ in } f \texttt{ end measure } < \tag{7}$$

where E is an expression of type Y, possibly containing f, and $< \subseteq X \times X$ is a well-founded relation, the *measure*, which must decrease with every recursive call of f. The notational proximity of (7) to SML is intended: (7) can be considered as a functional program.

As refinement relation, we will use model-inclusion — when refining a specification of some function f, the set of possible interpretations for f is reduced. The logical equivalent of this kind of refinement is the implication, which leads to the following definition:

$$\sqsubseteq : Bool \times Bool \to Bool \qquad P \sqsubseteq Q \stackrel{def}{=} Q \longrightarrow P$$

Based on this definition, we easily prove the theorems `ref_trans` and `ref_refl` (transitivity and reflexivity of $\sqsubseteq$). We can also prove that $\sqsubseteq$ is monotone for all boolean operators, e.g.

$$s \sqsubseteq t \Rightarrow s \wedge u \sqsubseteq t \wedge u \qquad \texttt{ref\_conj1}$$

Most importantly, we can show that

$$(B \Rightarrow s \sqsubseteq t) \Rightarrow \texttt{if } B \texttt{ then } s \texttt{ else } u \sqsubseteq \texttt{if } B \texttt{ then } t \texttt{ else } u \qquad \texttt{ref\_if}$$
$$(\neg B \Rightarrow u \sqsubseteq v) \Rightarrow \texttt{if } B \texttt{ then } s \texttt{ else } u \sqsubseteq \texttt{if } B \texttt{ then } s \texttt{ else } v \qquad \texttt{ref\_then}$$

which provides the contextual assumptions mentioned above. When instantiating the functor, we also have to specify equality as a refinement relation. Since we can reuse the relevant definitions for all theories based on HOL, they have been put in a separate functor `functor HolEqTrfThy(TrfThy : TRAFOTHY) : TRAFOTHY` In particular, this functor proves the weakening theorems (5) for all refinement relations, and appends them to the list `weak`. Thus, the full functor instantiation reads

```
structure HolRefThy =
  struct val name  = "HolRef"
         val trans = [ref_trans]
         val refl  = [ref_refl]
         val weak  = []
         val mono  = [ref_if, ref_else, ref_conj1, ref_conj2,
                      ref_disj1, ref_disj2, ...]
         val ref_conv = []
         ...
  end
structure TAS = TAS(HolEqTrfThy(HolRefThy))
```

The divide and conquer design transformation [20] implements a program $f : X \to Y$ by splitting X into two parts: the termination part of f, which can be directly embedded into the codomain Y of f, and the rest, where the values are divided into smaller parts, processed recursively, and reassembled. The core theorem for divide and conquer based on model-inclusion refinement and well-founded recursion reads:[5]

$$\begin{array}{l} A \longrightarrow (Pre(x) \longrightarrow Post(x, f(x)) \\ \quad \sqsubseteq \\ \quad Pre(x) \longrightarrow f = \texttt{let fun } F(x) = \texttt{if } isPrim(x) \texttt{ then } Dir(x) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else } Com(\langle G, F\rangle(Decom(x))) \\ \qquad\qquad\qquad\quad \texttt{in } F \texttt{ end measure } <) \end{array} \tag{8}$$

As explained above, the parameters of the transformation are the meta-variables appearing in the output pattern but not in the input pattern of the logical core theorem (8). Here, these are

- the termination criterion $isPrim : X \to Bool$;
- the embedding of terminal values $Dir : X \to Y$;
- the decomposition function of input values $Decom : X \to Z \times X$;
- a function $G : Z \to U$ for those values which are not calculated by recursive calls of F;
- the composition function $Com : U \times Y \to Y$ that joins the subsolutions given by G and recursive calls of F;
- and the measure $<$ assuring termination.

We will now apply this transformation to synthesise a sorting algorithm in the theory of lists. We start with the usual specification of *sort*, as shown on the left of Fig. 1. We can see the notepad, on which the transformation object `Divide & Conquer` is represented by an icon. The workspace shows the current state of the already started development. The highlighting indicates the focus set by the user. Now we drag the transformation onto the focus; TAS interprets this gesture as application of the transformation at the focus. In this case, TAS infers that there are parameters to be provided by the user, who is thus guided to the necessary design decisions. The parameter instantiations are fairly simple: the termination condition is the empty list, which is sorted (hence *Dir* is the identity). The decomposition function splits off the head and the tail; the tail is sorted recursively, and the head is inserted into the sorted list (hence, G is the identity). Finally, the measure relates non-empty lists to their tails (since the recursive call always passes the tail of the argument; a relation easily proven to be well-founded).

This transformation step readily produces the desired program (right of Fig. 1). However, this step is only valid if the application conditions of the transformation hold. When applying a transformation, these conditions turn into proof obligations underlying a special bookkeeping. The proof obligations

[5] $\langle f, g\rangle$ is the pairing of functions defined as $\langle f, g\rangle(x, y) \stackrel{def}{=} (f(x), f(y))$.

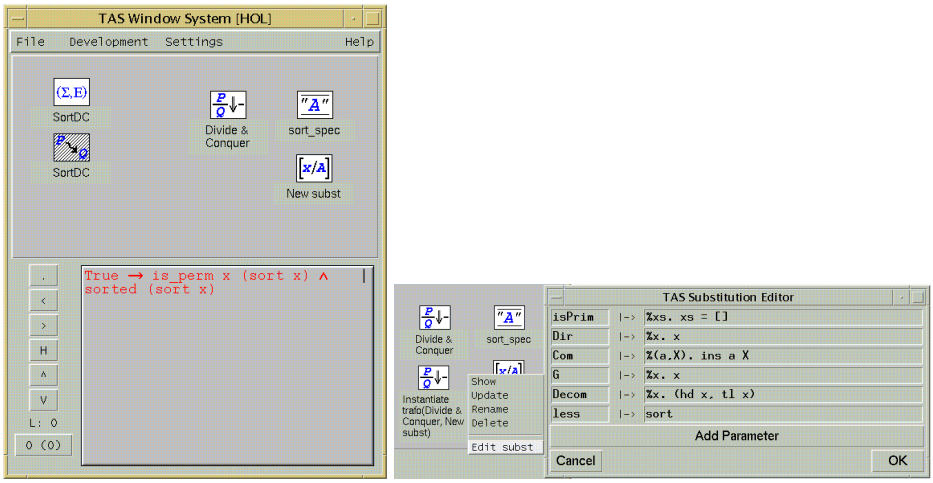


Fig. 1. TAS and its graphical user interface. To the left, the initial stage of the development, and the parameters supplied for the transformation; to the right, the development after applying the divide and conquer transformation. On the top of the window, we can see the notepad with the theory `SortDC`, the transformation `Divide&Conquer`, the specification `sort_spec`, the ongoing development (shaded) and the parameter instantiation `divconq_inst`.

can be proven with a number of proof procedures. Typically, these include automatic proof via Isabelle's simplifier or classical reasoner and interactive proof via IsaWin. Depending on the particular logic, further proof procedures may be at our disposal, such as specialised tactics or model-checkers integrated into Isabelle.

Another well-known scheme in algorithm design is *global search* which has been investigated formally in [22]. It represents another powerful design transformation which has already been formalised in an earlier version of TAS [13].

4 Process Modelling with CSP

This section shows how to instantiate TAS for refinement with CSP [19], and will briefly present an example how the resulting system can be used. CSP is a language designed to describe systems of interacting components. It is supported by an underlying theory for reasoning about their equivalences, and in particular their refinements. In this section, we use the embedding HOL-CSP [26] of CSP into Isabelle/HOL. Even though shortage of space precludes us the set out the basics of CSP here, a detailed understanding of CSP is not required in the following; suffice it to say that CSP is a language to model distributed programs as communicating processes.

CSP is interesting in this context because it has three refinement relations, namely trace refinement, failures refinement and failures-divergence refinement.

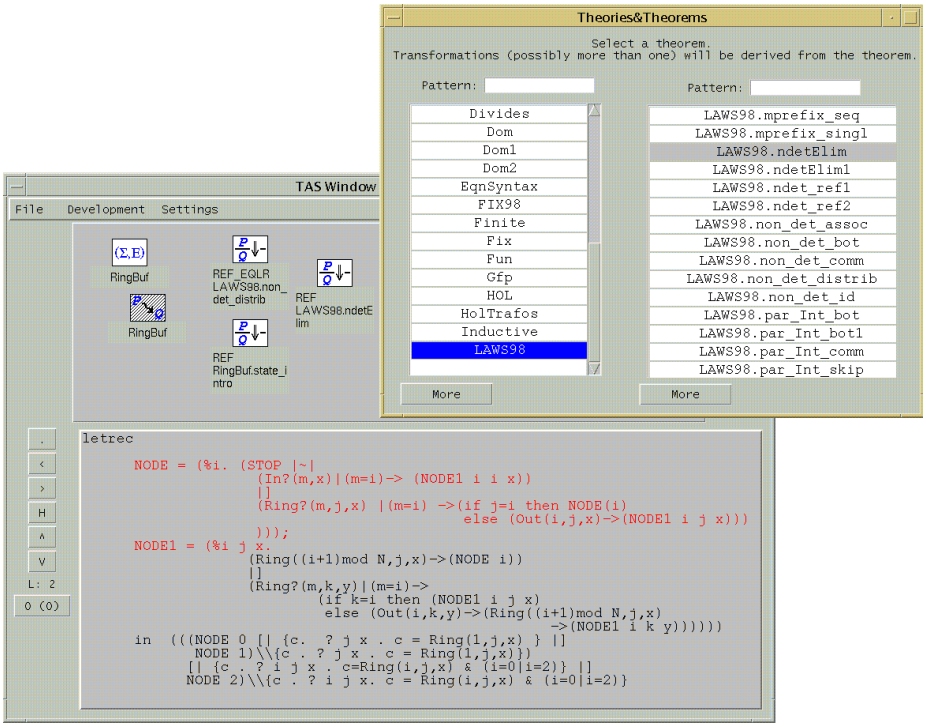


Fig. 2. TAS in the CSP instance. On the right, the construction history is shown. The development proceeded by subdevelopments on `COPY1` and `COPY2`, which can be shown and hidden by clicking on `[Subdevelopment]`. Similarly, proof obligations can be shown and hidden. In the lower part of the main window, the focus is set on a subterm, and all applicable transformations are shown. By clicking on the name of the transformations, their structure can be displayed (not shown).

Here, we only use the third, since it is the one most commonly used when developing systems from specifications, but e.g. trace refinement can be relevant to show security properties.

Recall from Sect. 2.5 that to instantiate TAS we need a theory encoding our formal method, and theorems describing the refinement relation. The relevant theory is called `CspTrafos`, which contains the core theorems of some (simple) transformations built on top of `Csp`, the encoding of CSP into Isabelle/HOL.

For brevity, we only describe instantiation with failure-divergence refinement; the other two refinements would be similar. The theorems stating transitivity and reflexivity of failure-divergence refinement are called `ref_ord_trans` and `ref_ord_refl`, respectively. For monotonicity, we have a family of theorems describing monotonicity of the operators of CSP over this relation, but since the relation is monotone only with respect to the CSP relations it is not a proper congruence. This gives us the following functor instantiation:

```
structure CspRefThy = struct
  val name  = "CspTrafos"
  val trans = [ref_ord_trans]
  val refl  = [ref_ord_refl]
  val mono  = [mono_mprefix_ref,mono_prefix_ref,mono_ndet_ref,
               mono_det_ref,mono_Ren_ref,mono_hide_set_ref,
               mono_PaI_ref,mono_Inter_ref]
  val weak  = []
  val ref_conv = []
  ...
  end
structure TAS = TAS(HolEqTrfThy(CspRefThy))
```

Fig. 2 shows the resulting, instantiated system in use. We can see an ongoing development on the left, and the opened construction history showing the development up to this point on the left. As we see, the development started with two processes in parallel; we focussed on both of these in turn to develop them separately, and afterwards rearranged the resulting process, using algebraic laws of CSP such as **sync_interl_dist** which states the distributivity of synchronisation over interleaving under some conditions. The development does not use powerful design transformations as in Sect. 3, but just employs a couple of the algebraic laws of CSP, showing how we can effectively use previously proven theorems for transformational development. Finding design transformations like divide and conquer for CSP is still an open research problem.

If we restrict ourselves to finite state processes (by requiring that the channels only carry finite messages), then we can even check the development above with the CSP model checker FDR [19], connected to Isabelle as a so-called *oracle* (a trusted external prover). This speeds up development at the cost of generality and can e.g. be used for rapid prototyping.

5 Data Refinement in the Refinement Calculus

In this section, we will emphasise a particular aspect of the genericity of TAS and demonstrate its potential for reuse of given logical embeddings. As we mentioned, TAS is generic with respect to the underlying refinement calculus, which in particular means that it is generic with respect to the underlying object logic. In the previous examples, we used higher-order logic (as encoded in Isabelle/HOL); in this example, we will use Zermelo-Fränkel set theory (as encoded in Isabelle/ZF). On top of Isabelle/ZF, Mark Staples has built a substantial theory for imperative program refinement and data refinement [24, 25] following the lines of Back's Refinement Calculus RC [2].

RC is based on a weakest precondition semantics, where predicates and predicate transformers are represented as sets of states and functions taking sets of states to sets of states respectively. The distinctive feature of Staples' work over previous implementations of refinement calculi is the use of sets in the sense of ZF based on an open type universe. This allows derivations where the types of

program variables are unknown at the beginning, and become more and more concrete after a sequence of development steps.

In order to give an idea of Staples' formalisation, we very briefly review some of the definitions of Back's core language in his presentation:[6]

$$\begin{aligned}
\mathtt{Skip}_{\mathtt{A}} &\stackrel{def}{=} \lambda q : \mathbb{P}(\mathtt{A}).q \\
a \ ; \ b &\stackrel{def}{=} \lambda q : dom(b).a \ `\ b \ `\ q \\
\mathtt{if}\ g\ \mathtt{then}\ a\ \mathtt{else}\ b\ \mathtt{fi} &\stackrel{def}{=} \lambda q : dom(a) \cup dom(b). \\
&\qquad (g \cap a \ `\ q) \cup ((\bigcup(dom(a) \cup dom(b)) - g) \cap b \ `\ q) \\
\mathtt{while}\ g\ \mathtt{do}\ c\ \mathtt{od} &\stackrel{def}{=} \lambda q : \mathbb{P}(A).\, lfp_A\, N.(g \cap c \ `\ N) \cup ((A - g) \cap q) \\
&\ldots
\end{aligned}$$

This theory could be used for an instantiation of TAS, called TAS/RC. The instantiation follows essentially the lines discussed in the previous sections; with respect to the syntactic presentation, the configuration for the pretty-printing engine had to provide special support for 5 print-translations comprising 100 lines of code, and a particular set-up for the tactics providing reasoning over well-typedness, regularity and monotonicity. (We omit the details here for space reasons). As a result, a larger case study in [24] for the development of an BDD-related algorithm as a data-refinement from truth tables to decision trees can be represented *inside* TAS.

6 Conclusions and Outlook

This paper has presented the transformation system TAS. TAS is generic in the sense that it takes a set of theorems, describing a refinement relation, and turns them into a window inference or transformation system, complete with an easy-to-use, graphical user interface. This genericity means that the system can be instantiated both to a transformation system for transformational program development in the vein of traditional transformation systems such as CIP, KIDS or PROSPECTRA, or as system for window inference. We have demonstrated this versatility by showing instantiations from the provenance of each the two areas just mentioned, complemented with an instantiation from a different area, namely reasoning about processes using CSP.

The effort required for the actual instantiation of TAS is very small indeed, since merely the values for the parameters of the functor need to be provided. (Only rarely will tactical programming be needed, such as mentioned in Sect. 5, and even then it only amounts to a few lines of code.) It takes far more effort to set up the logical encoding of the formal method, in particular if one does so conservatively.

TAS' graphical user interface complements the intuitiveness of transformational calculi with a command-language-free user interface based on gestures

[6] Note that the backquote operator ` is infix function application in Isabelle/ZF.

such as drag&drop and proof-by-pointing. It further provides technical infrastructure such as development management (replay, reuse, history navigation), object management and session management.

TAS is implemented on top of the prover Isabelle, such that the consistency of the underlying logics and its rules can be ensured by the LCF-style architecture of Isabelle and well-known embedding techniques. It benefits further from the LCF architecture, because we can use SML's structuring mechanisms (such as functors) to implement reusable, generic proof components across a wide variety of logics.

Internally, we spent much effort to organise TAS componentwise, easing the reuse of as much code as possible for completely different logical environments. The GUI and large parts of TAS (except the package `Trafos`) are designed to work with a different SML-based prover, and are readily available for other research groups to provide GUI support for similar applications. On the other hand, the logical embeddings (such as HOL-CSP) which form the basis of the transformation calculi do not depend on TAS either. This allowed the easy integration of Staples' encoding of the refinement calculus into our system, as presented in Sect. 5.

6.1 Discussion and Related Work

This work attempts to synthesise previous work on transformational program development [3, 21, 12] which developed a huge body of formalised developments and design schemes, but suffered from ad-hoc, inflexible calculi, correctness problems and lack of proof support, with the work on window inferencing [18, 11] and structured calculational proof [2, 1], which provides proven correctness by LCF design and proof support from HOL or Isabelle.

PRT [6] is a program refinement tool (using window inference) which is built on top of the Ergo theorem prover. It offers an interface based on Emacs, which allows development management and search functionalities. However, the Tk-WinHOL system [14] comes closest to our own system conception: it is based on Tcl/Tk (making it platform independent), and offers focusing with a mouse, drag&drop in transformational goals, and a formally proven sound calculus implemented by derived rules in HOL. On the technical side it uses Tcl directly instead of an encapsulation (which in our estimate will make it much harder to maintain). On the logical side, it is also generic in the sense that it can be used with different refinement relations, but requires more work to be adapted to a new refinement relation; for example, users need to provide a pretty-printer which generates the correct mark-up code to be able to click on subterms. In contrast, TAS extends Isabelle's infrastructure (like the pretty-printer) into the graphical user interface, leaving the user with less work when instantiating the system.

The essential difference between window inferencing and structured calculational proof [1] is that the latter can live with more than one transformational goal. This difference is not that crucial for TAS since it can represent more

than one transformational development on the notepad and is customisable for appropriate interaction between them via drag&drop operations.

Another possible generalisation would be to drop the requirement that all refinement relations be reflexive. However, this would complicate the tactical programming considerably without offering us perceivable benefit at the moment, so we have decided against it.

6.2 Future Work

Future work can be found in several directions. Firstly, the user interaction can still be improved in a variety of ways. Although in the present system, the user can ask for transformations which are applicable, this can considerably be improved by a best-fit strategy and, for example, stronger matching algorithms like AC-matching. The problem here is to help the user to find the few interesting transformations in the multitude of uninteresting (trivial, misleading) ones. Supporting design decisions at the highest possible user-oriented level must still count as an open problem, in particular in a generic setting.

Secondly, the interface to the outside world can be improved. Ideally, the system should interface to a variety of externally available proof formats, and export web-browsable proof scripts.

A rather more ambitious research goal is the reuse and abstraction of transformational developments. A first step in this direction would be to allow to cut&paste manipulation of the history of a proof.

Thirdly, going beyond classical hierarchical transformational proofs the concept of *indexed window inferencing* [29] appears highly interesting. The overall idea is to add an additional parameter to the refinement relation that allows to calculate the concrete refinement relation on the fly during transformational deduction. Besides the obvious advantage of relaxing the requirements to refinement relations to irreflexive ones (already pointed out in [23]), indexed window inferencing can also be used for a very natural representation of operational semantics rules. Thus, the system could immediately be used as an animator for, say, CSP, given the operational semantics rules for this language.

Finally, we would like to see more instances for TAS. Transformational development and proof in the specification languages Z and CASL should not be too hard, since for both embeddings into Isabelle are available [13, 16]. The main step here is to formalise appropriate notions of refinement. A rather simple different instantiation is obtained by turning the refinement relation around. This amounts to *abstracting* a concrete program to a specification describing aspects of its behaviour, which can then be validated by a model-checker. For example, deadlock checks using CSP and FDR have been carried out in this manner, where the abstraction has been done manually[4, 5, 17]. Thus we believe that TAS represents an important step towards our ultimate goal of a transformation system which is similarly flexible with respect to underlying specification languages and refinement calculi as Isabelle is for conventional logical calculi.

Acknowledgements We would like to thank Mark Staples and Jim Grundy for providing us with the sources for their implementations of window inference and the refinement calculus respectively, and the anonymous referees for threir constructive criticism. Ralph Back pointed out several weaknesses of a previous version of TAS and made suggestions for improvements.

References

1. R. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9:467–483, 1997.
2. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer Verlag, 1998.
3. F. L. Bauer. *The Munich Project CIP. The Wide Spectrum Language CIP-L.* Number 183 in LNCS. Springer Verlag, 1985.
4. B. Buth, J. Peleska, and H. Shi. Combining methods for the deadlock analysis of a fault-tolerant system. In *Algebraic Methodology and Software Technology AMAST'97*, number 1349 in LNCS, pages 60–75. Springer Verlag, 1997.
5. B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *Algebraic Methodology and Software Technology AMAST'98*, number 1548 in LNCS, pages 124–139. Springer Verlag, 1999.
6. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A Program Refinement Tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and New York: McGraw-Hill, 1989.
8. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.
9. D. Gries. *A Science of Programming*. Springer Verlag, 1981.
10. D. Gries. Teaching calculation and discrimination: A more effecticulum. *Communications of the ACM*, 34:45–54, 1991.
11. J. Grundy. Transformational hierarchical reasoning. *Computer Journal*, 39:291–302, 1996.
12. B. Hoffmann and B. Krieg-Brückner. *PROSPECTRA: Program Development by Specification and Transformation*. Number 690 in LNCS. Springer Verlag, 1993.
13. Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementations of transformation systems. In M. C. Gaudel and J. Woodcock, editors, *Formal Methods Europe FME'96*, number 1051 in LNCS, pages 629– 648. Springer Verlag, 1996.
14. T. Långbacka, R. Rukšėna, and J. von Wright. TkWinHOL: A tool for doing window inferencing in HOL. In *Proc.* 8^{th} *International Workshop on Higher Order Logic Theorem Proving and Its Applications*, number 971 in LNCS, pages 245–260. Springer Verlag, 1995.
15. C. Lüth and B. Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, 9(2):167– 189, March 1999.
16. T. Mossakowski, Kolyang, and B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In *Recent trends in algebraic development techniques. Proc* 13^{th} *International Workshop*, number 1376 in LNCS, pages 333– 348. Springer Verlag, 1998.
17. R. S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University, 1999.

18. P. J. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal for Logic and Computation*, 14(1):43–52, 1993.
19. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.
20. D. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.
21. D. R. Smith. KIDS — a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024– 1043, 1991.
22. D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305– 321, 1990.
23. M. Staples. Window inference in Isabelle. In *Proc. Isabelle Users Workshop.* University of Cambridge Computer Laboratory, 1995.
24. M. Staples. *A Mechanised Theory of Refinement.* PhD thesis, Computer Laboratory, University of Cambridge, 1998.
25. M. Staples. Representing wp semantics in isabelle/zf. In G. Dowek, C. Paulin, and Y. Bertot, editors, *TPHOLs: The 12th International Conference on Theorem Proving in Higher-Order Logics*, number 1690 in lncs. springer, 1999.
26. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe FME '97*, number 1313 in LNCS, pages 318–337. Springer Verlag, 1997.
27. D. van Dalen. *Logic and Structure.* Springer Verlag, 1994.
28. A. J. M. van Gasteren. On the shape of mathematical arguments. In *Advances in Software Engineering and Knowledge Engineering*, number 445 in LNCS, pages 1–39. Springer Verlag, 1990.
29. J. von Wright. Extending window inference. In *Proc. TPHOLs '98*, number 1497 in LNCS, pages 17–32. Springer Verlag, 1998.

Using Theory Morphisms for Implementing Formal Methods Tools

Achim D. Brucker und Burkhart Wolff

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 52, D-79110 Freiburg, Germany
{brucker,wolff}@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~{brucker,wolff}

Abstract Tools for a specification language can be implemented *directly* (by building a special purpose theorem prover) or *by a conservative embedding* into a typed meta-logic, which allows their safe and logically consistent implementation and the reuse of existing theorem prover engines. For being useful, the conservative extension approach must provide derivations for several thousand "folklore" theorems.
In this paper, we present an approach for deriving the mass of these theorems mechanically from an existing library of the meta-logic. The approach presupposes a structured *theory morphism* mapping library datatypes and library functions to new functions of the specification language while uniformly modifying some semantic properties; for example, new functions may have a different treatment of undefinedness compared to old ones.

Keywords: Formal Methods, Formal Semantics, Shallow Embeddings, Theorem Proving, OCL

1 Introduction

In contrast to a programming language, which defines computations, a *specification language* defines *properties* of computations, usually by extending a programming language with additional constructs such as quantifiers or universally quantified variables. Among the plethora of specification languages that has been developed, we will refer here only to examples such as Hoare-Logics [1, 2], Z [3, 4] or its semantic sister Higher-order Logics (HOL) [5], which has been advertised as "functional language with quantifiers" recently [6].

For the formal *analysis* of specification languages, their representation, i.e. their *embedding*, within a logical framework based on typed λ-calculi such as NuPRL [7], Coq [8] or Isabelle [9, 10] is a widely accepted technique that has been applied in many studies in recent years. With respect to *tools* implementing specification languages, the situation is not so clear-cut: while *direct* implementations in a programming environment are predominant [11, 12, 13], which result in special logic, special purpose theorem provers sometimes based on ad-hoc deduction technology, only a few tools are based on embeddings [14, 15, 16].

There are two main advantages of the embedding approach: Beside the reuse of existing theorem prover engines, building such tools based on a *conservative* embedding into a logical framework also guarantees the safety and relative logical consistency of the tool. Unfortunately, in order to be practically useful *and* consistency-aware, the conservative embedding approach must provide derivations for several thousand "folklore theorems" (such as the associativity of the concatenation on lists or the commutativity of the union on sets) of the underlying logics or the basic datatypes of a specification language.

Based on the observation that in many language embeddings the bulk of function definitions follows a common scheme, our contribution in this paper consists of a method to structure these definitions into a modular theory morphism and a technique that exploits this structure and attempts to automatically derive "folklore theorems" from their counterparts in the meta-logic. Thus, upgraded libraries of the meta-logic can lead automatically to new theorems in the object logic since generic tactical support can "transform" theorems over functions of the meta-level into theorems at the object level. To say it loud and clear: we do not expect that *all* functions of a language semantics will be amenable to our approach; for the 10 percent that are core language constructs, we expect more or less standard verification work for properties of the language. But for the 90 percent that are library functions, our approach may significantly facilitate the embedding approach and lead to more portability.

This work was partly motivated by the development of *HOL-OCL* [17, 18] a conservative embedding of the Object Constraint Language (OCL) [19, 20, 21] into HOL. OCL is a textual extension of the object-oriented Unified Modeling Language (UML) [22] which is widely used within the object oriented software development process. In principle, OCL is a subtyped, three-valued Kleene-Logic with equality that allows for specifying constraints on graphs of object instances whose structure is described by UML class diagrams.

This paper proceeds as follows: after a presentation of the foundation of this work, we propose a structuring of the theory morphism into layers and present for each layer some typical combinators that capture the essence of semantic transformation from a meta-logical function to an object-logical one. We discuss the theory of these combinators and conceptually describe the tactics that perform the generation of generic theorems and the transformation of meta-level "folklore theorems" to their object-logical counterparts by means of a conservative theory morphism.

2 Foundations

In the following section, we will introduce a formal framework in order to define the core notion of "conservative theory morphism" which leads to the key observations and their practical consequences for the construction of language embeddings. The purpose of these abstract definitions is to demonstrate that our approach is in fact fairly general and applies to a wide range of proof systems based on higher-order typed calculi. In the subsequent sections, we present a

comparison of embedding techniques and introduce the underlying terminology of our approach. Finally, we outline the context of our running example.

2.1 Formal Preliminaries: The Generic Framework

In this section, we will introduce a formal framework in order to define the core notion of "conservative theory morphism" which leads to the key observations and their practical observations for the construction of shallow embeddings. The terminology used here follows the framework of *institutions* [23]. Throughout this paper, however, it is sufficient to base our notions on simple set-theoretic concepts instead of full-blown category theory. The concept of signature is inspired by [24], but can be expressed in other typed λ-calculi too.

First we introduce the notion of *sorts*, *types* and *terms*; we assume a set ρ of sorts and a set χ of *type constructors*, e.g. $bool, \_ \to \_, list, \_\ set$. We assume a *type arity* ar, i.e. a finite mapping from type constructors to non-empty lists of sorts $ar : \chi \to_{\text{fin}} list_{\geq 1}(\rho)$. We define a set of types $\tau ::= \alpha \mid \chi(\tau, \ldots, \tau)$ based on the set of *polymorphic types* α. Further, we assume with $T(c, x)$ the *set of inductively defined terms* over constants c and variables x. For instance, for Isabelle-like systems, this set is defined as:

$$T(c, x) ::= c \mid x \mid T(c, x)T(c, x) \mid \lambda x.T(c, x) \ ,$$

while

$$\begin{array}{rll} \chi = \{\_ \to \_, bool\} & & \text{(type constructors)} \\ \rho = \{term\} & & \text{(set of sorts)} \\ ar = \{(bool \mapsto [term]), (\_ \to \_ \mapsto [term, term, term])\}. & & \text{(type arity)} \end{array}$$

A *signature* is a quadruple $\Sigma = (\rho, \chi, ar, c \to_{\text{fin}} \tau)$ and analogously the quadruple $\Gamma = (\rho, \chi, ar, x \to_{\text{fin}} \tau)$ is called an *environment*.

The following assumption incorporates a type inference and a notion of well-typed term: we assume a subset of terms called *typed terms* (written $T_{\Sigma,\Gamma}(c, x)$) and a subset *typed formulae* (written $F_{\Sigma,\Gamma}(c, x)$); we require that in these notions, ar, ρ and χ agree in Σ and Γ. For example, a *type inference system* for order-sorted polymorphic terms, can be found in [24]. Formulae, for example, can be typed terms of type *bool*.

We call $S = (\Sigma, A)$ with the axioms $A \subseteq F_{\Sigma,\Gamma}(c, x)$ a *specification*. The following assumption incorporates an inference system: with a *theory* $Th(S) \subseteq F_{\Sigma,\Gamma}(c, x)$ we denote the set of formulae derivable from A; in particular, we require $A \in Th(S)$ and Th to be monotonous in the axioms, i.e. $S \subseteq S' \implies Th(S) \subseteq Th(S')$ (we also use $S \subseteq S'$ for the extension of subsets on tuples for component-wise set inclusion).

A *signature morphism* is a mapping $\Sigma \to \Sigma$ which can be naturally extended to a *specification morphism* and a *theory morphism*.

The following specification extensions $S \subseteq S'$, called *conservative* specification extensions (see [5]), are of particular interest for this paper:

1. *type synonyms*,
2. *constant definitions*, and
3. *type definitions*.

A type synonym introduces a type abbreviation and is denoted as:

$$S' = S \uplus [\textbf{types}\ t(\alpha_1, \ldots, \alpha_n) = T(\alpha_1, \ldots, \alpha_n, t')] \ .$$

It is purely syntactical (i.e. it we will be used for abbreviations in type annotations only) such that the extension is defined by $S' = S$.

A constant definition is denoted as:

$$S' = S \uplus [\textbf{constdefs}\ \text{“}c = E\text{”}] \ .$$

A constant definition is *conservative*, if the following syntactic conditions hold: $c \notin \mathrm{dom}(\Sigma)$, E is closed and does not contain c, and no sub-term of E has a type containing a type variable, that is not contained in the type of c. Then S' is defined by $((\rho, \chi, ar, C'), A')$, where $S = ((\rho, \chi, ar, C), A)$ and $A' = A \cup \{c = E\}$ and $C' = C \cup \{(c \mapsto \tau)\}$ where τ is the type of E.

A type definition will be denoted as follows:

$$S' = S \uplus [\textbf{typedef}\ \text{“}T(\alpha_1, \ldots, \alpha_n) = \{x \mid P(x)\}\text{”}] \ .$$

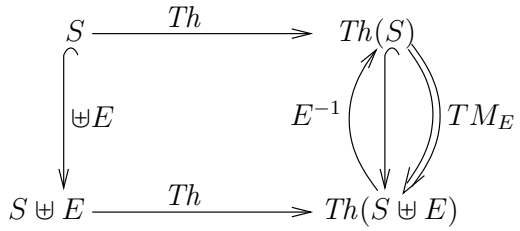
In this case, $S' = ((\rho, \chi', ar', C'), A')$ is defined as follows: We assume $S = ((\rho, \chi, ar, C), A)$, and $P(x)$ of type $P :: R \to \mathit{bool}$ for a base type R in χ. C' is constructed from C by adding $\mathit{Abs}_T : R \to T$ and $\mathit{Rep}_T : T \to R$. χ' is constructed from χ by adding the new type T (i.e. which is supposed to be not in χ). The axioms A' is constructed by adding the two isomorphism axioms

$$A' = A \cup \{\forall x. \mathit{Abs}_T(\mathit{Rep}_T(x)) = x, \forall x. P(x) \implies \mathit{Rep}_T(\mathit{Abs}_T(x)) = x\} \ .$$

The type definition is conservative if the proof obligation $\exists x. P(x)$, holds.

Instead of $S \uplus E_1 \uplus \cdots \uplus E_n$ we write $S \uplus E$. Technically, conservative language embeddings are represented as *specification increments* E, that contain the type definitions and constant definitions for the language elements and give a semantics in terms of a specification S.

The overall situation is summarized in the following commutative diagram:



The three morphisms on the right of the diagram require some explanation: The injection ($\hookrightarrow$) from $Th(S)$ to $Th(S \uplus E)$ is a consequence of the fact that $\uplus$

constructs extensions and Th is required to be monotonous. The theory morphism E^{-1} exists, since our extensions are *conservative*: all new theorems can be retranslated into old ones, which implies that the new theory is consistent whenever the old was (see [5] for the proof). The theory morphism TM_E (denoted by $\Rightarrow$) connects the $Th(S)$ to $Th(S \uplus E)$ and serves as specification for the overall goal of this paper, namely the construction of a partial function $LIFT_E : Th(S) \to TM_E(TH(S'))$ that approximates the functor TM_E.

Our Framework and Isabelle/HOL. Our chosen meta-logic and implementation platform Isabelle/HOL is the instance of the generic theorem prover Isabelle [10] with higher-order logic (*HOL*) [25, 26]. Isabelle directly implements order sorted types ([24]; Note, however, that we do not make use of the ordering on sorts throughout this paper), and supports the conservative extension schemes abstractly presented above. Isabelle/HOL is the instance of Isabelle that is most sophisticated with respect to proof-support and has a library of conservative theories. Among others, the HOL-core provides type *bool*, the number theories provide *nat* and *int*, the typed set theory provides *set(τ)* and the list specification provides *list(τ)*. Moreover, there are products, maps, and even a specification on real numbers and non-standard analysis. The HOL-library provides several thousand theorems — yielding the potential for reuse in a specialized tool for a particular formal method.

Our Framework in the Light of other Type Systems. It is straightforward to represent our framework in type systems that allow *types depending on types* [27], i.e. the four λ-calculi on the backside of Barendregt's cube. In the weakest of these systems, $\lambda\underline{\omega}$, the same notion of sorts is introduced as in our framework. For example, the sort $*$ in $\lambda\underline{\omega}$ corresponds to *term*. The arities correspond to *kinds*, which are limited to $*$ in $\lambda\underline{\omega}$, however, since kinds are defined recursively by $\mathbb{K} = *|\mathbb{K} \to \mathbb{K}$, there are *higher-order* type constructors in $\lambda\underline{\omega}$ that have no correspondence in our framework. The arities of type constructors can be encoded by kinds: the arity for $\_ \to \_$, namely $[term, term, term]$ corresponds to the kind $* \to * \to *$. Declarations of type synonyms **types** $t(\alpha_1, \ldots, \alpha_n) = T$ correspond to $\lambda\alpha_1 : *, \ldots, \alpha_n : * .T$, etc.

2.2 Embedding Techniques — An Overview

For our approach, it is necessary to study the technique of embeddings realized in a theory morphism in more detail. While these underlying techniques are known since the invention of typed λ-calculi (see for the special case of the quantifiers in [25]), it was not before the late seventies that the overall importance of *higher-order abstract syntax* (a term coined by [28]) for the representation of binding in logical rules and program transformations [29] and for implementations [28] was recognized. The term "shallow embedding" (invented in [30]) extends higher-order abstract syntax (HOAS) to a semantic definition and is contrasted to "deep embeddings". Moreover, throughout this paper, we will distinguish *typed*

and *untyped* shallow embeddings. Conceptually, these three techniques can be summarized as follows:

Deep embeddings represent the abstract syntax as a datatype; variables and constants are thus represented as constants in the meta-logic. A semantics is defined "over" the datatype using a transition relation $\rightarrow_r$ or an interpretation function *Sem* from syntax to semantics.

Untyped shallow embeddings use HOAS to represent the syntax of a language by declaring uninterpreted constant symbols for all constructs *except* variables which are directly represented by variables of the meta-logic; thus, binding and substitution are "internalized" on the meta-level, but not the typing. A semantics is defined similarly to a deep embedding.

Typed shallow embeddings use HOAS but include also the type system of the language in the sense that ill-typed expressions can not be encoded well-typed into the meta-logic. This paves the way for defining the semantics of the language constructs and its functions by a direct definition in terms of the meta-logic, i.e. its theories for e.g. orders, sets, pairs, and lists.

The difference between these techniques and their decreasing "representational distance" is best explained by the simplest example of a typed language: the simple typed λ-calculus itself. The syntax can be declared as follows:

	Deep	Untyped Shallow	Typed Shallow
VAR:	$\alpha \rightarrow$L(α ,β)		
CON:	$\beta \rightarrow$L(α ,β)	$\beta \rightarrow$L(β)	
LAM:	$\alpha \times$L(α ,β) $\rightarrow$L(α ,β)	(L(β)$\rightarrow$L(β))$\rightarrow$L(β)	L(γ,δ)$\rightarrow$L(γ,δ)
APP:	L(α ,β)$\times$L(α ,β)$\rightarrow$L(α ,β)	L(β)$\times$L(β)$\rightarrow$L(β)	L(γ,δ)$\times\gamma\rightarrow\delta$

where the underlying types can be defined by the equations:

Deep	Untyped Shallow	Typed Shallow
$L(\alpha,\beta) = \alpha \mid \beta$	$L(\beta) = \beta$	
$\mid \alpha \times L(\alpha,\beta)$	$\mid L(\beta) \rightarrow \times L(\beta)$	$L(\gamma,\delta) = \gamma \rightarrow \delta$
$\mid L(\alpha,\beta) \times L(\alpha,\beta)$	$\mid L(\beta) \times L(\beta)$	

The first type equation can be directly interpreted as a datatype and is thus inductive, the second can interpreted as datatype only with difficulties (requiring reflexive Scott Domains), while the third has clearly no inductive structure at all. Since the typed shallow embedding "implements" binding and typing efficiently by the meta-level, it is more suited for tool implementations. However, induction schemes over the syntax usually yield the crucial weapon for completeness proofs in various logics, for instance, and motivate therefore the use of deep embeddings in meta-theoretic reasoning.

To complete, we compare now the definition of semantics in all three settings:

Deep	Untyped Shallow	Typed Shallow
APP(LAM(x,F),A) $\rightarrow_\beta$	APP(LAM(F),A) $\rightarrow_\beta$F A	APP(F,A) = F A
subst(*alfa*(F,*free*(A)),x,A)		LAM(F) = F
+ congruence rules	+ congruence rules	

where $\rightarrow_\beta$ is just the usual inductively defined β-reduction relation, *subst* and *free* the usual term functions for substitution and computation of free variables and *alfa* is assumed to compute an α-equivalent term whose bound variables are disjoint from *free*(A). In an untyped shallow setting, these functions are not needed since variables and substitution are internalized into the meta-language. In the typed shallow embedding, APP is *semantically* represented by the application of the meta-language and LAM by the identity; the β-reduction APP(LAM(F),A) = F A is just a derived equality in the meta-logic. In a meta-logic assuming Leipnitz' Law for equality (such as HOL), congruence rules are not needed since equality is a universal congruence.

Note that the mapping in our typed shallow embedding between language and meta-language must not be so trivial as it is in this example; it can involve exception handling, special evaluation strategies such as call by value, backtracking, etc. Moreover, the relation between the type systems of the two languages may also be highly non-trivial. This is what our running example OCL will do in the next chapters.

Further, note the technical overhead between deep and shallow embeddings will even be worse if we introduce function symbols such as + and numbers 0,1,2, ... into our language. In the deep embedding, the whole syntax and semantics must be encoded into new datatypes and reduction relations over them, while in the typed shallow embedding, the operators of the meta-logic (possibly adapted semantically) can be reused more or less directly.

Summing up, a deep embedding on the one end of the spectrum requires a lot of machinery for binding, substitution and typing, while a the other end, binding and typing are internalized into the meta-logic, paving the way for efficient implementations using directly the built-in machinery of the theorem prover. Therefore, whenever we speak of an embedding in the sequel, we will assume a typed shallow embedding.

2.3 OCL in a Nutshell

The Unified Modeling Language (UML) is a diagrammatic specification language for modeling object oriented software systems. UML is defined in an open standardization process lead by the Object Management Group (OMG) and highly accepted in industry. Being specialized for the object-oriented software development process, UML allows to specify object-oriented data models (via class diagrams), using data encapsulation, *subtyping* (inheritance), *recursion* (in datatypes and function definitions) and *polymorphism* (overwriting).

While UML as a whole can only claim to be a semi-formal language, UML class-diagrams can be completed by the *Object Constraint Language* (OCL) to a (fully) formal specification language. A prominent use of OCL in [19] is the specification of class invariants and pre and post conditions of methods, e.g.:

```
context Account
  inv:  Account.allInstances->forAll(a1,a2 |
                        a1 <> a2 implies a1.id <> a2.id)
```

```
context Account::makeWithdrawal(amount:Real)
  pre:  (amount > 0) and (balance − mount) >= 0
  post: balance = balance@pre − amount
        and currency = currency@pre
```

The first example requires, that the attribute id of the class **Account** is unique for all instances in a given system state. The second example shows a simple pre/post condition pair, describing a method for withdrawal on an **Account** object. Note, that within post conditions one can access the previous state by using the **@pre**-keyword.

Being a typed logic that supports reasoning over object-graphs defined by object-oriented class diagrams, OCL reasons over path expressions of the underlying class diagram. Any path can be undefined *in a given state*; thus, the undefinedness is inherent in OCL.

3 Organizing Theory Morphisms into Layers

In practice, language definitions follow a general principle or a common scheme. In OCL, for example, there is the following requirement for functions except the explicitly mentioned logical connectors (_ **and** _, _ **or** _, **not** _) and the logical equality (_ = _):

> Object Constraint Language Specification [19] (version 1.4), page 6–58
>
> Whenever an OCL-expression is being evaluated, there is the possibility that one or more queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

In more standard terminology, one could rephrase this semantic principle as "all operations are strict", which is a special principle describing the handling of exceptions[1]. Further semantic principles are, for example, "all collection types are smashed" (see below), or, principles related to the embedding technique.

Instead of leaving these principles implicit inside a large collection of definitions, the idea is to capture their essence in *combinators* and to make these principles in these definitions explicit. Such combinators occur both on the level of types in form of type constructors and on terms in form of constant symbols.

As such, this approach is by no means new; for example, for some semantic aspects like exception handling or state propagation, *monads* have been proposed as a flexible means for describing the semantics of a language "facet by facet" in a modular way [31, 32]. While we will not use monads in this work (which is a result of our chosen standard example, OCL, and thus accidental), and while we do not even suggest a similar fixed semantic framework here, merely a discipline to capture these principles uniformly in combinators (may they have monad structure or not), we will focus on the potential of such a discipline, namely to express their theory once and for all and to exploit it in tactical programs.

[1] In this view, the logical equality can be used to "catch exceptions".

We turn now to the layering of our theory morphism. We say that a theory morphism is layered, iff in each form of conservative extension the following decomposition is possible:

$$\begin{aligned}
&\textbf{types} && \text{“}T('a_1 \ldots' a_m) = C_n(\cdots(C_1(T'))\text{”}\\
&\textbf{typedef} && \text{“}T(\alpha_1, \ldots, \alpha_m) = \{x :: C_n(\cdots(C_1(T')) \mid P(x)\}\text{”}\\
&\textbf{constdefs} && \text{“}c = (E_n \circ \cdots \circ E_1)(c')\text{”}
\end{aligned}$$

where each C_i or, respectively, E_i are (type constructor) expressions build from *semantic combinators* of *layer* S_i and T' respectively. Note, that c' is a construct from the meta logic. A *layer* S_i is represented by a specification defining the *semantic combinators*, i.e. constructs that perform the semantic transformation from meta-level definitions to object-level definitions. In Fig. 1, we present a classification for such layers.

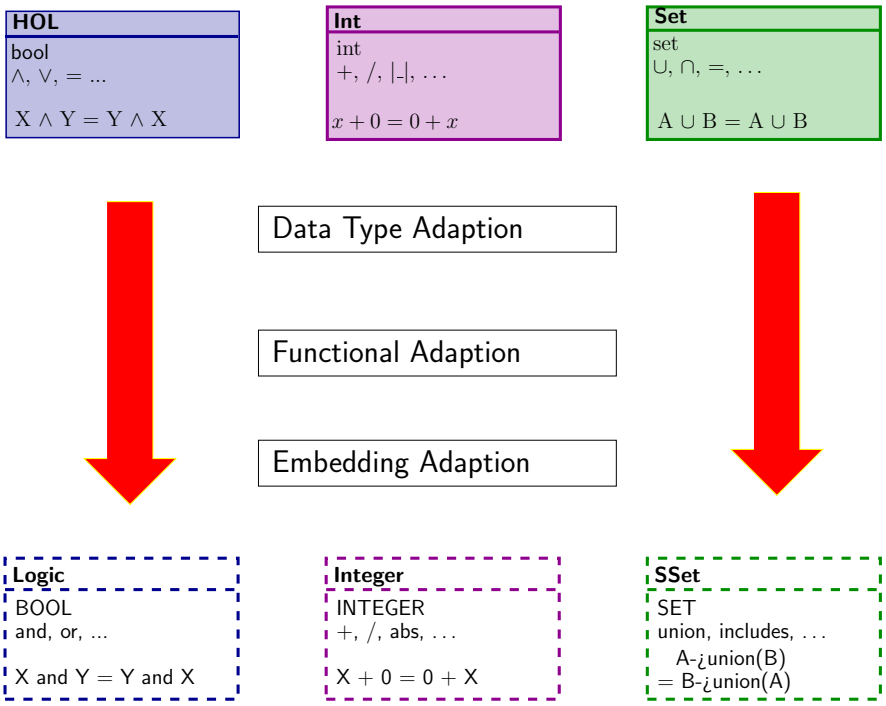


Figure 1. Derivation of the OCL-library

In the following sections, we will present a typical collection of layers and their combinators. We will introduce the semantic combinators one by one and collect them in a distinguished variable SEMCOM. Finally, we will put them together for our example OCL and describe generic theorem proving techniques that exploit the layering of the theory morphism for OCL.

3.1 Datatype Adaption

Datatype adaption establishes the link between meta-level types and object-level types and meta-level constants to object-level constants. While meta-level definitions in libraries of existing theorem prover systems are geared toward good tool support, object-level definitions tend to be geared to a particular computational model, such that the gap between these two has to be bridged. For example, in Isabelle/HOL, the *head*-function applied to an empty list is defined to yield an arbitrary but fixed element; in a typical executable object-language such as SML, Haskell or OCL, however, this function should be defined to yield an exception element that is treated particularly. Thus, datatype adaption copes with such failure elements, the introduction of boundaries (as maximal and minimal numbers in machine arithmetics), congruences on raw data (such as *smashing*; see below) and the introduction of additional semantic structure on a type such as complete partial orders (cpo).

We chose the latter as first example for a datatype adaption. We begin with the introduction of a "simple cpo" structure via the specification extension by sort *cpo*0 and the definition of our first semantic (type) combinator; simple cpo means that we just disjointly add a failure-element such as $\bot$ (see, e.g. [1], where the following construction is also called "lifting"). Note, that an extension to full-blown cpo's would require the additional definition of the usual partial definedness-ordering with $\bot$ as least element and completeness requirements; such an extension is straight-forward and useful to give some recursive constructs in OCL a semantics but out of the scope of this paper.

We state:

$$\textbf{datatype}\ \mathrm{up}(\alpha\ ) = \text{"}\lfloor(\_)\rfloor\text{"}\ \ \alpha \mid \bot$$

which is a syntactic notation for a type definition and two constant definitions for the injections into the sum-type. In the sequel, we write $t_\bot$ instead of $up(t)$. For example, we can define the object-level type synonym *Bool* based on this combinator:

$$\textbf{types}\ \mathit{Bool} = \mathit{bool}_\bot \qquad \textbf{types}\ \mathit{Integer} = \mathit{integer}_\bot \qquad \ldots$$

These type abbreviations reflect the effect of the datatype adaption.

We turn now to the semantical combinators of this layer. We define the inverse to $\lfloor\_\rfloor$ as $\lceil\_\rceil$. We have defined a small specification extension providing the semantic combinators: $(\_)_\bot, \bot, \lfloor\_\rfloor, \lceil\_\rceil \in \mathrm{SEMCOM}$.

As an example for a congruence construction, we chose *smashing on sets*, which occurs in the semantics of SML or OCL, for example. In a language with semantic domains providing $\bot$-elements, the question arises how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can have:

$$(\bot, X) \neq \bot \qquad \{a, \bot, b\} \neq \bot \qquad \ldots$$

or:

$$(\bot, X) = \bot \qquad \{a, \bot, b\} = \bot \qquad \ldots$$

The latter variant is called *smashed product* and *smashed set*. In our framework, we define a semantic combinator for smashing as follows:

constdefs smash :: [[β:: cpo0, α :: cpo0] →bool, α] →α
"smash f X ≡ if f ⊥X then ⊥else X"

and define, for example, *Set*'s as follows:

typedef α Set = "{X:: (α :: cpo0) set up.(smash (λx X. x : ⌈X⌉) X) = X}"

An embedding of smashed sets into "simple cpo's" can be done as follows:

instance Set :: ord(term)
arities Set :: cpo0(term)
constdefs UU_Set_def "⊥ ≡Abs_{Set}⊥"

We have defined the semantic combinators smash, $\bot :: Set(\alpha), \mathrm{Abs}_{Set}, \mathrm{Rep}_{Set} \in$ SEMCOM.

3.2 Functional Adaption

Functional adaption is concerned with the semantic transformation of a meta-level function into an object-level function. For example, this may involve the

- *strictification* of functions, i.e. the result of the function is undefined if one of its arguments is undefined,
- *late-binding-conversion* of a function. This semantic conversion process is necessary for converting a function into an function in an object-oriented language.

Technically, strictification can be achieved by the definition of the semantic combinators. We will introduce two versions: a general one on the type class *cpo*0, another one for the important variant:

constdefs
strictify :: "($\alpha_\bot$ →β:: cpo0) →$\alpha_\bot$→β"
"strictify f x ≡ if x=⊥ then ⊥else f x"
strictify' :: "($\alpha_\bot$ →β:: cpo0) →$\alpha_\bot$→β"
"strictify' f x ≡ case x of ⌊v⌋ →(f v) | ⊥→⊥"

(strictify', strictify ∈ SEMCOM).

A definition like OCL's union (that is the strictified version of HOL's union over the smashed and transformed HOL datatype *set*) is therefore represented as:

constdefs
union :: Set(α) →Set(α) →Set(α)
"union ≡strictify(λ X. strictify(λ Y. Abs_{Set}⌊⌈Rep_{Set}X⌉ ∪⌈Rep_{Set}Y⌉⌋))"

Many object-oriented languages provide a particular call-scheme for functions, called *method invocation* which is believed to increase the reusabilty of code. Method invocation is implemented by a well-known construction in programming language theory called *late-binding*. In order to demonstrate the flexibility of our framework, we show in the following example how this important construction can be integrated and expressed as a semantic combinator. The late-binding-conversion requires a particular pre-compilation step that is not semantically treated by combinators: For each method declaration

Method m: $\mathrm{t}_1, \ldots, \mathrm{t}_n \rightarrow \mathrm{t}$

in a class-declaration A, a look-up table lookup_m has to be declared with type:

$\mathrm{lookup}_m :: \mathrm{set(A)} \rightarrow \mathrm{A} \rightarrow \mathrm{t}_1 \times \ldots \times \mathrm{t}_n \rightarrow \mathrm{t}$

In an "invocation" $\mathrm{A.m}(\mathrm{a}_1, \ldots, \mathrm{a}_n)$ of a "method of object A", the dynamic type of A is detected, which is used to lookup the concrete function in the table, that is executed with A as first argument (together with the other arguments). The dynamic type of a "class of objects A" can be represented by set[2]. Thus, the semantics of method invocations can be given by the following semantic combinators:

match lookup obj $\equiv$the (lookup (LEAST X: α . X : dom lookup $\wedge$obj: X)))
methodify lookup obj arg $\equiv$(match lookup obj)(arg)

where we use predefined Isabelle/HOL functions for "the", "dom" and "LEAST" with the 'obvious' meaning. Since OCL possesses subtyping but *not* late-binding at the moment, we will not apply these combinators throughout this paper. The discussion above serves only for the demonstration that late-binding can in fact be modeled in our framework. A detailed account on the handling of subtyping can be found in [17].

3.3 Embedding Adaption for Shallow Embedding

This type of semantic combinators is related to the embedding technique itself. Recalling section 2.2, any function $op : T_1 \rightarrow T_2$ of the object-language has to be transformed to a function:

$$Sem_\sigma[\![op]\!] : V_\sigma(T_1) \rightarrow V_\sigma(T_2) \text{ where types } V_\sigma(\delta) = \sigma \rightarrow \delta \ .$$

The transformation is motivated by the usual form of a semantic definition for an operator op and an expression e in a deep embedding:

$$Sem_\sigma[\![op\, e]\!] = \lambda\sigma.(Sem_\sigma[\![op]\!]\sigma)(Sem_\sigma[\![e]\!]\sigma)$$

for some environment or state σ. Consequently, the semantics of an expression e of type T is given by a function $\sigma \rightarrow T$ (written as $V_\sigma(T)$). In a typed shallow

[2] This requires a construction of a "universe of objects" closed under subtypes generated by inheritance; in [17], such a construction can be found.

embedding, the language is constructed directly without the detour of the concrete syntax and *Sem*. Hence, all expressions are converted to functions from their environment to their value in T, which implies that whenever a language operators is applied to some arguments, the environment must be passed to them accordingly. This "plumbing" with the environment parameter σ is done by the semantic combinators K, lift_1 or $\text{lift}_2 \in$ SEMCOM that do the trick for constants, unary or binary functions. They are defined as follows:

$$\begin{array}{rrl}
\text{K} & :: & \alpha \to \text{V}_\sigma(\alpha) \\
\text{"K a} & \equiv & (\lambda \text{st. a})\text{"} \\
\text{lift}_1 & :: & (\alpha \to \beta) \to \text{V}_\sigma(\alpha) \to \text{V}_\sigma(\beta) \\
\text{"lift}_1\ \text{f X} & \equiv & (\lambda \text{st. f (X st)})\text{"} \\
\text{lift}_2 & :: & ([\alpha, \beta] \to \gamma) \to [\text{V}_\sigma(\alpha), \text{V}_\sigma(\beta)] \to \text{V}_\sigma(\gamma) \\
\text{"lift}_2\ \text{f X Y} & \equiv & (\lambda \text{st. f (X st)(Y st)})\text{"}
\end{array}$$

Our "layered approach" becomes particularly visible for the example of the logical absurdity or the the logical negation operator (standing for similar unary operators):

$$\begin{array}{rrl}
\textbf{constdefs}\ \bot_{\mathscr{L}} & :: & \text{V}_\sigma(\text{Bool}) \\
\text{"}\bot_{\mathscr{L}} & \equiv & \text{K}(\lfloor \bot \rfloor)\text{"} \\
\mathsf{true} & :: & \text{V}_\sigma(\text{Bool}) \\
\text{"}\mathsf{true} & \equiv & \text{K}(\lfloor \text{ true } \rfloor)\text{"} \\
\mathsf{false} & :: & \text{V}_\sigma(\text{Bool}) \\
\text{"}\mathsf{false} & \equiv & \text{K}(\lfloor \text{ false } \rfloor)\text{"} \\
 & & \\
\text{not} & :: & \text{V}_\sigma(\text{Bool}) \to \text{V}_\sigma(\text{Bool}) \\
\text{"not} & \equiv & (\text{lift}_1 \circ \lfloor _ \rfloor \circ \text{strictify'})\ (\neg)\text{"}
\end{array}$$

From this definition, the usual logical laws for a strict negation can be derived:

$$\mathsf{not}(\bot_{\mathscr{L}}) = \bot_{\mathscr{L}} \qquad \mathsf{not}(\mathsf{true}) = \mathsf{false} \qquad \mathsf{not}(\mathsf{false}) = \mathsf{true}$$

As an example for a binary function like Union (based on union defined in the previous section), we present its definition:

$$\begin{array}{rrl}
\textbf{constdefs}\ \text{Union} & :: & \text{V}_\sigma(\text{Set}(\alpha)) \to \text{V}_\sigma(\text{Set}(\alpha)) \to \text{V}_\sigma(\text{Set}(\alpha)) \\
\text{Union} & \equiv & \text{lift}_2\ \text{union}
\end{array}$$

We will write BOOL for V_σ(Bool), INTEGER for V_σ(Integer) and SET(α) in the sequel. These type abbreviations reflect the effect of the embedding adaption on types.

4 Automatic Generation of Library Theorems

We distinguished two ways to generate theorems for newly embedded operators of an object language: instantiations from generic theorems over the semantic combinators or the application of $LIFT_E$, a tactic procedure that attempts to reconstruct meta-level theorems on the object-level.

4.1 Generic Theorems

In our example application OCL, definedness is a crucial issue that has been coped with by semantic combinators. Definedness is handled by the predicate $\text{is\_def} : V_\sigma(\alpha) \to \text{BOOL}$ that lifts the predicate DEF $t \equiv (t \neq \bot)$ to the level of the OCL logic. Since the latter "implanted" undefinedness on top of the meta-level semantics, it is not surprising that there are a number of properties that are valid for all functions that are defined accordingly to the previous sections.

$$\begin{aligned}
&\text{is_def}\,(\text{lift}_1\,(\text{strictify'}(\lambda x.\ \lfloor f\ x \rfloor))\ X) = \text{is_def}\ X\\
&\text{is_def}\,(\text{lift}_1\,(\text{strictify'}(\lambda x.\ \lfloor f\ x \rfloor))\ X) = \text{is_def}\ X\\
&\text{is_def}\,(\text{lift}_2\,(\text{strictify'}(\lambda x.\\
&\qquad\qquad \text{strictify'}(\lambda y.\ \lfloor f\ x\ y \rfloor)))\ X\ Y) = (\text{is_def}\ X \text{ and } \text{is_def}\ Y)
\end{aligned}$$

$$\begin{aligned}
&\text{lift}_1\,(\text{strictify'}\ f)\ \bot_{\mathscr{L}} = \bot_{\mathscr{L}}\\
&\text{lift}_2\,(\text{strictify'}(\lambda x.\ \text{strictify'}(f\ x)))\ \bot_{\mathscr{L}} X = \bot_{\mathscr{L}}\\
&\text{lift}_2\,(\text{strictify'}(\lambda x.\ \text{strictify'}(f\ x)))\ X\ \bot_{\mathscr{L}} = \bot_{\mathscr{L}}
\end{aligned}$$

For any binary function defined in the prescribed scheme, these theorems already result in four theorems simply by instantiating f appropriately!

Surprisingly, the embedding adaption combinators K, lift_1 and lift_2 turn out to have a quite rich theory of their own. First, it is possible to characterize the "shallowness" of a context C in the sense that the environment/store is just "passed through" this context. This characterization can be formulated semantically and looks as follows:

$$\begin{aligned}
&\textbf{constdefs}\ \text{pass} &&:: ([V_\sigma(\gamma), \sigma] \to \beta) \to \text{bool}\\
&\qquad\qquad \text{pass(C)} &&\equiv (\exists f.\ \forall\ X\ st.\ C\ X\ st = f\ (X\ st)\ st)
\end{aligned}$$

This predicate enjoys a number of useful properties that allow for the decomposition of a larger context C to smaller ones; for instance, trivial contexts pass and passing is compositional:

$$\begin{aligned}
&\text{pass}(\lambda X.\ c) \qquad\qquad \text{pass}(\lambda X.\ X)\\
&[\![\ \text{pass}\ P;\ \text{pass}\ P'\ ]\!] \Rightarrow \text{pass}(P \circ P')
\end{aligned}$$

Moreover, any function following the prescribed scheme is shallow (since this was the very reason for introducing the pass-predicate):

$$\begin{aligned}
&[\![\ \text{pass}\ P\ ]\!] &&\Rightarrow \text{pass}(\lambda X.\ \text{lift}_1\ f\ (P\ X))\\
&[\![\ \text{pass}\ P;\ \text{pass}\ P'\ ]\!] &&\Rightarrow \text{pass}(\lambda X.\ \text{lift}_2\ f\ (P\ X)\ (P'\ X))"
\end{aligned}$$

This leads to a side-calculus enabling powerful logical rules like trichotomy (for the language composed by the operators):

$$\begin{aligned}
&[\![\ \text{pass}\ P;\ \text{pass}\ P';\ P\ \bot_{\mathscr{L}} = P'\ \bot_{\mathscr{L}};\ P\ \mathsf{true} = P'\ \mathsf{true};\ P\ \mathsf{false} = P'\ \mathsf{false}]\!]\\
&\quad \Rightarrow P\ X = P'\ X
\end{aligned}$$

Moreover, there are also fundamental rules that allow for a split of defined and undefined cases and that form the bases for the generic lifter to be discussed in the next section:

$$[\![\ \text{pass}\ P;\ \text{pass}\ P';\ P\ \bot_{\mathscr{L}} = P'\ \bot_{\mathscr{L}};\ X \neq \bot_{\mathscr{L}} \Rightarrow P\ X = P'\ X\ ]\!] \Rightarrow P\ (X) = P'\ X$$

4.2 Approximating the TM_E by $LIFT_E$

Now we are ready to describe conceptually the tactic procedure. The main parts of the implementation in Isabelle/HOL are presented in the appendix, see section A. It is based on the set of semantic combinators SEMCOM and their theory, which has been defined elementwise in the previous sections. In order to allow a certain flexibility in the syntactic form of theorems to be lifted, we extend SEMCOM to the set CO with the set of logical connectives of our meta-language (=, $\wedge$, $\vee$ or $\forall$).

The $core(E)$ of a conservative theory extension E is is defined as the map

$$\{(c \mapsto c') | \textbf{constdefs}\ \text{“}c \equiv e(c')\text{”} \in axioms\_of(E) \wedge constants\_of(e) \subseteq CO\}\,,$$

i.e. we filter all constant definitions that are constructed by our semantical combinators and simple logical compositions thereof.

A theorem $thm \in Th(S)$ is *liftable* iff it only contains constant symbols that are elements of $\mathrm{ran}(core(E))$ or a logical connective.

Liftable theorems can now be converted by substituting the constants in the term of *thm* along $core(E)$, i.e. we apply an inverse signature morphism constructed from $core(E)$ (note that the inverse signature morphism may not be unique; in such cases, all possibilities must be enumerated). A converted theorem may be *convertable* iff the converted term is typable in $\Sigma \uplus E$. All convertable terms thm' are fed as proof goals into a a generic tactical proof procedure that executes the following steps (exemplified with the commutativity):

1. the proof-state is initialized with thm', e.g. ((X:: α INTEGER)+ Y)=Y+X,
2. we apply extensionality and unfold the definitions for lift_1 and lift_2 yielding
   1. $\bigwedge$st. strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)
3. for each of the free variables (e.g. X and Y) we introduce a case split over definedness $DEFx$, i.e. difference of x from $\perp$ (e.g. DEF(Xst) and DEF($Y\ st$)),
   1. $\bigwedge$st. $[\![$ DEF (X st); DEF (Y st) $]\!]$
      $\Rightarrow$ strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)
   2. $\bigwedge$st. $[\![$ DEF (X st); $\neg$DEF (Y st) $]\!]$
      $\Rightarrow$ strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)
   3. $\bigwedge$st. $\neg$ DEF (X st)
      $\Rightarrow$ strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (X st) (Y st)
      = strictify'(λx. strictify' (λy. $\lfloor$x + y$\rfloor$)) (Y st) (X st)
4. we exploit the additional facts in the subgoals by simplifying with the rules for strictify$'$. This yields:
   1. $\bigwedge$st x xa. $[\![$ ... $]\!]$ $\Rightarrow$x + xa = xa + x
5. and by applying *thm* (the commutativity on *int*) we are done.

These steps correspond to the treatment of the different layers discussed in the previous chapter: step one erases the embedding adaption layer, step two establishes case distinctions for all occurring variables and applies generic lemmas for the elimination of the semantic combinators of functional layer. In an example involving a datatype adaption layer (for example quotients like smashing in OCL), similar techniques will have to be applied.

Of course, this quite simple — since conceptual — lifting routine can be extended to a more sophisticated one that can cover a larger part of the set of convertables. For example, the combinators of the datatype adaption layer may involve reasoning over invariants that must be maintained by the underlying library functions. In our OCL theory, for example, such situations result in subproofs for

$$[\![\ \bot \notin \mathrm{Rep}_{Set}\mathrm{A};\ \bot \notin \mathrm{Rep}_{Set}\mathrm{B}\ ]\!] \Rightarrow \bot \notin (\mathrm{Rep}_{Set}\ \mathrm{A} \cup \mathrm{Rep}_{Set}\mathrm{B})$$

Depending from the complexity of the combinators for the datatype adaption, such invariant proof can be arbitrarily complex and will require hand-proven invariance lemmas.

A particular advantage of our approach is that the lifting of theorems can be naturally extended to the lifting of the *configurations* of the automatic proof engine as well. With configuration, we mean here a number of rule sets for introduction and elimination rules for the classical reasoner `fast_tac` or `blast_tac` and sets for standard rewriting or ACI rewriting. By $LIFT_E$, these sets can be partially lifted and extended by corresponding rules on the object level. Since it is usually an expert task to provide a suitable configuration for a logic, this approach attempts to systematically extend this kind of expert knowledge from the meta-level to object level.

5 Experience gained from our OCL example

We give a short overview of the application of our approach in the typed shallow embedding of OCL into Isabelle/HOL (see [17, 18] for details). In our example scenario, we can profit a lot from the fact, that most of the functions for the datatypes Integer, Real (e.g. $=, -, /, \leq, <, \ldots$), Sequences (e.g. union, append, size, etc.), and String (e.g. concat, size, ...) can be derived in the same way as described for + in the last section.

The current application of our module `thy_morpher.ML` to our OCL embedding with 85 operators produces the following statistics (based on Isabelle/HOL version 98):

```
Relevant HOL theorems : 1593
Liftable theorems     :  423
Convertable theorems  :  212
Lifted theorems       :  102
Generic theorems      :  254
```

From the 85 operators of OCL, 77 are amenable to our approach in principle. With "relevant theorems" we mean those contained in specifications imported by the specifications containing our embedding. From our experience, improvements in the generic theorems section will lead to better results easily. In contrast, the design of new schemata of lifting proof routines is a more complex, but still rewarding task. Summing up, based on a still quite simple $LIFT_E$ technology, we successfully generated over 350 theorems which are automatically derived from the base libraries and generic theorems over semantic combinators.

6 Conclusion

We have presented a method for organizing the mass of library function definitions for typed shallow embeddings in a layered theory morphism. Moreover, we developed a technique that allows for the exploitation of this structure in a tactic-based (partial) program that lifts meta-level theorems to their object-level counterparts and meta-level prover configurations to object-level ones. Our approach can be seen as an attempt to liberate the shallow embedding technique from the "point-wise-definition-style" in favor of more global semantic transformations from one language level to another. We abstracted the underlying conceptual notions into a generic framework that shows that the overall technique is applicable in a wide range of embeddings in type systems; embedding-specific dependencies arise only from the specifications of semantic combinators (the *layers*), and technology specific dependencies from the used tactic language.

At present, the technique is limited essentially to the class of first-order Horn-clause equations; for this class, the (partial) program succeeds in our application in all cases in our non-trivial application language. Although a more precise characterization of success is impossible here due to the generality of the framework, we believe that the approach will be applicable for language embeddings for SML, Haskell or Z [16] with similar success since the underlying semantic combinators are the same. Additionally, our implementation of $LIFT_E$ will also be reusable. The same holds for many basic generic theorems over semantical combinators from the embedding adaption layer, the functional adaption layer and — to a lesser extent — the data adaption layer. In principle, the overall construction is also applicable for other higher-order typed theorem proving systems such as Coq [8] or ALF [33]; however, the theories over the semantic combinators and the core of the tactic procedure will have to be adapted to these frameworks.

Besides the obvious need for more generic theorems and more powerful lifting proof procedures, in particular for formulae like $\neg\forall x : A.Px = \exists x : A.\neg Px$, the potential of our approach for untyped shallow or even deep embeddings should be explored. This means, that similarly to invariance proofs of data adaption operators, automatic proofs for the maintenance of well-typing have to be constructed, whereas in a deep embedding, the invariance of binding correctness ("no name-clashes") has also be handled in these proof routines. Beyond the obvious increase of complexity, it seems unclear what kind of limitations for such a setting will arise.

References

[1] Winskel, G.: The Formal Semantics of Programming Languages. MIT Press (1993)
[2] Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing **10** (1998) 171–186
[3] Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. Prentice Hall International Series in Computer Science (1992)
[4] Kolyang, Santen, T., Wolff, B.: A structure preserving encoding of Z in Isabelle/HOL. In von Wright, J., Grundy, J., Harrison, J., eds.: TPHOLs. LNCS 1125, Springer (1996)
[5] Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge Press (1993)
[6] Nipkow, T., von Oheimb, D., Pusch, C.: μJava: Embedding a programming language in a theorem prover. In Bauer, F.L., Steinbrüggen, R., eds.: Foundations of Secure Computation. Volume 175 of NATO Science Series F: Computer and Systems Sciences., IOS Press (2000) 117–144
[7] http://www.nuprl.org.
[8] http://pauillac.inria.fr/coq/.
[9] http://isabelle.in.tum.de.
[10] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
[11] http://www.ora.on.ca/z-eves/welcome.html.
[12] http://svrc.it.uq.edu.au/pages/Ergo.html.
[13] http://i11www.ira.uka.de/ kiv/.
[14] Reetz, R.: Deep Embedding VHDL. In E.T. Schubert, P.J. Windley, J. Alves-Foss, eds.: 8th International Workshop on Higher Order Logic Theorem Proving and its Applications. Volume 971 of Lecture Notes in Computer Science., Springer (1995) 277–292
[15] Ozols, M.A., Eastaughffe, K.A., Cant, A., Collignon, S.: DOVE: A tool for design modelling and verification in safety critical systems. In: 16th International System Safety Conference. (1998)
[16] Brucker, A.D., Rittinger, F., Wolff, B.: HOL-Z 2.0: A proof environment for Z-specifications. Journal of Universal Computer Science **9** (2003)
[17] Brucker, A.D., Wolff, B.: A proposal for a formal OCL semantics in Isabelle/HOL. In Muñoz, C., Tahar, S., Carreño, V., eds.: Theorem Proving in Higher Order Logics. Number 2410 in LNCS. Springer (2002) 99–114
[18] Brucker, A.D., Wolff, B.: HOL-OCL: Experiences, consequences and design choices. In Jezequel, J.M., Hussmann, H., Cook, S., eds.: UML 2002: Model Engineering, Concepts and Tools. Number 2460 in LNCS. Springer (2002)
[19] OMG: Object Constraint Language Specification. [22] chapter 6
[20] Warmer, J., Kleppe, A.: The Object Contraint Language: Precise Modelling with UML. Addison-Wesley (1999)
[21] Warmer, J., Kleppe, A., Clark, T., Ivner, A., Högström, J., Gogolla, M., Richters, M., Hussmann, H., Zschaler, S., Johnston, S., Frankel, D.S., Bock, C.: Response to the UML 2.0 OCL RfP. Technical report (2001)
[22] OMG: Unified Modeling Language Specification (Version 1.4). (2001)
[23] Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. Journal of the ACM (JACM) **39** (1992) 95–146
[24] Nipkow, T.: Order-sorted polymorphism in Isabelle. In Huet, G., Plotkin, G., eds.: Logical Environments. (1993) 164–188

[25] Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5** (1940) 56–68
[26] Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press (1986)
[27] Barendregt, H.: Lambda Calculi with Types. In: Handbook of Logic in Computer Science. Clarendon Press (1992) 117–309
[28] Frank Pfenning, C.E.: Higher-order abstract syntax. In: PLDI 1988. (1988) 199–208
[29] G. Huet, B.L.: Proving and applying program transformations expressed with second order patterns. (Acta Informatica)
[30] Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In Stavridou, V., Melham, T.F., Boute, R.T., eds.: Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience. Volume A-10 of IFIP Transactions., Nijmegen, The Netherlands, North-Holland/Elsevier (1992) 129–156
[31] Wadler, P.: Comprehending monads. In: Proc. 1990 ACM Conference on Lisp and Functional Programming. (1990)
[32] King, D.J., Wadler, P.: Combining monads. In: Glasgow functional programming workshop. (1992)
[33] Altenkirch, T., Gaspes, V., Nordström, B., von Sydow, B.: A User's Guide to ALF. Chalmers University of Technology, Sweden. (1994)

A Implementation of the Theory Morpher

```
(*************************************************************************)
(*          Generic Theorem-Morpher Lifting Meta-Level Theorems           *)
(*          ===================================================           *)
(*                                                                        *)
(* (c) 2002          Achim D. Brucker <brucker@informatik.uni-freiburg.de> *)
(*                     Burkhart Wolff   <wolff@informatik.uni-freiburg.de> *)
(*                                                                        *)
(*************************************************************************)



(*************************************************************************)
(*                                                                       *)
(*  Basic Operations                                                     *)
(*                                                                       *)
(*************************************************************************)

   fun store_thm2 p (name,thm) =
       let val clean_dot = Lib.String.map
                               (fn x => if x = #"." then #"_" else x)
       in bind_thm(clean_dot((p())^name),thm) end;

   fun constants_of (Const(s,_)) = [s]
     | constants_of (Abs(_,_,t)) = constants_of t
     | constants_of (s $ t)      = (constants_of s)@(constants_of t)
     | constants_of _            = [];

   fun free_of (Free(s,_))       = [s]
     | free_of (Abs(_,_,t))      = free_of t
     | free_of (s $ t)           = (free_of s)@(free_of t)
     | free_of _                 = [];

   val SEM_COM         = ["op o", "Let",
                          "Lifting.lift1","Lifting.lift2",
                          "Lifting.lift", "Lifting.drop", "Lifting.UU",
                          "OCL_Set.Abs_SSet", "OCL_Set.Rep_SSet"]

   val strict_symbols = ["Lifting.strictify'","Lifting.strictify"]

   fun subtract xs xt = filter (fn x => forall(fn y => not(x = y)) xt)(xs)

   fun find_thym_pair (name,thm) = (* (cterm,cterm,thm) option *)
       case concl_of thm of
          (Const("==",_ ) $ lhs $
            (Const ("Lifting.lift2", _) $
               (Const(strict1_op,_) $ Abs (_,_,
                Const(strict2_op,_) $ Abs(_,_,rhs)))))
           =>(if exists (fn x => x=strict1_op) strict_symbols andalso
                 exists (fn x => x=strict2_op) strict_symbols
              then (case subtract (constants_of rhs) SEM_COM of
                      [t] => Some ((lhs,Const(t,dummyT)),(name,thm))
                    | _   => None)
              else None)
       | (Const("==",_ ) $ lhs $
            (Const ("Lifting.lift1", _) $ rhs))
           => (case subtract (constants_of rhs) SEM_COM of
                 [t] => Some ((lhs,Const(t,dummyT)),(name,thm))
               | _   => None)
       | _ => None;


   fun core thy =
       mapfilter (find_thym_pair) (axioms_of thy);
```

```
(*************************************************************************)
(*                                                                       *)
(*  Generic Theorem Generator                                            *)
(*                                                                       *)
(*************************************************************************)

(* basic lemmas proof code *)

goalw thy []
"!!g. f == lift2 (strictify'(%x. strictify'(g x))) ==> f X undef = undef";
auto();
qed"lift2_undef2_fw";

goalw thy []
"!!g. f == lift2 (strictify'(%x. strictify'(g x))) ==> f undef X = undef";
auto();
qed"lift2_undef1_fw";

(* [ ... ] ( analogous lemma proof code cut away in presentation ) *)

(* basic lemmas *)


fun mRS S (name,thm) = mapfilter (fn a => Some(name,thm RS a)
                                  handle THM _ => None) (S);

fun derive_sieve thms =
(* generates from a list of definitions several classes of rules
   and provides setup for automatic procedures ... *)
    let val trans  = flat (map (mRS [st_trans_lift1_fw,
                                     st_trans_lift2_fw]) thms)
        val undefs = flat (map (mRS [lift2_undef2_fw,lift2_undef1_fw,
                                     lift2_undef1a_fw,lift2_undef2a_fw,
                                     lift1b_undef_fw,
                                     lift1_undef_fw]) thms)
        val is_defs= flat (map (mRS [lift1_strict_is_isdef_fw,
                                     lift1_strictify_is_isdef_fw,
                                     lift2_strict_is_isdef_fw,
                                     lift2_strictify_is_isdef_fw]) thms);
        val toggle              = ref(~1);
        fun toggle_name name() = (toggle:= ((!toggle+1) mod 2);
                                  name^Int.toString(!toggle))
    in
        writeln("\n derive_sieve: \n");
        writeln("\n==============\n");
        writeln("st_trans - rules: "^Int.toString(length trans));
        writeln("undef    - rules: "^Int.toString(length undefs));
        writeln("is_def   - rules: "^Int.toString(length is_defs));

        AddSIs (map (fn (_,t)=>t) trans);
        map (store_thm2 (K"OCL_st_trans")) trans;

        Addsimps (map (fn (_,t)=>t) undefs);
        map (store_thm2 (toggle_name"OCL_undef")) undefs;

        Addsimps (map (fn (_,t)=>t) is_defs); (* really ??? *)
        map (store_thm2 (toggle_name"OCL_is_def")) is_defs;

        () (* type *)
    end;


(*************************************************************************)
(*                                                                       *)
(*  The Lifter (LIFT_E in documentation)                                 *)
(*                                                                       *)
(*************************************************************************)
```

```
fun convert_default_thy_subst (name,thm) =
    let fun conv (Free(s,_))     = Free(s,dummyT)
          | conv (Var((s,_),_)) = Free(s,dummyT)
          | conv (Const(s,t))    = (case Symtab.lookup(subst, s) of
                                      None      => Const(s,dummyT)
                                    | Some(p,a) => p)
          | conv (Abs(s,_,t))    = Abs(s,dummyT,conv t)
          | conv (s $ t)         = (conv s) $ (conv t)
          | conv x               = x (* Bound variable *)
        val cc = conv(term_of(cprop_of thm))
        fun fst (x,y)            = x
        fun type_inference ct    = fst(Sign.infer_types (sign_of default_thy)
                                       (K None) (K None) [] true ([ct],propT))
     in Some(name,cterm_of(sign_of default_thy)(type_inference cc))
        handle _ => None
     end;


(* Some example proof code for a theorem-lifting

goalw thy  [plus_def,lift2_def]
"((X::'a INTEGER)+ Y)=Y+X";
br ext 1;
by(case_tac "DEF(X st)" 1);
by(case_tac "DEF(Y st)" 1);
by(ALLGOALS(asm_full_simp_tac (HOL_ss addsimps [not_DEF_X_up])));
by(ALLGOALS(asm_full_simp_tac (HOL_ss addsimps [DEF_X_up])));
auto();
br zadd_commute 1;
qed"plus_commute";

*)


fun eq_lifter_prover thms (name,goal) =
    let val thyname  = hd(Lib.String.tokens(fn x => x = #".")(name))
        val core_thm = get_thm(theory_of thyname)(name)
                       handle _ => refl (* just to prevent a crash here*)
        val v        = distinct(free_of(term_of goal))
        val fnm      = map (fn X=> "DEF("^X^" st)") v
        val case_tacs= map (fn X=> case_tac X 1) fnm
    in  Some(name,prove_goalw_cterm thms goal
                (fn prems => [ cut_facts_tac prems 1,
                               rtac ext 1]
                             @ case_tacs @
                             [ ALLGOALS(asm_full_simp_tac
                                        (HOL_ss addsimps [not_DEF_X_up])),
                               ALLGOALS(asm_full_simp_tac
                                          (HOL_ss addsimps [DEF_X_up])),
                               auto_tac (claset(),simpset()),
                               rtac core_thm 1]))
        handle _ => None
    end;


fun lift_root_thy core_E thms =
    let fun ran core_E   = map (fn ((_,y),_)=> y) core_E;
        val the_logical_connectives = ["Trueprop","op =", "All", "==>"]
        val the_ran_core_E          = (map (fn (Const(s,_))=> s)(ran core_E))
        fun is_liftable (name,thm)  =
                       let val cc = (* map (fn (Const(s,_))=> s) *)
                                    (constants_of(term_of(cprop_of thm)))
                           val cc1= subtract cc the_logical_connectives
                           val cc2= subtract cc1 the_ran_core_E
                       in  not(null(cc1)) andalso null(cc2) end
        val liftables    = filter is_liftable thms;
```

```
        val subst        =
                Symtab.make(map(fn((x,Const(s,_)),a)=>(s,(x,a)))core_E)
        val convertables = mapfilter (convert root_thy subst) liftables
        val defs         = lift1_def::lift2_def::
                               (map (fn (_,(_,t)) => t) core_E)
        val proven_lifts = mapfilter (eq_lifter_prover defs) convertables
    in
        writeln("\n Lifting from HOL to OCL: \n");
        writeln("=========================\n");
        writeln(" HOL Theorems : "^Int.toString(length thms));
        writeln(" Liftables    : "^Int.toString(length liftables));
        writeln(" Convertables : "^Int.toString(length convertables));
        writeln(" Proven Lifts : "^Int.toString(length proven_lifts));
        map (store_thm2 (K"OCL_")) proven_lifts;
        ()
    end;


(*************************************************************************)
(*                                                                       *)
(*  Control                                                              *)
(*                                                                       *)
(*************************************************************************)


fun main root_thy =
    let val (E,holthys)
                = partition(fn x => subthy(Lifting.thy,x))
                           (root_thy::(ancestors_of root_thy));
        val core_E = flat (map core E);
        val _      = derive_sieve (map (fn(_,x)=>x) core_E);
        val thms   = flat (map thms_of holthys);
        val _      = lift root_thy core_E thms;

    in  ()
    end;
```

Symbolic Test Case Generation for Primitive Recursive Functions

Achim D. Brucker and Burkhart Wolff

Information Security, ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland.
{brucker, bwolff}@inf.ethz.ch

Abstract We present a method for the automatic generation of test cases for HOL formulae containing primitive recursive predicates. These test cases can be used for the animation of specifications as well as for black-box testing of external programs.
Our method is two-staged: first, the original formula is partitioned into test cases by transformation into a Horn-clause normal form (HCNF). Second, the test cases are analyzed for ground instances satisfying the premises of the clauses. Particular emphasis is put on the control of test hypotheses and test hierarchies to avoid intractability.
We applied our method to several examples, including AVL-trees and the red-black tree implementation in the standard library from SML/NJ.
Keywords: symbolic test case generations, black box testing, theorem proving, Isabelle/HOL

1 Introduction

Today, essentially two software validation techniques are used: *software verification* and *software testing*. Whereas verification is rarely used in "large-scale" software development, testing is widely used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call *Dijkstra's verdict* [10, p.6]:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

More recently, three research areas, albeit driven by different motivations, converge and result in a renewed interest in testing techniques:

- *Abstraction Techniques:* model-checking raised interest in techniques to abstract infinite models to finite ones. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [5, 9].
- *Systematic Testing:* the discussion over *test adequacy criteria* [20], i.e., criteria answering the question "when did we test enough to meet a given test hypothesis", led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [11, 12].

– *Specification Animation:* constructing counter-examples has raised interest also in the theorem proving community, when combined with animations of evaluations, they may help to find modeling errors early and to increase the overall productivity [14].

The first two areas are motivated by the question "are we building the program right?", the latter is focused on the question "are we specifying the right program?". While the first area shows that Dijkstra's Verdict is no longer true under all circumstances, the latter area shows that it simply does not apply to important situations in practice. In particular, if a formal model of the environment of a software system (e.g., based on, amongst other things, the operating system, middleware or external libraries) must be reverse-engineered, testing — in the sense of "experimenting" — is without alternative (see [7]).

Following standard terminology [20], our approach is a *specification-based unit test*. A test procedure for such an approach can be divided into:

– *Test Case Generation:* for each operation, the pre/post-condition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.
– *Test Data Selection:* for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.
– *Test Execution:* the implementation is run with the selected test input data in order to determine the test output data.
– *Test Result Verification:* the pair of input/output data is checked against the specification of the test case.

As an example for a specification-based unit-test approach, QuickCheck [8] has attracted interest in various research communities. QuickCheck performs random tests, potentially improved by hand-programmed test data generators, and provides a simple test execution and test result verification environment for programs written in Haskell.

However, it is well-known that random test can be ineffective in many cases[1]; in particular, if complex preconditions of programs like "the input tree must be balanced" or "the input must be a well-formed abstract syntax tree" rule out most of randomly generated data. In our approach, we will exploit the specification of pre- and postconditions of a program — the *test specification* — in a preprocessing step, the *test case generation*. Our implementation `TestGen` of a test case generator is built on top of the theorem prover Isabelle/HOL [17]. Isabelle is programmed to execute the underlying symbolic computations in an automatic, but logically safe way. Based on the resulting *test cases*, a random test based data selection procedure can be controlled in a problem-oriented way and achieve a significantly better test coverage. As a particular feature, the automated deduction-based process can log the test hypothesis underlying the test.

[1] Consider `abs(x-2) >= 0` where `abs` computes the absolute value over the Haskell data type `Int`. Here it is very unlikely that QuickCheck finds the problem.

Provided that the test hypotheses are valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification *and* the test hypotheses.

We proceed as follows: we will introduce our implementation built on top of the theorem prover Isabelle by a tiny, but classical example (Sec. 2). This demonstration serves as a means to motivate concepts like *test specification*, *testing normal form*, *test cases*, *test statements*. In Sec. 3, we will discuss the test case generation in more detail. In Sec. 4, we will discuss a technique for controlling the *state explosion* by generating *abstract test cases*. Finally, we apply our technique to a number of non-trivial examples (Sec. 5) involving recursive data types and recursive predicates and functions over them.

2 Symbolic Test Case Generation: A Guided Tour

Our test case generator `TestGen` is integrated into the specification and theorem proving environment Isabelle/HOL. As a specification language, HOL offers data types, recursive function definitions and fairly rich libraries with theories of, e.g., arithmetics; it is often viewed as a "functional programming language with logical quantifiers". As a theorem proving environment, Isabelle is based on a relatively small proof engine (based on higher-order resolution) providing a *proof state* that can be transformed via elementary *tactics* into logically equivalent ones, until a final proof state is reached where a derived formula has the appropriate form.

Our running example for automatic test case generation is described as follows: given three integers representing the lengths of the sides of a triangle, a small algorithm has to check, whether these integers describe an equilateral, isosceles, scalene triangle, or no triangle at all. First we define an abstract data type describing the possible results in Isabelle/HOL:

datatype Triangles := equilateral | scalene | isosceles | error

For clarity (and as an example for specification modularization) we define an auxiliary predicate deciding if the three lengths are describing a triangle:

$$\begin{aligned}\textbf{constdefs}\ & \mathit{triangle} :: [\mathrm{nat}, \mathrm{nat}, \mathrm{nat}] \to \mathrm{bool} \\ & \mathit{triangle}\ x\ y\ z \equiv (0 < x) \wedge (0 < y) \wedge (0 < z) \wedge (z < x + y) \\ & \qquad \wedge (x < y + z) \wedge (y < x + z)\end{aligned}$$

Now we define the behavior of the triangle program by initializing the internal Isabelle proof state with the test specification TS:

$$\begin{aligned}\texttt{prog}(x, y, z) = \textbf{if}\ & \mathit{triangle}\ x\ y\ z\ \textbf{then} \\ & \textbf{if}\ x = y\ \textbf{then} \\ & \quad \textbf{if}\ y = z\ \textbf{then}\ \text{equilateral}\ \textbf{else}\ \text{isosceles} \\ & \textbf{else if}\ y = z\ \textbf{then}\ \text{isosceles} \\ & \quad \textbf{else if}\ x = z\ \textbf{then}\ \text{isosceles}\ \textbf{else}\ \text{scalene} \\ & \textbf{else}\ \text{error}\end{aligned}$$

Note that the variable `prog` is used to label an arbitrary implementation as the current *program under test* that should fulfill the test specification.

In the following we show how our test package `TestGen` can be applied to the automatic test data generation problem for the triangle problem. Our method proceeds in the following steps:

1. By applying `gen_test_case_tac` we bring the proof state into *testing normal form* (TNF). In this example, we decided to generate test cases up to depth 0 (discussed later) and to unfold the *triangle* predicate by its definition before the process. This leads to a formula with 26 clauses, among them:

$$\begin{aligned} [\![0 < z; z < z + z]\!] &\Longrightarrow \texttt{prog}(z, z, z) = \text{equilateral} \\ \left[\!\!\left[\begin{array}{l} x \neq z; 0 < x; 0 < z; \\ z < x + z; x < z + z \end{array} \right]\!\!\right] &\Longrightarrow \texttt{prog}(x, z, z) = \text{isosceles} \\ [\![y \neq z; z \neq y; \neg z < z + y]\!] &\Longrightarrow \texttt{prog}(z, y, z) = \text{error} \end{aligned}$$

   We call each Horn-clause of the proof state a *symbolic test case*. As a result of `gen_test_case_tac`, we can extract the current proof state and get the *test theorem* which has the form $[\![A_1; \ldots; A_{26}]\!] \Longrightarrow TS$ where the A_i abbreviate the above test cases.
2. We compute the concrete *test statements* by grounding the symbolic test cases for "`prog`" via a random test procedure (`genadd_test_data`). The latter operation selects the test cases from the test theorem and produces the test statements (excerpt):

$$\texttt{prog}(3, 3, 3) = \text{equilateral} \qquad \texttt{prog}(4, 6, 0) = \text{error}$$

A test statement can be compiled into a test program by simply mapping all operators to external code (where `prog` is the code for calling the program under test). This can be automated with Isabelle's code-generator. If such a compilation is possible for a formula A, i.e., if A only consists of constant symbols for which this map is defined, we call A *executable*. This definition essentially rules out unbounded logical quantifiers and more arcane HOL constructs like the Hilbert-operator.

In our triangle example, standard simplification was able to eliminate the assumptions of the (grounded) test cases automatically. In general, assumptions in test statements (also called *constraints*) may remain. Provided that all test statements are executable, clauses with constraints can nevertheless be interpreted as an abstract test program. For its result, three cases may be distinguished: (i) if one of the clauses evaluates to false, the test is *invalid*, otherwise *valid*. A valid test may be (ii) a *successful test* if and only if the evaluation of all conclusions (including the call of `prog`) also evaluates to true; (iii) otherwise the test contains at least one *test failure*. Rephrased in this terminology, the ultimate goal of the test data selection is to construct successful tests, which means that ground substitutions must be found that make the remaining *constraints* valid.

Coming back to our example, there is a viable alternative for the process above: instead of unfolding *triangle* and trying to generate ground substitutions satisfying the constraints, one may keep *triangle* in the test theorem, treating it as a building block for new constraints. It turns out that a special test theorem and test data (like "$\mathit{triangle}(3,4,5) = \text{True}$") can be generated "once and for all" and inserted before the *test data selection* phase producing a "partial" grounding. It will turn out that the main state explosion is shifted from the test case generation to the test data selection phase, possibly at the cost of test adequacy. This technique to modularize test data generation will be discussed in Sec. 4 in more detail.

3 Concepts of Test Case Generation

As input of the test case generation phase, the *test specification*, one might expect a special format like $\text{pre}(x) \rightarrow \text{post}\ x\ (\texttt{prog}(x))$. However, this would rule out trivial instances such as $3 < \texttt{prog}(x)$ or just $\texttt{prog}(x)$ (meaning that `prog` must evaluate to True for x). Therefore, we do not impose any other restriction on a specification other than the final test statements being executable, i.e., the result of the process can be compiled into a test program.

Processing this test specification, our method `gen_test_case_tac` can be separated into the following conceptual phases (in reality, these phases were performed in an interleaved way):

- *Tableaux Normal Form Computation:* via a tableaux calculus (see Tab. 1), the specification is transformed into Horn-clause normal form (HCNF).
- *Rewriting Normal Form Computation:* via the standard rewrite rules the current specification is simplified.
- *TNF Computation:* by re-ordering of the clauses, the calls of the program under test are rearranged such that they only occur in the conclusion, where they must occur at least once.
- *TNF Minimization:* redundancies, e.g., clauses subsumed by others, are eliminated.
- *Exploiting Uniformity Hypothesis:* for free variables occurring in recurring argument positions of primitive recursive predicates, a suitable *data separation lemma* is generated and applied (leading to a test hypothesis *THYP*).
- *Exploiting Regularity Hypothesis:* for all Horn-clauses not representing a test hypothesis, a uniformity hypothesis is generated and exploited.

After a brief introduction of concepts and use of Isabelle in our setting, we will follow the sequence of these phases and describe them in more detail in the subsequent sections. We will conclude with a discussion of coverage criteria.

3.1 Concepts and Use of Isabelle/HOL

Isabelle [17] is a generic theorem prover of the LCF prover family; as such, we use the possibility to build programs performing symbolic computations over formulae in a logically safe (conservative) way on top of the logical core engine: this is

what `TestGen` technically is. Throughout this paper, we will use Isabelle/HOL, the instance for Church's higher-order logic. Isabelle/HOL offers support for data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

Isabelle's proof engine is geared towards Horn-clauses (called "subgoals"): $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1}$, written $[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$, is viewed as a rule of the form "from assumptions A_1 to A_n, infer conclusion A_{n+1}". A *proof state* in Isabelle contains an implicitly conjoint sequence of Horn-clauses ϕ_1,...,ϕ_n and a *goal* ϕ. Since a Horn-clause

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

is logically equivalent to

$$\neg A_1 \vee \cdots \vee \neg A_n \vee A_{n+1},$$

a Horn-clause normal form (HCNF) can be viewed as a conjunctive normal form (CNF). Note, that in order to cope with quantifiers naturally occurring in specifications, we generalize the idea of a Horn-clause to Isabelle's format of a *subgoal*, where variables may be bound by a built-in meta-quantifier:

$$\bigwedge x_1, \ldots, x_m.[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}$$

Subgoals and goals may be extracted from the proof state into theorems of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$; this mechanism is used to generate test theorems. The meta-quantifier $\bigwedge$ is used to capture the usual side-constraints "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as free variables. Further, Isabelle supports meta-variables (written $?x, ?y, \ldots$), which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification.

3.2 Normal Form Computations

In this section, we describe the tableaux, rewriting and testing normal form computations in more detail. In Isabelle/HOL, the automated proof procedures for HOL formulae depend heavily on tableaux calculi [13] presented as (derived) natural deduction rules. The core tableaux calculus is shown in Tab. 1 in the Appendix. Note, that with the notable exception of the elimination rule for the universal quantifier (see Tab. 1(c)), any rule application leads to a logically equivalent proof state: therefore, all rules (except $\forall$ elimination) are called *safe*. When applied bottom up in backwards reasoning (which may introduce meta-variables explicitly marked in Tab. 1), the technique leads in a deterministic manner to a HCNF.

Horn-clauses can be normalized by a number of elementary logical rules (e.g., False $\Longrightarrow P =$ True), the usual injectivity and distinctness rules for constructors implied by data types and computation rules resulting from recursive definitions. Both processes together bring an original specification into *Rewriting HCNF.*

However, these forms do not exclude clauses of the form:

$$[\![\neg(\texttt{prog}\ x = c); \neg(\texttt{prog}\ x = d)]\!] \Longrightarrow A_{n+1}$$

where `prog` is the program under test. Equivalently, this clause can be transformed into

$$[\![\neg(A_{n+1})]\!] \Longrightarrow \texttt{prog}\ x = c \vee \texttt{prog}\ x = d$$

We call this form of Horn-clauses *testing normal form* (TNF). More formally, a Horn-clause is in TNF for program under test F if and only if

- F does not occur in the constraints, and
- F does occur in the conclusion.

Note that not all specifications can be converted to TNF. For example, if the specification does not make a suitably strong constraint over program F, in particular if F does not occur in the specification. In such cases, `gen_test_case_tac` stops with an exception.

3.3 Minimizing TNF

A TNF computation as described so far may result in a proof state with redundancies. Redundancies in a proof state may result in superfluous test data and should therefore be eliminated. A proof state may have:

1. several occurrences of identical clauses
2. several occurrences of clauses with subsuming assumption lists; this can be eliminated by the transformation

$$\frac{[\![P; R]\!] \Longrightarrow A; \quad [\![P; Q; R]\!] \Longrightarrow A;}{[\![P; R]\!] \Longrightarrow A;}$$

3. and in particular, clauses that subsume each other after distribution of $\vee$; this can be eliminated by the transformation

$$\frac{[\![P; R]\!] \Longrightarrow A; \quad [\![\neg P; Q]\!] \Longrightarrow B; \quad [\![R; Q]\!] \Longrightarrow A \vee B;}{[\![P; R]\!] \Longrightarrow A; \quad [\![\neg P; Q]\!] \Longrightarrow B;}$$

The notation above refers to logical transformations on a subset of clauses within a proof state and not, as usual, on formulae within a clause. Since in backward proofs the proof state below is a refinement of the proof state above, the logical implication goes from bottom to top.

3.4 Exploiting Uniformity Hypothesis for Recursive Predicates

In the following, we address the key problem of test case generation in our setting, i.e.; recursive predicates occurring in preconditions of a program. As an

introductory example, we consider the membership predicate of an element in a list:

$$\begin{array}{lll} \textbf{primrec} & x\ mem\ [\,] & = \text{False} \\ & x\ mem\ (y\#ys) & = \textbf{if}\ y = x\ \textbf{then}\ \text{True}\ \textbf{else}\ x\ mem\ ys \end{array} \tag{1}$$

which occurs as precondition in an (abstract) program specification:

$$x\ mem\ S \rightarrow \texttt{prog}\ x\ S$$

For the testing of recursive data structure, Gaudel suggested in [12] the introduction of a *uniformity hypothesis* as one possible form of a test hypothesis, a kind of weak induction rule:

$$\frac{\begin{array}{c}[|x| < k] \\ \vdots \\ P\ x\end{array}}{P\ x}$$

This rule formalizes the hypothesis that provided a predicate P is true for all data x whose *size*, denoted by $|x|$, is less then a given depth k, it is always true. The original rule can be viewed as a meta-notation: In a rule for a concrete data-type, the premises $|x| < k$ can be expanded to a number of premises enumerating constructor terms.

For all variables in clauses that occur as (recurring) arguments of primitive recursive functions, we will use a testing hypothesis of this kind — called *data separation lemma* — in an exercise in poly-typic theorem proving [19] described in the following.

The Isabelle/HOL data type package generates definitions of poly-typic functions (like case-match and recursors) from data type definitions and derives a number of theorems over them (like induction, distinctness of constructors, etc.). In particular, for any data type, we can assume the size function and reduction rules allowing to compute $\big|[a, b, c]\big| = 3$, for example. Moreover, there is an *exhaustion-theorem*, which for lists has the form

$$[\![y = [\,] \Longrightarrow P; \bigwedge x\ xs.y = x\#xs \Longrightarrow P]\!] \Longrightarrow P$$

Now, since we can separate any data x belonging to a data type τ into:

$$x \in \big\{z :: \tau.|z| < d\big\} \vee x \in \big\{z :: \tau.d \leq |z|\big\} \tag{2}$$

i.e., x is either in the set of data smaller d or in the remaining set. Note that both sets are infinite in general; the bound for the size produces "data test cases" and not just finite sets of data. Consequently, we can derive for each given type τ and each d a destruction rule that enumerates the data of size $0, 1, \ldots, k-1$. For lists and $d = 2, 3$, it has the form:

$$\big(x : \{z :: \alpha\ \text{list}.|z| < 2\} \rightarrow x = [\,]\big) \vee \big(\exists a.x = [a]\big)$$
$$\big(x : \{z :: \alpha\ \text{list}.|z| < 3\} \rightarrow x = [\,]\big) \vee \big(\exists a.x = [a]\big) \vee \big(\exists ab.x = [a, b]\big)$$

Putting lemma (2) together with the destruction rule for $d = 2$, instead of the unsafe uniformity hypothesis in the sense of Gaudel we automatically construct the safe data separation lemma of the form

$$\frac{\begin{array}{cccc}[x = []] & & [x = [a]] & [x = [a,b]] \\ \vdots & & \vdots & \vdots \\ P(x) & \bigwedge a. \quad P(x) & \bigwedge a\ b. \quad P(x) & THYP\big(2 \leq |x| \rightarrow P(x)\big)\end{array}}{P(x)}$$

The purpose of this rule in backward proof is to split a statement over a program into several cases, each with an additional assumption that allows to "rewrite-away" the x appropriately. Here, the constant $THYP :: \text{bool} \rightarrow \text{bool}$ (defined as the identity function) is used to label the test hypothesis in the proof state. Since we do not unfold it, formulae labeled by $THYP$ are protected from decomposition by the tableaux rules shown in Tab. 1.

The equalities introduced by this rule of depth $d = 2$ allow for the simplification of the primitive recursive predicate *mem* which leads to further decompositions during the TNF computation. Thus, for our test specification:

$$x\ mem\ S \rightarrow \texttt{prog}\ x\ S$$

executing `gen_test_case_tac` results in the following TNF:

1. $\texttt{prog}\ x\ [x]$
2. $\bigwedge b.\texttt{prog}\ x\ [x, b]$
3. $\bigwedge a.a \neq x \rightarrow \texttt{prog}\ x\ [a, x]$
4. $THYP(3 \leq |S| \rightarrow x\ mem\ S \rightarrow \texttt{prog}\ x\ S)$

The simplification of the *mem* predicate along its defining rules (1) leads to nested "**if then else**" constructs. Their decomposition during HCNF computation results in the constraint that the lists fulfilling the precondition must have a particular structure. Even the simplest "generate-and-test"-method for test data selection will now produce adequate test statements, while it would have produced mostly test failures when applied directly to the original specification.

The handling of quantifiers ranging over data types can be done analogously: since $\forall x.P(x)$ is equivalent to $\forall x : UNIV.P(x)$ and since $UNIV = \{z :: \tau.|z| < d\} \cup \{z :: \tau.d \leq |z|\}$, the universal quantifier can be decomposed into a finite conjunction for the test cases smaller than d and a test hypothesis $THYP$ for the rest.

From the above example it follows that the general form of a test theorem is $[\![A_1; \ldots; A_n; THYP(H_1); \ldots; THYP(H_m)]\!] \Longrightarrow TS$. Here the A_i represent the test cases, the H_i the test hypothesis, and TS the testing specification.

3.5 Exploiting Regularity Hypothesis

After introducing the regularity hypothesis and computing a TNF (except for clauses containing *THYP*s), we use the clauses to construct another form of

testing hypothesis, namely the *regularity hypothesis* [12] (sometimes also called *partitioning hypothesis*) for each test case. This kind of hypothesis has the form:

$$THYP(\exists x_1, \ldots, x_n.P\ x_1, \ldots, x_n \rightarrow \forall x_1, \ldots, x_n.P\ x_1, \ldots, x_n)$$

This means that whenever there is a successful test for a test case, it is assumed that the program will behave correctly for *all* data of this test case.

Using a uniformity hypothesis for each (non-$THYP$) clause allows for the replacement of free variables by meta-variables; e.g., for the case of two free variables, we have the following transformation on proof states:

$$\frac{[\![A_1\ x\ y; \ldots; A_n\ x\ y]\!] \Longrightarrow A_{n+1}\ x\ y}{[\![A_1\ ?x\ ?y; \ldots; A_n\ ?x\ ?y]\!] \Longrightarrow A_{n+1} ?x\ ?y;\ \ THYP((\exists xy.P\ x\ y) \rightarrow (\forall xy.P\ x\ y))};$$

where $P\ x\ y \equiv A_1\ x\ y \wedge \ldots \wedge A_n\ x\ y \rightarrow A_{n+1}\ x\ y$. This transformation is logically sound. Moreover, the construction introduces individual meta-variables into each clause for the ground instances to be substituted in the test data selection; this representation allows for partial grounding and is also a necessary prerequisite for structured test data selection as discussed in Sec.4.

3.6 Coverage Criteria: A Discussion

In their seminal work, Dick and Faivre [11] propose to transform the original specification into disjunctive normal form (DNF), followed by a case splitting phase converting the disjunctions $A \vee B$ into $A \wedge B$, $\neg A \wedge B$ and $A \wedge \neg B$ and further (logical and arithmetic) simplifications and minimizations on the disjunctions. The resulting cases are also called the *partitions of the specification* or the *(DNF) test cases*. The method suggests the following test adequacy criterion: a set of test data is *partition complete* if and only if for any test case there is a test data. Consequently, a program P is tested adequately to partition completeness with respect to a specification S if it passes a partition complete test data set.

Our notion of a *successful test*, see Sec. 2, is a HCNF based adequacy criterion. DNF and HCNF based adequacy result in the same partitioning in many practical cases, as in the triangle example, while having no clear-cut advantage in others. Since the DNF technique has the disadvantage of producing a double exponential blow-up (the case splitting phase alone can produce an exponential blow-up) while HCNF computation is simply exponential, and since HCNF-computation can be more directly and efficiently implemented in the Isabelle proof engine, we chose the latter.

HCNF adequacy subsumes another interesting adequacy criterion under certain conditions, namely *path coverage* with respect to the specification. Path coverage means that in any (mutual) recursive system of functions, all reachable paths, e.g., of the **if** P **then** A **else** B statements, were activated at least once. For a mutual recursive system consisting only of *primitive* recursive functions, (i.e., with each call the size of data will decrease exactly by one), it can be concluded that if the testing depth d is chosen larger than the size of

the maximal strong component of the call graph of the recursive system, each function is unfolded at least once. Since the unfold results in conditionals that were translated to $(P \to A) \wedge (\neg P \to B)$, any branch will lead to a test case.

Thus, while `gen_test_case_tac` often produces reasonable results for arbitrarily recursive functions, we can assure only for primitive recursions that the underlying HCNF adequacy of our method subsumes path coverage.

4 Structured Test Data Selection

The motivations to separate test data selection from test case generation are both conceptual and technical. Conceptually, test data selection is a process where we would also like to admit more heuristic techniques like random data generation or generate-and-test with the constraints. Test data selection yields sequences of ground theorems (no meta-variables, no type variables); this paves the way for evaluation by compiled code, a new approach is needed to cope with the unavoidable state explosion in the late stages. A purely technical motivation for this separation is Isabelle-related: within a test theorem, it is not possible to instantiate polymorphic type variables α in different ways when generating test statements, however, this flexibility may be desirable.

The generation of a multitude of ground test statements from one test theorem containing the test cases and the test hypothesis is essentially based on a random-procedure followed by a test of the satisfaction of the constraints (similar to QuickCheck). For each type, this default procedure may be overwritten in `TestGen`-specific generators that may be user defined; thus, the usual heuristics like trying $[0, 1, 2, \mathit{maxint}, \mathit{maxint}+1]$ can be easily implemented, or the counter-example generation integrated in Isabelle's arithmetic procedure can be plugged in (which, in our experience, is difficult to control in larger examples).

Now we will discuss the issue of structured test data generation. Similar to theorem proving, the question of "how many definitions should be unfolded" is crucial; exploiting suitable abstractions is the major weapon against complexity. In our first attempt to generate a test theorem for the triangle example (see Sec. 2), the auxiliary predicate *triangle* is unfolded in the test specification. This resulted in the aforementioned 26 cases. If we do not unfold it, the resulting test theorem has only 10 test cases, but contains "abstract constraints" such as:

$$\begin{aligned}
[\![\mathit{triangle}\ z\ z\ z]\!] &\Longrightarrow \mathtt{prog}(z,z,z) = \text{equilateral}\\
[\![\neg\mathit{triangle}\ z\ z\ z]\!] &\Longrightarrow \mathtt{prog}(z,z,z) = \text{error}\\
[\![y \neq z; z \neq y; \mathit{triangle}\ z\ y\ z]\!] &\Longrightarrow \mathtt{prog}(z,y,z) = \text{isosceles}
\end{aligned}$$

Thus, a substantial part of the proof state explosion can be postponed by treating *triangle* as a building block in the constraints or, in other words, by generating more *abstract* test cases.

Now, if we could generate a *local test theorem* for *triangle* as such, generate the *local* test data separately and resolve the resulting test statements for it into the test theorem for the global computation, the state explosion could be shifted

to the test data selection. The trick can be done as follows: we define a trivially true proof goal for:

$$\texttt{prog}(x, y, z) = \mathit{triangle}\ x\ y\ z \Longrightarrow \texttt{prog}(x, y, z) = \mathit{triangle}\ x\ y\ z$$

unfold *triangle* and compute TNF(`prog`). When folding back *triangle* via the assumption we get the following local test cases:

$$\begin{array}{ll}
\neg \mathit{triangle}\ 0\ y\ z & \neg z < x + y \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\
\neg \mathit{triangle}\ x\ 0\ z & \neg x < y + z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\
\neg \mathit{triangle}\ x\ y\ 0 & \neg y < x + z \Longrightarrow \neg \mathit{triangle}\ x\ y\ z \\
\left[\!\!\left[\begin{array}{c} 0 < x; 0 < y; 0 < z; \\ z < x + y; x < y + z; y < x + z \end{array} \right]\!\!\right] \Longrightarrow & \mathit{triangle}\ x\ y\ z
\end{array}$$

which can easily be converted into *abstract test statements* such as *triangle* 1 1 1. When resolving the latter in all combinations into the abstract global test theorem, (partial) ground instances for variables were generated that make random choices for them superfluous. Thus, the test statements of previously developed theories can be reused when building up larger units. Of course, when building up test data in a modular way, this comes at a price: since the local test statements do not have the same logical information available as their application context in a more global test theorem, the instantiation may result in unsatisfiable constraints. Nevertheless, since the criterion for success of a decomposition is clear — at the very end we want constraint-free test statements achieving a full coverage of the TNF— the implementor of a test has more flexibility here helping to deal with larger problems. In our example, there is no loss at all: test data for the local predicate is valid for the global goal, and by construction, the set of test statements is still complete for HCNF coverage.

5 Applications

We applied our method to specifications of two widely used variants of balanced binary search trees: AVL trees and red-black trees. These case studies were performed using Isabelle 2003 compiled with SML of New Jersey running on Linux with 512 MBytes of RAM, and an Intel 1.6 GHz P4 processor.

5.1 AVL Trees

In 1962 Adel'son-Vel'skiĭ and Landis [3] introduced a class of balanced binary search trees (called AVL trees) that guarantee that a tree with n internal nodes has height $O(\log n)$. Based on an AVL-theory from the Isabelle library we generated test cases for the following invariant: if an element y is in the tree after insertion of x in the tree t then either $x = y$ holds or y was already stored in t. Based on the depth 3, this *test specification* leads to an amazing 236 test cases which were computed in less than 30 seconds.

5.2 Red-Black Trees

A widely used variant of balanced search trees was presented by Bayer [4]. In this data structure, the balancing information is stored in one additional bit per node. This is called "color of a node" (which can either be red or black), hence the name *red-black trees*. A valid (balanced) red-black tree must fulfill the following two invariants:

- *Red Invariant:* each red node has a black parent.
- *Black Invariant:* each path from the root to an empty node has the same number of black nodes.

We aimed for testing a "real-world" implementation of red-black trees and decided to test the red-black trees provided in the standard library of SML of New Jersey (SML/NJ) [2]. There, red-black trees are used for implementing finite sets and maps which are intensively used throughout the SML/NJ compiler itself.

Our specification is based on the formalization [16] of the SML/NJ red-black trees (based on version 110.44 of SML/NJ). The specification starts with the basic data type declaration for binary trees:

datatype color $= R \mid B$
α tree $= E \mid T$ color (α tree) (α item) (α tree)

In this example we have chosen not only to check if keys are stored or deleted correctly in the trees but also to check if the trees fulfill the balancing invariants. Therefore our specification has to formalize the red and black invariants. This is done by the following recursive predicates:

consts
redinv :: (α item) tree $\Rightarrow$ bool
blackinv :: (α item) tree $\Rightarrow$ bool

recdef *redinv* "measure ($\lambda t.$(size t))"
"*redinv E* = True"
"*redinv* (*T B a y b*) = (*redinv a* $\wedge$ *redinv b*)"
"*redinv* (*T R* (*T R a x b*) *y c*) = False"
"*redinv* (*T R a x* (*T R b y c*)) = False"
"*redinv* (*T R a x b*) = (*redinv a* $\wedge$ *redinv b*)"

recdef *blackinv* "measure ($\lambda t.$(size t))"
"*blackinvE* = True"
"*blackinv*(*T color a y b*) = ((*blackinv a*) $\wedge$ (*blackinv b*)
$\wedge$((*max_B_height a*) = (*max_B_height b*)))"

We use the following test specification for checking if the delete operation fulfills these invariants:

$$(redinv\ t \wedge blackinv\ t) \rightarrow (redinv\ (delete\ x\ t) \wedge blackinv\ (delete\ x\ t))$$

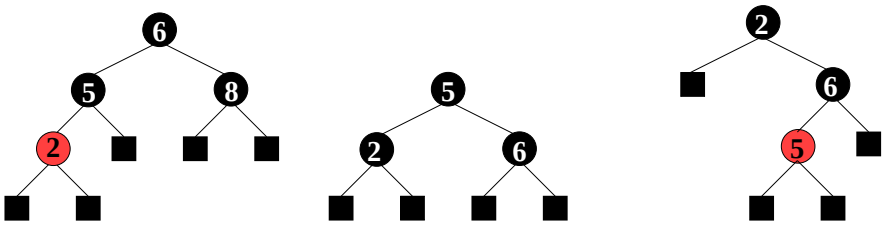
In other words, for all trees the deletion operation maintains the red and black invariant. For testing purposes, we instantiated *item* with Integers. The test case generation takes less than two minutes and results in 348 test cases. Among them

$$\begin{aligned} &\mathit{delete}\ 8\ (T\ B\ (T\ B\ (T\ R\ E\ 2\ E)\ 5\ E)\ 6\ (T\ B\ E\ 8\ E)) \\ &\qquad = (T\ B\ (T\ B\ E\ 2\ E)\ 5\ (T\ B\ E\ 6\ E)) \end{aligned}$$

which describes that the deletion of the node 8 in the tree shown in Fig. 1(a) must result in the tree shown in Fig. 1(b). This test case revealed a major error in the standard library of SML/NJ. Using a simple SML test script one observes:

```
val input = T (B,T (B,T (R,E,2,E),5,E),6,T (B,E,8,E))
- val output = delete(input,8);
val output = T (B,E,2,T (B,T (R,E,5,E),6,E))
```

Obviously, the black invariant does not hold for `output` (see Fig. 1(c)).



(a) pre-state (b) correct result (c) result of SML/NJ

Figure 1. Test Data for Deleting a Node in a Red-Black Tree

This example shows that specification based testing can find efficiency bugs: combinations of insert and delete operations of the SML/NJ implementation easily lead to trees that degenerate to sorted lists. In our case, the revealed flaw has not been detected in the last 12 years, although red-black trees are widely used within the SML/NJ compiler itself. Fixing this bug will presumably lead to a perceptible performance gain of the SML/NJ compiler.

Based on our definitions, the bug could be reproduced by QCheck/SML [1], a QuickCheck-like random testing tool. Although this particular bug can even be found without using a hand-programmed test data generator, the QuickCheck method would have, in general, imposed one to write such a test data generator. Moreover, our method allows to conclude that certain coverage criteria are fulfilled and makes all underlying test hypotheses explicit. Further, our approach can profit from the underlying theories for data-types offering the potential for problem-specific case splits.[2]

[2] ... such as $[\![P(\texttt{minBound} :: \texttt{Int}); a \neq \texttt{minBound} :: \texttt{Int} P(-a)]\!] \Longrightarrow P(-a))$ which also produces the problem $x = \texttt{minBound} :: \texttt{Int} + 2$ for `abs(x-2)>=0` after unfolding `abs`.

6 Conclusion

We have presented the theory and implementation of a test case generator for unit tests. Our approach is focused on functional programs, but since imperative programs can be provided with a functional interface (by compiling a functional call to a statement sequence consisting of (i) initialization, (ii) executing constructors representing data types, (iii) calling the program under test, and (iv) checking the result), this is not a real limitation of our approach. We demonstrated its practical feasibility for the systematic test of libraries of large software systems by testing functions from the SML/NJ library, which revealed a major bug leading to inefficiency in basic data structures of the SML/NJ compiler.

In our opinion, test data generation is an activity that clearly needs *some* user interaction: as in model-checking, one has to experiment with the form of the specifications and basic parameters (depth of *data separation*, the level of abstraction, etc.) in order to get a feasible test data set for the test of a "real program". Therefore, we believe such an activity is best supported by an integration into an *interactive* theorem proving environment such as Isabelle. Since `TestGen` is ca. 400 lines of SML code that is loaded into Isabelle, we still consider our approach fairly "lightweight". Nevertheless, `TestGen` is at present the only implementation of a test case generator that combines state-of-the-art deduction technology based on derived rules (formally proven inside Isabelle) with a powerful logic.

We believe that there is another line of criticism against Dijkstra's verdict. A successful test together with explicitly stated test hypotheses is not fundamentally different from program verification: all sorts of modeling assumptions were made, adding test hypothesis is just one more of them. The nature and trustworthiness of these assumptions may be different, but a clear-cut line between testing and verification does not exist.

6.1 Future Work

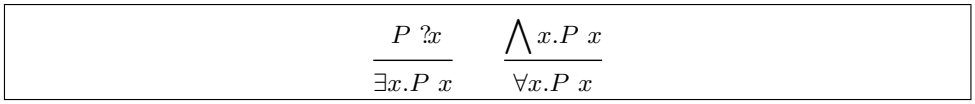
We see the following lines of extension of our work:

1. *Investigating the test hypothesis*: a new test hypothesis (like congruence hypothesis on data, for example) may dramatically improve the viability of the approach. Furthermore, it should be explored if the verification of the test hypothesis for a given abstract program offers new lines of automation.
2. *Better control of the process*: at the moment, our implementation can only be controlled by very globally applied parameters such as depth. The approach could be improved by generating the test hypothesis and the test data depending on the local context within the test theorems.
3. *Integration tests*: integrating/combining our framework into behavioral modeling leads to the generation of *test sequences* as in [15, 18].
4. *Generating test data for many-valued logics* such as *HOL-OCL* [6] should make our approach applicable to formal methods more accepted in industry.

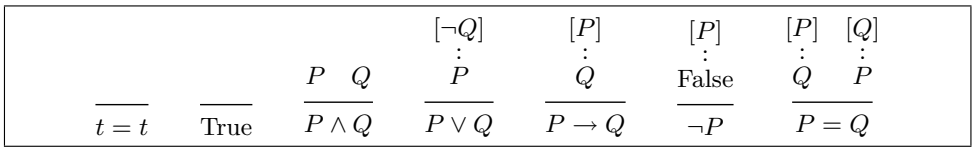
References

[1] QCheck/SML. `http://contrapunctus.net/league/haques/qcheck/`.
[2] SML of New Jersey. `http://www.smlnj.org/`.
[3] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
[4] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.
[5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*. Number 58 in Advances In Computers. 2003.
[6] A. D. Brucker and B. Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In C. Muñoz, S. Tahar, and V. Carreño, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.
[7] A. D. Brucker and B. Wolff. A case study of a formalized security architecture. In T. Arts and W. Fokkink, editors, *FMICS'03*, volume 80 of *Electronic Notes in Theoretical Computer Science*, Roros, 2003. Elsevier Science Publishers.
[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000.
[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.
[10] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 1972.
[11] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *FME 93*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, 1993.
[12] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *LNCS*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.
[13] R. Hähnle. Tableaux for many-valued logics. In M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*, pages 529–580. Kluwer, Dordrecht, 1999.
[14] S. Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.
[15] F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - a tool for distributed systems specification. In *FTRTFT 96*, volume 1135 of *LNCS*, pages 467–470. Springer-Verlag, 1996.
[16] A. Kimmig. Red-black trees of smlnj. Studienarbeit, Universität Freiburg, Freiburg, 2003.
[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
[18] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In E. Brinksma and J. Tretmans, editors, *Proc. Formal approaches to testing of software*, pages 47–60. BRICS, 2001.
[19] K. Slind and J. Hurd. Applications of polytypism in theorem proving. In D. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 103–119. Springer-Verlag, Rome, Italy, 2003.
[20] H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

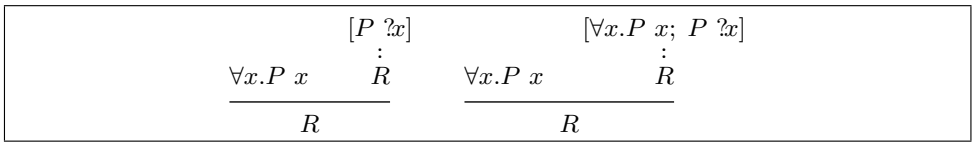
A Appendix



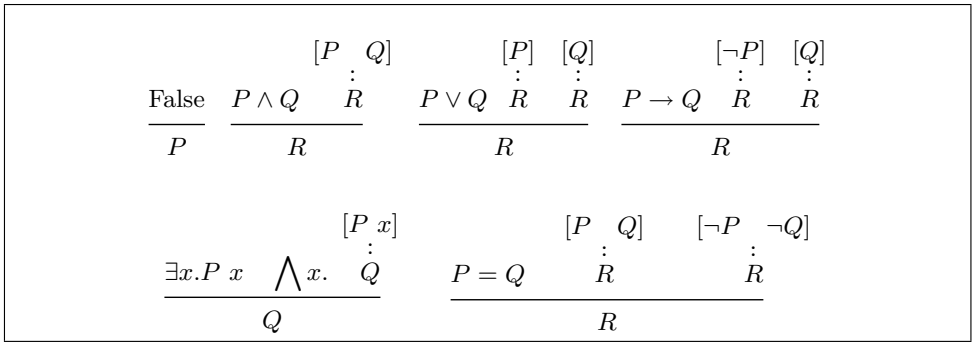
(a) Quantifier Introduction Rules



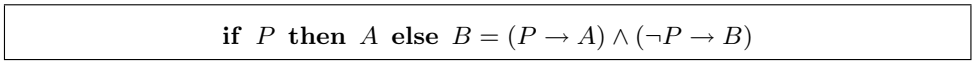
(b) Safe Introduction Rules



(c) Unsafe Elimination Rules



(d) Safe Elimination Rules



(e) Rewrites

Table 1. The Standard Tableaux Calculus for HOL

Part III

Selected Papers: Encapsulation and Tool Integration

HOL-Z in the UniForM-Workbench - a Case Study in Tool Integration for Z *

C. Lüth[1], E. W. Karlsen[1], Kolyang[1], S. Westmeier[1], and B. Wolff[2]

[1] Bremen Institute for Safe Systems, FB 3, Universität Bremen
Postfach 330440, 28334 Bremen
{cxl,ewk,kol,litfox}@informatik.uni-bremen.de
[2] Universität Freiburg, Institut für Informatik
wolff@informatik.uni-freiburg.de

Abstract. The UniForM-Workbench is an open tool-integration environment providing type-safe communication, a toolkit for graphical user-interfaces, version management and configuration management.

We demonstrate how to integrate several tools for the Z specification language into the Workbench, obtaining an instantiation of the Workbench suited as a software development environment for Z. In the core of the setting, we use the encoding HOL-Z of Z into Isabelle as semantic foundation and for formal reasoning with Z specifications. In addition to this, external tools like editors and small utilities are integrated, showing the integration of both self-developed and externally developed tools.

The resulting prototype demonstrates the viability of our approach to combine public domain tools into a generic software development environment using a strongly typed functional language.

1 Introduction

The need for tool integration has been widely recognised throughout software engineering. There is no single tool for all purposes. Moreover, we live in a highly distributed software production culture. Hence, it is likely to be impractical and unproductive to prescribe *ex cathedra* the particular tools to be used in a given development. Rather, it seems more advantageous to let software engineering teams employ the tools they are most comfortable with, and combine the various tools into one integrated *Software Development Environment* (SDE).

In response to this need, a number of tool-integration techniques have been developed. In the most simple approach, tools run independently, with the file system as a persistent store, and communication achieved by string-based "glueing" using Tcl [22], Expect [19] or Emacs Lisp. Unfortunately, this approach does not scale up to more sophisticated development environments, where features such as type-safe communication, persistent and

* This work has been supported by the German Ministry for Education and Research (BMBF) as part of the project UniForM under grant No. FKZ 01 IS 521 B2.

distributed storage, version management and workflow management are required. In particular, this is the case for formal methods, where the semantic integrity of the documents produced by the various formal method tools have to be maintained, in order to enable a fine-grained configuration management of individual theorems, lemmata, proof obligations and proofs.

During the last decade, several attempts to meet this challenge were made based on environments for synthesising *tightly integrated* SDE's from the basis of abstract language specifications such as the Cornell Synthesizer Generator [27], Gandalf [8], PSG [1] or the Ipsen system [20]. These were not entirely satisfactory, due to either the development costs involved in requiring tools to be developed from scratch in a homogeneous language framework, or to the inapplicability of the language framework itself in the problem domain.

In the UniForM approach, tool integration is based on a *loosely coupled* architecture [18,28] where prefabricated tools are integrated on top of a *tool integration framework*. The UniForM Workbench, introduced in section 2, is the implementation of this framework, which offers support for data, control and presentation integration. It is generic, and we will here introduce its instantiation to the Z specification language (section 3), the Z Workbench. The semantic cornerstone of this instantiation is the encoding of Z into the theorem prover Isabelle, called HOL-Z. The encoding lends a semantic aspect to the integration, by providing type checking and the ultimate certification of the correctness of proof-scripts, allowing to maintain the semantic integrity of documents during the development process, and moreover allowing formal reasoning within Z specifications. HOL-Z and Isabelle will be the focus of section 4. This is followed by the main section (section 5) of our paper concerned with the data model of our Z Workbench. We demonstrate the Z Workbench at work with the canonical example at hand in section 6.

2 The UniForM-Workbench

The design of the UniForM-Workbench [13] reflects the guidelines of the ECMA Reference Model [4] (see Figure 1), which outlines the abstract functionality required to support the various dimensions of a tool integration process:

Data Integration addresses the issue of sharing and exchanging data between tools. It is mainly provided by the *repository manager*.

Control Integration is concerned with communication and inter-operation among tools and between tools and the integration framework. It is provided by the *subsystem interaction manager*.

Presentation Integration addresses the issue of tool appearance and user interaction, i.e. look-and-feel. It is provided by the *user interaction manager*.

Process Integration is concerned with the functions between tools of the environment and the end user, i.e. workflow management. It will be provided by the *development manager* (work in progress).

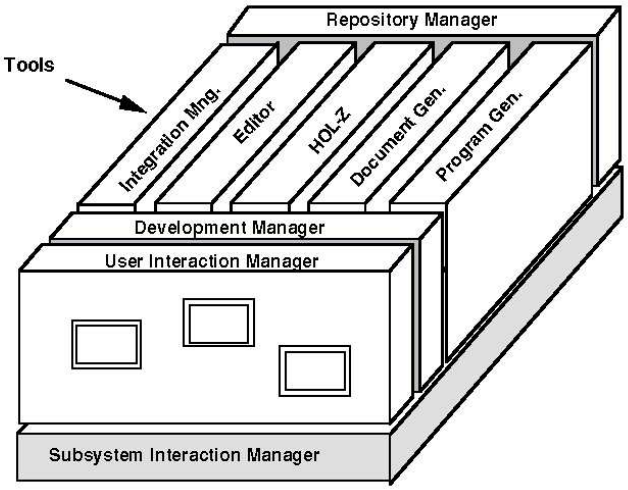


Fig. 1. ECMA Toaster Model

The ECMA reference model is also called the "ECMA Toaster Model", with the integration services in the rôle of the toaster. In this integration framework SDE's are constructed by *encapsulating* existing development tools such as editors, model checkers and proof tools. The pure functional language Haskell [9,24], extended with a higher order approach to concurrency [11] is used as a central integration language, i.e. as GlueWare[1] in this context. Each tool is encapsulated by wrapping Haskell interfaces around it using the integration manager. Section 5 will give an example of this process.

In the UniForM Workbench, an SDE is viewed as a reactive, event driven system. Events in such an environment amount to user interactions of the user interaction manager, change notifications of the active repository manager, operating system events and individual tool events. The subsystem interaction manager [10] takes the rôle of the central control component in this reactive systems architecture. It is structured as a network of communicating agents called *interactors*, whose behaviour is expressed using composable event values in the style of Concurrent ML [26]. New events can be defined from the basis of existing ones, using the *guarded choice* operator that provides a choice between two events, or the *event-action* combinator that combines an event with some additional reactive behaviour. This way, integration can be expressed at a very high level of abstraction.

The repository manager [31] provides databases services for the persistent storage of objects, and their exchange between tools. It is implemented by a Haskell encapsulation of the Portable Common Tool Environment (PCTE) [5] standard extended with version, configuration and workspace management. The repository manager is an *active database*: changes to an object

[1] Thank you to Phil Trinder for coming up with this wonderful term!

(i.e. committing a new version of an object) result in change notification events being sent to all other tools accessing the object. In order to integrate one or more tool, one first develops a data model in terms of the extended entity-relationship model underlying PCTE. From this semi-formal model, the actual implementation in terms of Haskell data types and type classes is derived in a systematic way. During the integration process, each tool is set up to work with persistent objects of the repository manager, rather than the plain files of the file system. Alternatively, the repository manager can export and import objects from the repository into the file system and back; this is less safe, but one does not need to change an existing tool's data access.

The user interaction manager [12] provides Haskell-Tk, a strongly typed, fully concurrent graphical user interface, with which graphical user interfaces for integrated tools can be constructed. Another component, the graph visualization system daVinci [7], is used to provide graphical user interfaces to the repository itself such as version and configuration graphs.

The Workbench obeys established industry standards such as the ECMA Reference Model, PCTE and Motif [6]. It has been implemented on the basis of public domain, off-the-shelf components supporting these standards. The repository manager for example is based on H-PCTE [3], whereas the user interaction manager integrates Tk [22] and the graph visualisation system daVinci [7]. Moreover, the Workbench offers a higher level of abstraction and uniformity than its underlying components, as well as additional utilities to support tool integration. The integration process is therefore much easier than if we had used these tools in their bare-bone form. One example is Haskell-Tk, which, as opposed to Tcl/Tk that it is based on, is strongly typed and fully concurrent. Another example is the repository manager, which provides version, configuration and workspace management on top of H-PCTE.

3 The Z-Workbench

The *Z-Workbench* is a Software Development Environment for Z built using the integration services provided by the UniForM Workbench — in other words, an instantiation of the generic framework provided by the Workbench to software development using Z. Its main component, providing a variety of services and the semantical underpinning of the integration, is the encoding of Z into Isabelle/HOL called HOL-Z [16].

HOL-Z reads Z specifications in the email format and does a type and consistency check (i.e. it will reject specifications which do not typecheck or are obviously inconsistent). One can then prove theorems within the specification, generate formatted documentation, or if the specification can be shown to be executable, generate program code. Thus, HOL-Z offers the following services:

- type and consistency check;
- symbolic theorem proving;

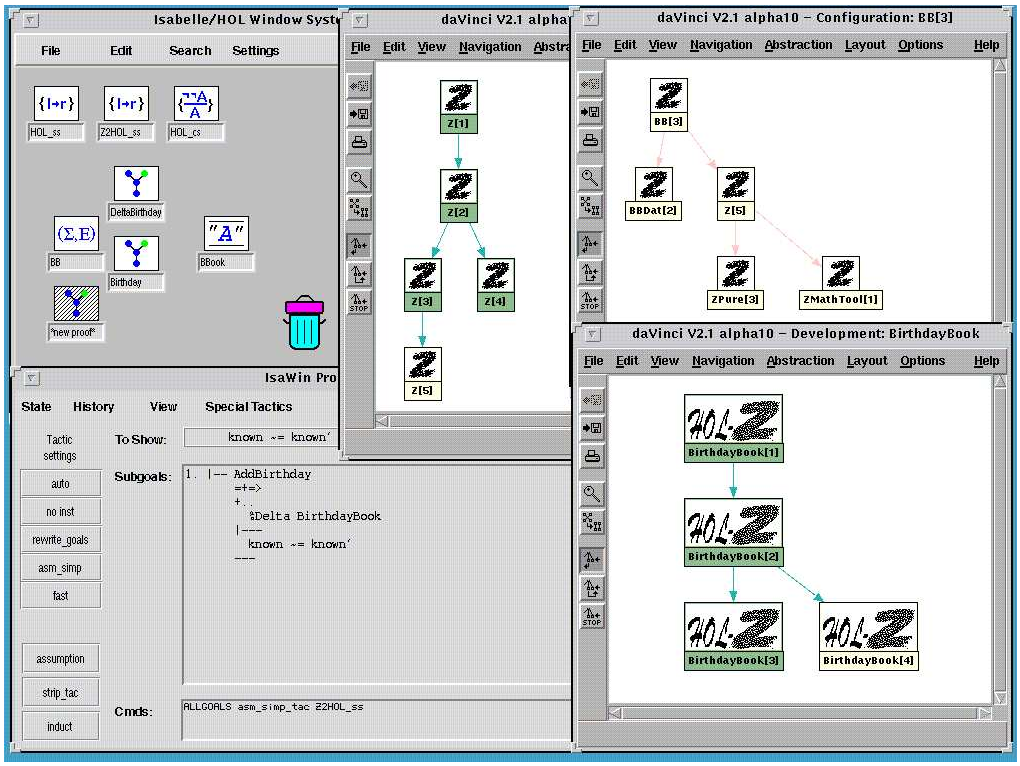


Fig. 2. Z-Workbench

– documentation generation;
– code generation.

Furthermore, to edit specifications a text editor is integrated. At present, this is a basic Unix text editor, but a structure-oriented schema editor could be integrated very easily as well. Finally, utilities such as the Unix tool `diff` are integrated, which is used by the version management.

A session with the Z-Workbench is pictured in Figure 2, where HOL-Z that has been started up in order to develop version `BB[3]` of the birthday book theory. The session has been invoked from within the development graph at the lower right corner by selecting the development named `BirthDayBook[2]` which yields a new session `BirthDayBook[4]`. The corresponding configuration graph for the birthday-book theory is shown in the upper right corner. This configuration imports version `Z[5]` of the standard Z library, whose version graph is in turn shown to the left. The persistent objects taking part in the session are visualized in a brighter colour than objects that are not used by the session. These are also the objects that have been exported to the file system (behind the scene), in order to make them accessible to HOL-Z.

Integrating these tools into a Z-Workbench has numerous advantages over an implementation using stand alone tools working with a plain file system: all objects of the Z-Workbench, such as specifications and proofs are versioned

and permanently kept consistent in a distributed, multi-user environment. Not only Z specifications, but also tools and proofs are put under control of the version management system. Old versions are never deleted, only outdated, and can always be reverted to. Hence, a formal development is always kept consistent — if a specification is outdated, the development using the old, outdated specification is still available.

The views provided by the Z-Workbench are at all times kept consistent with the current state of the repository. Changes to the repository are broadcasted to all running tools, such that changes made by one user can be recognised immediately by others. Developers may for example be notified immediately when a proof has been either proved correct or rejected, to avoid that development resources are wasted on redundant work or blind alleys.

Interaction with the environment is on a query by navigation basis, where the user browses through the object base using graphical user interfaces such as folder, version and configuration graphs visualised by daVinci. Tools are then invoked from within these views.

The Z-Workbench is, being based on the generic integration services of the UniForM Workbench, an *open integration framework* that can be extended with other development tools if needed. However, integration of other tools is not always as easy as could be hoped for, since it can be hampered by idiosyncrasies of prefabricated tools. In particular, the plethora of existing styles for Z specifications (email, LaTeX, box style) and deviations from these styles by existing tools may impose the need of converting a specification before it is passed to a tool. Still, the Workbench provides an excellent framework for hiding such technicalities, since the specifications are treated as logical objects, and the conversions will happen behind the scenes whenever possible.

4 Isabelle, HOL-Z and IsaWin

In short, Isabelle [23] is a *generic tactical LCF theorem prover*. Here, *generic* means that it is particularly suited for the encoding of different logics and formal methods, *tactical* means that it offers user-programmable proof support, and the *LCF design* means that the prover is centred around an abstract data type *theorems*, whose objects can only be constructed by applying the basic logical rules. The overall correctness on all formal activities is based on this (relatively small and well-investigated) logical engine.

As implementation, the prover can be viewed as a collection of ML types modelling theorems, proofs and theories, and ML functions modelling the possible proof activities.

The encoding of Z into Isabelle, called HOL-Z, benefited in particular from Isabelle's genericity, while the LCF design allowed the implementation of a versatile graphical user interface called IsaWin, not only for Isabelle itself, but also for other applications based on Isabelle, such as HOL-Z. We will now describe HOL-Z and IsaWin in greater detail. Their combination will yield

a tool Win-Z for formal reasoning in Z specifications with a graphical user interface.

4.1 HOL-Z

HOL-Z is a shallow embedding which has the ability to preserve the structure of large specifications. This has the advantage that HOL-Z can be used for theorem proving in "real" Z-Specifications.

HOL-Z essentially consists of three parts:

- Isabelle Theories: An Isabelle theory (supporting the e-mail format of Z along the Z Standard [21]), a collection of extensions of the Isabelle/HOL-standard and the mathematical toolkit, the "library" of Z which consists itself of a collection of theories (relations, bags, sequences etc.)
- A loader for Z specifications
- A collection of tactics to support working with Z (prover)

HOL-Z supports the email format as proposed in the Draft Standard for Z. Thus, a schema is given in the e-mail format; a loader converts it into a semantic representation where the corresponding schema is a boolean-valued function.

The following schema known from the Birthday Book will be represented in the email input syntax as follows:

```
"+-- BirthdayBook ---
          birthday:(Name -|-> Date);
          known : Pow Name
|---
          known = dom(birthday)
 ---
"+-- AddBirthday  ---
        date_i:Date;
        name_i:Name;
        %Delta BirthdayBook
|---
        name_i~:known &
        birthday'= birthday Un {(name_i,date_i)}
 ---
"
```

The Z loader takes this schemas and produces a theorem in Isabelle which is pretty printed as shown below:

```
+-- BirthdayBook ---
    birthday : (Naturals -|-> Date); known : Pow Name
   |---
    known = dom(birthday)
   ---
```

4.2 IsaWin and Win-Z

IsaWin is a graphical user interface (GUI) for the theorem prover Isabelle. It represents the theorem prover's objects — theorems, proofs, theories, sets of rewriting rules — by icons. The operations on these objects making up the proof activity, are mostly affected by drag&drop, or to a lesser extent by activating buttons or menu entries. This gives the user access to nearly the full proof power of Isabelle without having to concern himself with ML and its syntax.

An interesting feature of the system architecture of IsaWin is its versatility. Since it is implemented entirely in Standard ML "on top" of Isabelle, it makes use of Standard ML's powerful abstraction and modularisation concepts, in particular its parameterised modules, called *functors*. This way, to obtain a graphical user interface for an encoding or other application based on Isabelle, we merely need to instantiate the functor with different parameters[2].

The instantiation of IsaWin for HOL-Z is called Win-Z. It contains *ZTheories* which are collections of schema declarations (sections) and serve essentially the same functions that Isabelle-theories do during theorem proving - forming the context a theorem is living in. Furthermore they incorporate an environment, *ZEnv*, containing an extralogical information about the occurrence of schemas, to respect the 'define before' principle of Z. In Figure 2, Win-Z can be seen on the left side.

5 The Integration

The development of integrated SDE's within the UniForM Workbench starts out with a number of prefabricated development tools that have not been integrated yet. In order to reach an integrated framework for Z development, issues regarding control, data and presentation integration must be addressed. Control integration means that the tool is given a Haskell API so that it can be controlled by the subsystem interaction manager, data integration that it is set up to interface the development objects maintained by the repository manager, and presentation integration that a graphical user interface along the guidelines of the Motif standard are wrapped around the tool, unless of course, the tool already comes with such an interface.

The resulting SDE is organised according to the *Model-View-Controller paradigm* [17], with the repository manager in the rôle of the *Model*, the user interaction manager in the rôle of the *View* and the subsystem interaction manager in the rôle of the *Controller* [14].

[2] In fact, IsaWin itself is the instantiation of a more generic graphical user interface called GenGUI [15].

5.1 Data Integration

The data model for the Z-Workbench is given in Figure 3 as an extended Entity-Relationship diagram, which is then systematically converted into the actual Haskell modelling (see Section 2). The two most basic object types are *folders* and *versioned objects*. Folders are the basic structuring mechanism within the repository, comparable to directories in conventional file systems. Each folder may however be contained in a number of parent folders, and each folder may contain, in addition, a number of versioned objects. Versioned objects are the basic building blocks of the application. Each versioned object has two kinds of relationships: *dependency* and *development*. Dependency links model structural dependency, such as imports, and are given by all those links between the four subtypes of versioned objects marked Ⓓ in Figure 3. The development relationship defines *revisions* of an object, and can only exist between versioned objects of the same subtype. Dependency links are essential to model change propagation: if a new revision of an object is created, the revisions of the objects depending on this object will have to be updated as well. Thus, if we create a new theory by editing an existing one, all proof scripts living in the old theory will have to be updated as well in order to live in the new theory.

Versioned objects come in five subtypes: *tools*, *theories*, *sessions* and *proof scripts*.

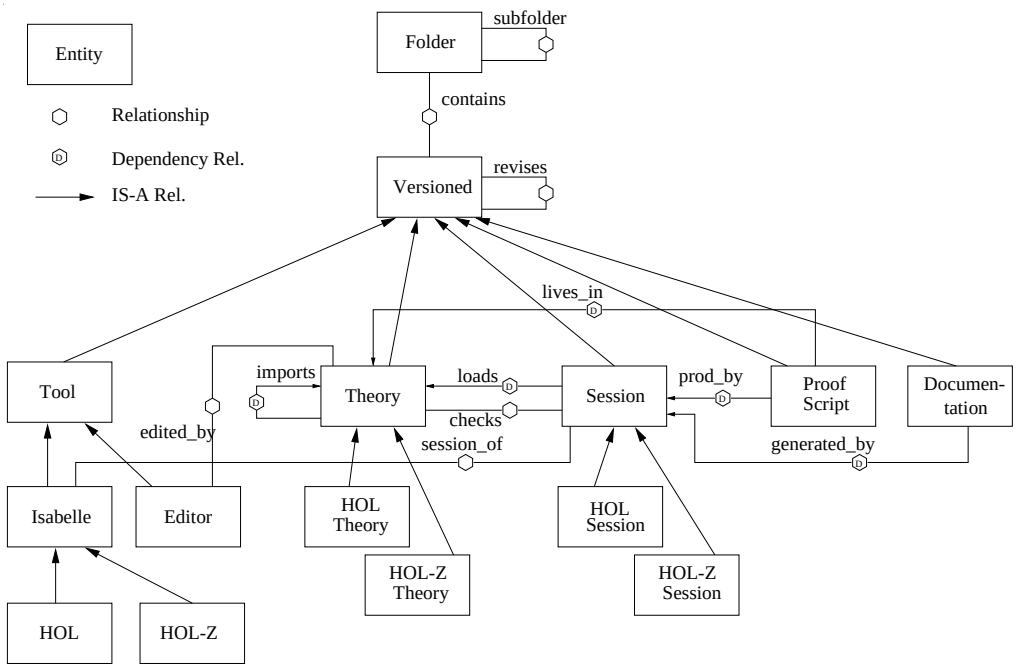


Fig. 3. Z-Workbench Data Model

Tools are either those based on Isabelle, such as HOL or the Z-encoding HOL-Z, or a text editor. By modelling tools as versioned objects, the Workbench is able to maintain and invoke different versions of the same tool. Tool revisions are used to represent new versions of the tool which are incompatible to existing developments; this is in particular relevant for Isabelle (and hence HOL-Z), where old tactical scripts will quite often not run on newer versions of Isabelle.

Theories represent specifications, type and datatype definitions and so forth. They come in subtypes corresponding to the different flavours of Isabelle; thus, there are HOL-Theories and Z-Theories, read by Isabelle/HOL and HOL-Z, respectively. Theories are edited externally (i.e. not within Isabelle itself). Theories are *checked*, if they have been successfully read by Isabelle (or HOL-Z), and become unchecked if they are subsequently edited. Theories in Isabelle are structured hierarchically, and hence come with an *import* relationship. Theories are *edited_by* the text editor.

Sessions represent the persistent state of a session with Isabelle or HOL-Z. Essentially this means that the user may save a session, and resume work later. When a session is resumed, the proof work being done up to that point is reconstructed. To maintain consistency of sessions, the distinction between checked and unchecked theories is important, since each session is reconstructed with the last checked version of the theory (which may be different from the last version of the theory, if it is unchecked). Every session is related (by the *is_session_of* link) to one specific HOL tool, which it runs on. Further, every session may load several theories (as indicated by the *loads* link), and if it successfully loads them makes them into checked theories (as indicated by the *checked* link).

Proof scripts are self-contained tactical Isabelle scripts. Since there is a multitude of objects and tactical operations within Isabelle making up these tactical scripts, modelling all of these in detail would be impractical, and we just treat them abstractly as a tactical script represented textually. Every proof script lives in the context of a theory, given by the *lives_in* link, and is produced by one Isabelle session given by the *prod_by* link.

Finally, documentation can be generated out of a session having loaded several theories. These documentations can be subdivided into several modes (concerning theories, lists of theorems, proofs) and formats (e.g. HTML as done presently by Isabelle, LaTeX or RTF). The situation for generated documentation is analogous to generated *program code* (in the form of Standard ML) that can be generated from executable specifications. The program generator used for Z is currently under development.

Figure 3 is actually an abstraction from the more complicated modelling used in our integration. In particular, there are different types of sessions and proof scripts just as there are different kinds of theories and Isabelle tools; and the relationships between theories, sessions and proof scripts are on the level of these subtypes. There is however just one type of program code (*viz* Standard ML).

The Workbench uses Haskell classes to structure the code and to generalise the API's. The operations and events of relevance to the class of *versioned objects*, one of the central concepts of the repository, is for example outlined by the following class declaration:

```
class (RMIdObjectC vo) => RMVersionedObjectC vo where
   revise          :: vo -> IO vo
   getRevisions    :: vo -> IO [vo]
   getPredecessors :: vo -> IO [vo]
   revised         :: vo -> EV vo
```

The class provides (among others of course) a `revise` operation for creating new revisions, two computations for traversing the version tree (`getRevisions`, `getPredecessors`) together with a `revised` event that occurs whenever a versioned object has been revised. This class is then instantiated on need with theories, sessions and proof scripts.

5.2 Control Integration

All the tools of the Z-Workbench happen to be interactive Unix tools. Even HOL-Z can, although it comes with a graphical user interface, be squeezed into this category, since it is started and controlled from within a ML session.

The Workbench provides a utility called `Haskell-Expect` (inspired by `expect` [19]) for integrating such interactive Unix tools. It runs the Unix tool in the background, and lets the Workbench take over communication with the tool by simulating the user dialogue. The encapsulation of HOL-Z comes quite straightforward in this setting, as we shall demonstrate by the following computation that starts up HOL-Z loaded with a given session and theory:

```
startHolZ session theory = do {
  hz <- newExpect "holz" [];
  sync (match hz "^- ");
  sendCmd hz (load session theory);
  interactor (finished hz >>> do{sendCmd hz "quit();\n"; stop});
  return hz
  } where load s t = "load " ++ show s ++ " " ++ t ++ ";\n"
```

An `Expect` tool is created first that starts up HOL-Z as a ML session. We then wait until the ML interpreter responds with a prompt, as specified by the `match` event and responds by sending a command (`sendCmd`) that will start up WIN-Z and `load` the given `session` and `theory`. The command is forwarded in the form of a string as specified by the function `load`. An interactor is finally spawned off to catch the events that occur when WIN-Z finally returns control to the ML interpreter. The interactor responds by quitting the session with ML before it terminates itself by calling `stop`.

The Workbench is based on a higher order composable approach to event handling, where events are first class values. Base events can be combined into composite events using the guarded choice operator (`e1 +> e2`), and additional reactive behaviour can be glued onto existing events using the event-action combinator (`e >>> a`). The event that occurs when the session with HOL-Z is over, consists for example of two base events:

```
finished hz = (match hz "^- ") +> (match hz "uncaught.*\n")
```

The first event occurs when the session has terminated normally, i.e. `load` has finished and the ML interpreter has generated a prompt and awaits "user input". The second event should actually never occur, since it is generated when the session with HOL-Z has ended abruptly with an uncaught exception.

The Workbench can furthermore communicate with HOL-Z running as a server, in order to request services of HOL-Z or to inform HOL-Z about some external event of relevance to the session (new theorems etc.). The way this is achieved has already been demonstrated: `sendCmd` is used to submit commands to the tool and `match` for looking for specific response patterns.

5.3 Presentation Integration

The need for presentation integration within the Z-Workbench is quite restricted since all tools come with a graphical user interface. What remains is to develop the version and configuration graphs of the system, and provide the user with additional menus and user dialogues for calling and customising the services of the integrated Workbench.

When the user requests the system to open up a new view such as a version graph, an initial view is built first, i.e. the repository is traversed and the appropriate visualisation commands are passed to daVinci. A bunch of interactors are then associated with the graph to maintain consistency between the view and the underlying repository. An interactor for monitoring a single version looks like:

```
monitor g vo = interactor (
   revised vo >>>= (\rev -> do
       showRevision g vo rev
       monitor g rev
       redrawGraph g
       ))
```

The interactor reacts to the `revised` event by showing the new revision link within the current version graph. It then spawns off a monitor for the new revision `rev` and redraws the graph. It actually is as simple as this, although the interactor is in reality set up to listen to a couple of events more.

Generating views, and maintaining the consistency of views, is one issue, tool invocation is another. The services of the Workbench are invoked by using pull-down menus that are associated with the nodes within a view. For example, the version graph has a single interactor that listens to node selection events and menu invocation events:

```
controller g m o = interactor (
      nodeSelected g >>>= \o' -> become (controller g m o')
  +>  invoked m      >>>= \f -> f o
  )
```

The controller takes as parameter a handle to the current graph `g`, the application menu `m` and the selected object `o`. It responds to a node selection event by becoming an interactor that will apply commands to the new selection `o'` rather than the old one.

The second guard is concerned with handling menu invocations. The Workbench is higher order meaning that menus and menu items are functors that may return some value of interest to the application when the menu is invoked. In the case above, the application menu is set up to return the computation that defines the behaviour associated with the menu item. This computation is then applied to the object currently selected.

The missing link is the following piece of code that ensures that a new session with HOL-Z is started whenever the user clicks the `"Revise"` button associated with a HOL-Z session:

```
button [text "Revise", command (return newHolZSession)]

newHolZSession vo = do
    vo' <- revise vo
    (session,theory) <- exportSession vo'
    startHolZ session theory
```

A new revision is created first and all files of relevance to the HOL-Z session are then exported to the file system. A new development using HOL-Z is then started in the context of the given session and theory.

5.4 Experiences gained from the Prototype

The Workbench provides, being based on Haskell extended with a higher order approach to concurrency, a high level of abstraction and expressive power that comes very close to the one of constructive formal specifications. There are several key features in achieving this.

First of all we benefit from Haskell's expressive power by being a higher order and strongly typed language. Classes are furthermore used to structure the code and to standardise the interfaces to the system. The sequential behaviour of the system is expressed in terms of IO computations that have

a theoretical foundation in the monadic approach to IO. A more practical consequence is that we have an extensible language framework that can be enhanced with new computational paradigms on need. The innovative part of the system however is the approach to event handling that treats all events of the system, whether they are user interactions, data base change notifications, operating system signals or individual tool events in terms of first class composable values that entirely hide the source of the event. Tool integration can therefore be expressed in a style that comes close to the one used, had we just made a formal specification using process algebras. The difference though is that Haskell is executable - and highly efficiently compiled as well.

6 Proving in the Birthday Book

In this section we will use the classical birthday book example to demonstrate the look-and-feel for Win-Z and its interaction with the surrounding Z-Workbench. This will be done with a proof of a tiny property of *AddBirthday*, namely that the set of names known to the system will be different with the addition of a given new name.

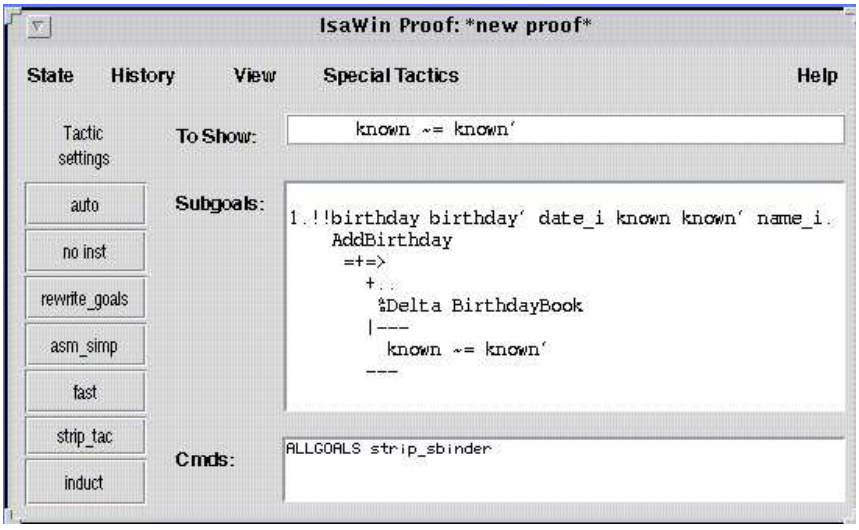


Fig. 4. Making implicit variables explicit

First of all, we defined a Z-theory *BB.zthy*, which is a Z paragraph containing the Z schemas defining the Birthday Book. The leading fragment of this paragraph has already been shown in section 4.

We state our desired property as follows:

```
|-- AddBirthday =+=> +.. %Delta BirthdayBook |--- known ~= known'
```

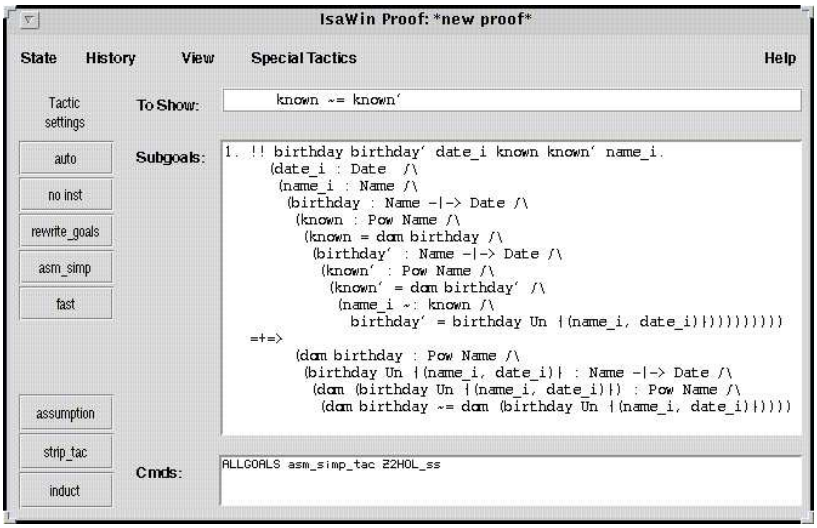


Fig. 5. After Schema Expansion and Simplification

We start with a Z-session where *BB.zthy* has already been loaded, and enter this goal with Win-Z. Figure 4 shows a screenshot where the goal above has been refined with one tactical step, that makes all the implicit free variables (the parameters of the schema), explicit by universally quantified variables.

We proceed now by expanding the schema definitions for *AddBirthday* and *BirthdayBook*. Further, elementary simplification leads to the following proof-state as shown in Figure 5. At this level, our fictive user interrupts the development, and wishes to save the session. Figure 8 shows this possibility, where the Workbench saves the HOL-Z session.

Another user goes to the version graph of the Z-session shown in section 3. The Workbench has meanwhile reacted to the change notifications of the repository manager, and has updated the version graph accordingly to show the new revision. The user just clicks on this generated version, and the Z-Workbench will start Win-Z in exactly the state saved by our first user.

After applying some minor tactics the goal is divided now in small subgoals as shown in Figure 6. The last three subgoals are proven automatically by using Isabelle's decision procedures for sets. The first one involves some knowledge about domains. One has therefore to use the simplifier sets related to the Z Mathematical toolkit. Figure 7 shows the result of this simplification.

At this level of the development the user needs a lemma about subsets and union, stating that if B is not included in A, then A is not equal to the union of A and B:

```
not(B <= A) ==> A ~= A Un B
```

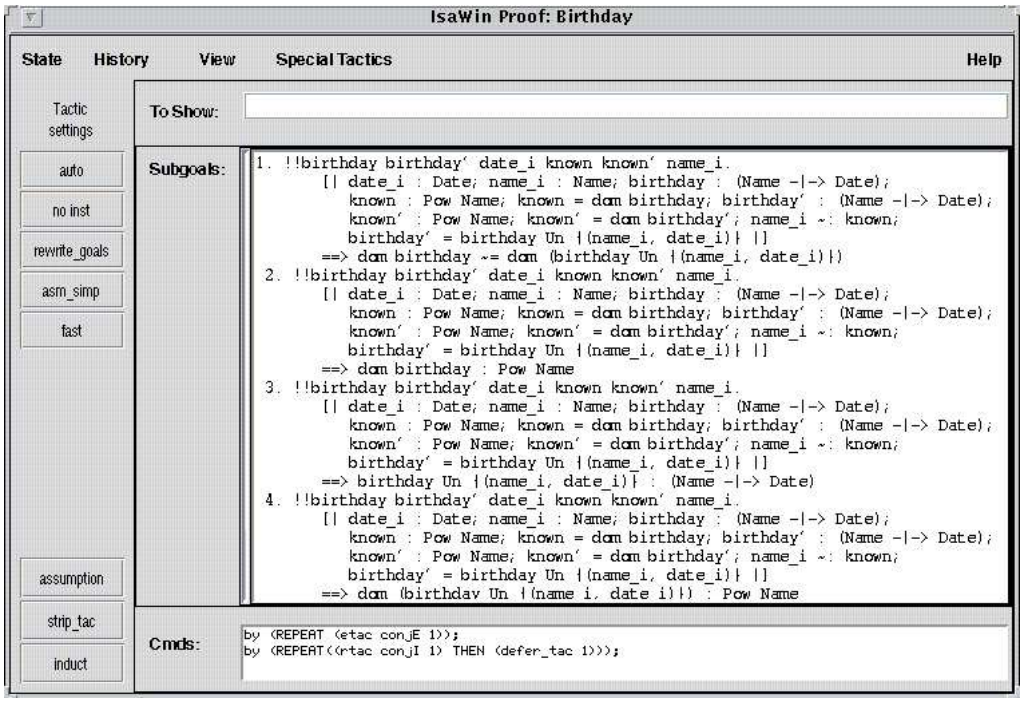


Fig. 6. Breaking up the Conjunction of the Conclusion

Now, using this lemma produces the following proof state as shown in Figure 8, Isabelle's decision procedure will do the rest. Again, our new user can save the newly generated session and extract from it the proof-script underlying the demonstrated proof-development.

7 Conclusion

We have seen an instantiation of the UniForM-Workbench for Z based on a specific LCF-style prover environment called HOL-Z[16]. The resulting prototype gives an impression of the power of the modular, generic and functional technology employed.

The modular aspect allows that the components of the Workbench can be developed by different groups of developers and users. It is perfectly possible to use the pure encoding HOL-Z or its combination with Win-Z in itself. It is possible to use the GlueWare of the Workbench for completely different tools, not necessarily connected to Formal Methods at all.[3] But it is the combination of these three that scales up a correctness-oriented, but notoriously difficult to use prover-environment to an formal methods environment, that enables versioning (hence reproducibility) and overall semantic integrity (this piece

[3] This has been done for the Hugs-Workbench [10,14].

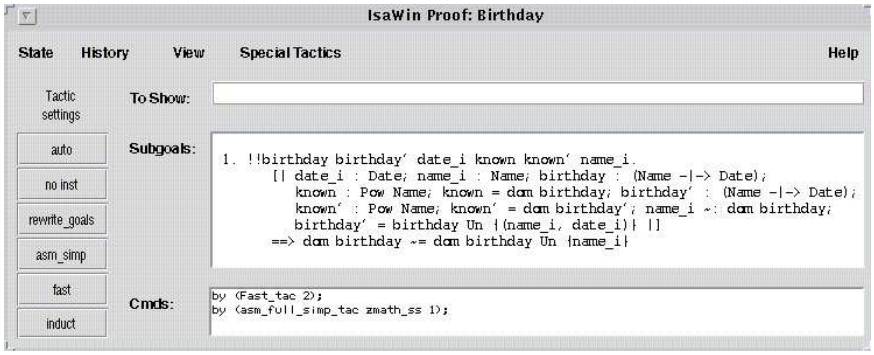


Fig. 7. Simplification with the Mathematical Toolkit

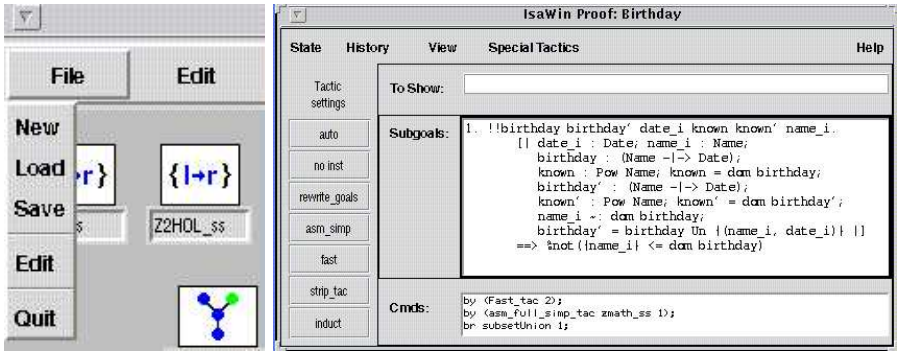


Fig. 8. HOL-Z Session (left), Proof State after Lemma Introduction (right)

of code or this generated documentation belongs to this specific state of a Z-theory) even in a multi-user setting. Thus, it is possible to reconfigure the Workbench with little effort with new external tools on the one hand and new semantic embeddings like [30].

The pervasive generic aspect of our technology stresses this toolkit-character of the Workbench even more. The tools themselves are built by many components, for example the generic graphical user-interface GenGUI, that are designed to be independent from each other and that can be *instantiated* for there particular application: In principle, GenGUI could be used for a completely different, sml-based LCF-prover such as the HOL/HOL System.

We are very happy with our decision to use strongly-typed, state-of-the-art functional programming languages for both the Workbench and Win-Z. The combination of a purely functional language extended with a higher order model to concurrency, allows the integration to be done at a very high level of abstraction. Working with tool support for formal methods this has become a crucial aspect, since the technology allows us to experiment with new ideas without being slowed down by a large implementation and maintenance overhead.

7.1 Related Work

A short discussion of related *general* tool-integration techniques has already been done in section 2. In this section, we will therefore concentrate on integration techniques used for formal method tools.

Roughly, we divide existing approaches into two categories; first, ad-hoc approaches, which we refer as "Glue&Pray", and secondly more refined ones labelled "Glue&Play". This division is clearly debatable, and the two categories are rather the extreme poles in a continuum, and a tool belonging to the former, admittedly polemically labelled class is not necessarily a bad tool. But we believe that formal method tools as particularly complicated software products suffer to a larger extent from poor implementation technology than other software, be it with respect to the reliability of their results or be it with respect to maintainability necessary to cope with the rapidly evolving field. A case in point being PVS, which although as a prover is as powerful as Isabelle, and more widely used, suffers from an untyped implementation and interface language; or Tcl-based interfaces to LCF-provers like TkHOL or XIsabelle, which rarely survive the rapidly changing revisions of the underlying prover.

Examples for the "Glue&Play" approach are the Cogito [2] system or KIV [25]. As tools, these are clearly superior to our Workbench in terms of availability, user-friendliness and stability; but again, we believe they suffer from a system architecture which does not have a theorem prover as powerful and versatile as Isabelle at its heart, and which does not allow them to keep up with changes as good as our Workbench, the modular design of which allows easy exchange of parts which are outdated or superseded by newer developments.

7.2 Future Work

With respect to the Workbench for Z, more tools supporting different documentation formats like LaTeX, RTF or HTML on the one hand and animators on the other would be desirable extensions to the existing prototype. The Win-Z component needs the integration of the code-generator currently under development and more GUI-specific tactical support.

Another line of extension is the integration of other Z-Tools (like animators or test-case generation tools) *not* based on the Isabelle-Kernel. This kind of integration requires, due to the variety of syntactical constructs of Z, very carefully constructed translators, which often represent a reliability problem to the integration in practice. Paradoxically, it seems easier for us to integrate other Isabelle-Encodings (like our CSP-Encoding) plus formally proven sound combination formalisms for Z and CSP (such as [29]) into our logical core engine, turning the Workbench into a technical framework for combined reasoning over them.

In our actual prototype, the granularity of data to be exchanged between different sessions is still rather coarse. For instance, the Workbench has no

access to the components of proof scripts. A refinement of the data model could allow individual proofs to be transferred from one session to another. This could be substantially enhanced if general merge techniques on proof scripts (as developed for a specialised logic inside KIV) were available. All these techniques could be combined with an active *change propagation* mode of the Workbench — i.e. the Workbench starts the logical engine in a batch mode, passes it a textually modified Z-theory, causes a re-evaluation of depending proof scripts with the aim to recertify as much as possible, and to save the resulting state of the logical engine in a session that is ready for further interactive development.

References

1. R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, October 1986.
2. A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: A Methodology and System for Formal Development. *International Journal of Software Engineering*, 4(3), 1995.
3. The H-PCTE Crew. H-PCTE vs. PCTE, version 2.8. Technical report, Universität Siegen, June 1996.
4. ECMA. Reference Model for Frameworks of Software Engineering Environments. Technical Report TR/55, European Computer Manufacturers Association, June 1993.
5. ECMA. *Portable Common Tool Environment (PCTE) — Abstract Specification.* European Computer Manufacturers Association, 3 edition, December 1994. Standard ECMA-149.
6. Open Software Foundation. *OSF/Motif Series.* Prentice Hall, 1992.
7. M. Fröhlich and M. Werner. daVinci V2.0.3 Online Documentation. Universität Bremen, `http://www.informatik.uni-bremen.de/~davinci`, 1997.
8. A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, December 1985.
9. P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell - a non strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):1–162, 1992.
10. E. W. Karlsen. Integrating Interactive Tools using Concurrent Haskell and Synchronous Events. In *CLaPF'97: 2nd Latin-American Conference on Functional Programming*, September 1997.
11. E. W. Karlsen. The UniForM Concurrency Toolkit and its Extensions to Concurrent Haskell. In *GWFP'97: Glasgow Workshop on Functional Programming*, September 1997.
12. E. W. Karlsen. The UniForM User Interaction Manager. Technical report, FB 3, Universität Bremen, 1998.
13. E. W. Karlsen. The UniForM WorkBench - a Higher Order Tool Integration Framework. Technical report, FB 3, Universität Bremen, 1998.
14. E. W. Karlsen and S. Westmeier. Using Concurrent Haskell to Develop Views over an Active Repository. In *IFL'97: Implementation of Functional Languages*, September 1997.

15. Kolyang, C. Lüth, T. Meier, and B. Wolff. TAS and IsaWin: Generic interfaces for transformational program development and theorem proving. In M. Bidoit and M. Dauchet, editors, *TAPSOFT 97': Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 855–859, Lille, France, April 1997. Lecture Notes in Computer Science.
16. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, number 1125 in Lecture Notes in Computer Science, pages 283 –298. Springer Verlag, 1996.
17. G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
18. M. Lacroix and M. Vanhoedenaghe, editors. *Tool Integration in an Open Environment*, volume 387 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.
19. D. Libes. expect: Scripts for Controlling Interactive Processes. In *Computing Systems, Vol 4, No. 2*, Spring 1991.
20. Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer, 1996.
21. J. Nicholls and The members of the Z Standards Panel. Z notation, September 1995. Available at `http://www.comlab.ox.ac.uk/oucl/groups/zstandards/`.
22. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
23. L. C. Paulson. *Isabelle - A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer Verlag, 1994.
24. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96), Florida*, 1996.
25. W. Reif, G. Schellhorn, and K. Stenzel. Proving system correctness with KIV. In M. Bidoit and M. Dauchet, editors, *TAPSOFT 97': Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 859– 862, Lille, France, April 1997. Lecture Notes in Computer Science.
26. J. H. Reppy. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.
27. T. Reps. *Generating Language Based Environments*. PhD Thesis, Cornell University. MIT Press, 1983.
28. D. Schefström and G. van den Broek. *Tool Integration*. Wiley, 1993.
29. G. Smith. A semantic integration of object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 62– 81. Springer Verlag, 1997.
30. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of the FME '97 — Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 318– 337. Springer Verlag, 1997.
31. S. Westmeier. Verwaltung versionierter persistenter Objekte in der UniForM WorkBench (UniForM OMS Toolkit). Master thesis, FB 3, Universität Bremen, January 1998.

Functional Design and Implementation of Graphical User Interfaces for Theorem Provers

C. Lüth †

Bremen Institute for Safe Systems, TZI, FB 3, Universität Bremen
cxl@informatik.uni-bremen.de

B. Wolff †

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
bu@informatik.uni-freiburg.de

Abstract

The design of theorem provers, especially in the LCF-prover family, has strongly profited from functional programming. This paper attempts to develop a metaphor suited to visualize the LCF-style prover design, and a methodology for the implementation of graphical user interfaces for these provers and encapsulations of formal methods. In this problem domain, particular attention has to be paid to the need to construct a variety of objects, keep track of their interdependencies and provide support for their reconstruction as a consequence of changes. We present a prototypical implementation of a *generic* and *open* interface system architecture, and show how it can be instantiated to an interface for Isabelle, called `IsaWin`, as well as to a tailored tool for transformational program development, called `TAS`.

1 Introduction

The story of graphical user interfaces (GUI's) for theorem provers and formal method tools as a whole is not exactly a success story so far. There is widespread scepticism (Merriam & Harrison, 1997) that GUI's adopting techniques from the field of human computer interaction (HCI) can increase productivity to a similar extent as they did, say, in the area of office applications. GUI's for widely used tools like PVS, FDR or Isabelle are still dominated by text-based subwindows barely hiding the roots of the underlying tool. We believe this has predominantly historic reasons.

The history of functional languages, in particular ML, has been deeply intertwined with the genesis of the LCF theorem prover family, for which it was originally developed as a meta language. The essential idea in LCF-style provers (like HOL (Gordon & Melham, 1993) or Isabelle (Paulson, 1994)) is to encapsulate the

† This work has been partially supported by the German Ministry for Education and Research (BMBF) as part of the project UniForM under grant No. FKZ 01 IS 521 B2.

logical engine in an abstract datatype, the objects of which can only be constructed by operations implementing the rules of the underlying logic. This yields the basis for an open system design allowing user-programmed extensions in a logically sound way. The flexibility, generality and expressiveness of LCF-style provers makes them symbolic programming environments, into which other languages can be logically embedded, e.g. Haskell (Regensburger, 1994), Java (Nipkow & von Oheimb, 1998), Z (Bowen & Gordon, 1994; Kolyang, Santen & Wolff, 1996b) or CSP (Tej & Wolff, 1997). Together with appropriate, customised proof support and a graphical user interface which hides the details of the embedding, this leads to an implementation technology for formal method tools which we call *encapsulation*.

Thus, while the LCF-design has its undoubted advantages, these systems have inherited a very restricted model of user interaction based on a command line interface, and not profited as much as possible from recent advances in interface design (Shneiderman, 1998; Thimbleby, 1990; Dix *et al.*, 1998). As Bornat and Sufrin (1998) put it, this problem cannot be overcome by "bolting a bit of Tcl/Tk onto a text-command-driven theorem prover in an afternoon's work".

Our contributions towards filling the gap between classical command-line interaction and more modern concepts of graphical user interaction are firstly to develop a *new metaphor* for the visualization of LCF-style provers. The metaphor serves as a vehicle to make the data structure of the prover accessible to *pervasive direct manipulation*. Secondly, the metaphor develops an abstract notion of user interaction, and is compatible with the need for their *systematic replay*. Replaying proofs is a central issue in theorem proving. Thirdly, the metaphor is implemented in a *generic system architecture*, based on the structuring mechanisms of Standard ML, using a functional encapsulation of Tcl/Tk and the theorem prover Isabelle. Besides a graphical user interface for a theorem prover, this gives an encapsulation technique for formal methods.

This paper is organized as follows: we will first discuss issues relating to the conceptual design and the metaphor. We will then turn to the architecture of the system, introducing a data model and a process model. This will be followed by a section expanding on some aspects of the implementation, and a section introducing a different instantiation of the generic system. We close with an evaluation of the proposed work, and a comparison to related work.

2 Conceptual Design Issues

Direct manipulation, a term attributed to Shneiderman (1982), is a widely known technique in HCI and graphical user interface design (Shneiderman, 1998; Thimbleby, 1990; Dix *et al.*, 1998), characterized by *continuous representation* of the objects and actions of interest with a *meaningful visual metaphor* and *incremental, reversible, syntax-free operations* with *rapid feedback* on all actions. In this section, we will introduce the notepad metaphor, serving as a vehicle to make the internal objects of a theorem prover accessible for direct manipulation.

2.1 The Notepad Metaphor

As a motivating example, consider the way we do everyday mathematics and calculations: one typically uses a piece of paper or a blackboard to write down intermediate results, calculations or lemmas, but overall in an unstructured way, adding a column of numbers in one part of the pad, while doing a multiplication in the lower corner and a difficult diagram-chase in the middle.

A simple instance of this is a small notepad on which we can write down numbers and arbitrary text. The operations would either be arithmetic (e.g. add two numbers), or textual (write some new text). Technically, the notepad could be visualized as a window in which the user can manipulate *objects*, represented as *icons*, by *drag&drop*. The world of objects on our notepad is structured by an inherent notion of *typing* (here, numbers and texts). This typing is crucial when considering the operations, because an operation taking numbers as arguments is different from an operation taking texts as arguments. The operations are applied by drag&drop, so if we drop a number onto a number, we may want to add them up, whereas if we drop a text onto a text, we may want to concatenate them. Figure 1 illustrates our example: On the left, we can see objects representing numbers 2, 4 and 5, and two pieces of text. If we move the number 2 on the number 4 (second from the left), they are added up, and we obtain a new object: the number 6 appears (third from the left).

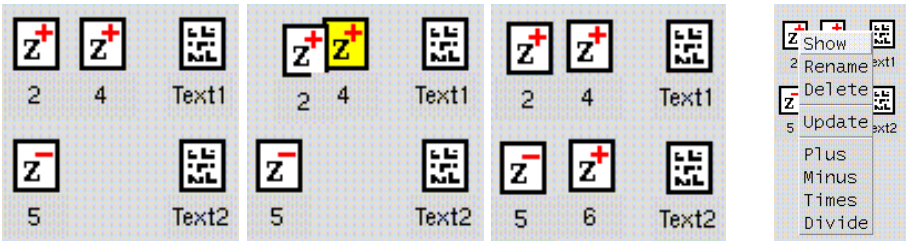


Fig. 1. Introducing the notepad metaphor and manipulation by drag&drop.

This shows the first main principle of a functional GUI: objects represent values, and hence the interaction of objects produces new objects, rather than changing existing ones. Dropping an object onto another corresponds to function application. These functions are called *binary operations*; passing several objects to a binary operation is possible by grouping objects via multiple selections. Additionally, *unary operations* may be defined for each object type, which take exactly one argument, and are invoked via a pop-menu (see Fig. 1 on the right, where the standard operations *Show*, *Rename* and *Delete* can be seen.)

In practice, the simple typing discipline has proven insufficient — e.g. instead of adding two numbers, we might as well want to subtract, multiply or divide them. To this end, we introduce the concept of a *mode* that an object may have. In our example, each object of type natural number has four modes: *plus*, *minus*, *times* and *divide*. The function applied by drag&drop is determined by the mode of the

object being dropped onto: dropping a number onto another number in *times* mode multiplies the two numbers.

At first sight the modes seem to contradict our principle of a functional GUI, since they allow a form of state. However, the modes only serve to disambiguate or simplify user interaction. This context information may help the system to provide additional parameters that had to be provided explicitly otherwise; we do not allow side effects when applying operations. As a general rule of interface design(Thimbleby, 1990), modes should not be hidden, so the icon of an object is determined by both the mode and the type of the object. This way, the action which will take place is always transparent to the user. In Fig. 1, the modes of the numbers are shown by an additional sign on the upper right corner of the symbol. The user can change the mode of an object by a pop-up menu (Fig. 1 on the right).

2.2 Undo, Persistence and Replay

According to the main principle of a functional GUI, function application can not change the arguments of the function. This allows an easy implementation of *undo*: we just delete the object created by applying the function. Moreover, the functional approach makes it easy to reconstruct an object. By recording the operations which have been applied to construct an object, we can reconstruct the object later by *replaying* the operations. This is needed to implement a persistent state, and to deal with external objects.

By *persistent state*, we mean that we want to be able to save the current state at a given moment, exit the system, and later restart the system in the same state where we left it. Under the assumption that only operations, but not objects, can be saved externally, persistence is achieved by recording for every object how it was constructed, and reconstructing the object by replaying the operations upon restart.

As an example of *external objects*, suppose that the texts on our notepad are given as post-it notes stuck to the pad. Their value is the text written on them. We can concatenate two texts, but if we then write something different on one of the notes, the value of the concatenation should change accordingly. In our example, suppose the text objects (like *Text1* and *Text2* in Fig. 1) are read from external files. By dragging *Text1* on *Text2*, we create a new object, say *Text3*. If now *Text1* is *reloaded*, the value of the object may change, and consequently the value of *Text3* should change as well. We say *Text3* is *outdated*, which is is indicated by shading the icon of *Text3*. Outdated objects can be updated again by replaying its *history*. Updating is invoked via the pop-up menu (as in Fig. 1).

Replay is very important in the theorem proving context, because most theorem provers read declarations and definitions from external files, which are frequently modified by the user, and have to be reloaded. Yet, systematic replay is to our knowledge never supported on the level of the GUI.

2.3 Construction Objects

For certain objects, manipulation without regard to their internal structure, and hence their history, is insufficient. For instance, we want to admit editing of text objects in our simple example. The history of such editing operations will then consist of a protocol of operations like `delete "javelin" at position 1.12` or `insert "spear" at position 1.12`. Navigating forward and backward in this history corresponds to undoing and replay.

These objects will be called *construction objects.* They can be opened by double-clicking, which leads to the creation of two new windows, namely the *construction area* and the *history navigation window* (see Fig.2). Both windows have a *focus*, i.e. a mark on some position in the text, and some position in the history, respectively. The history focus controls the content of the construction area.

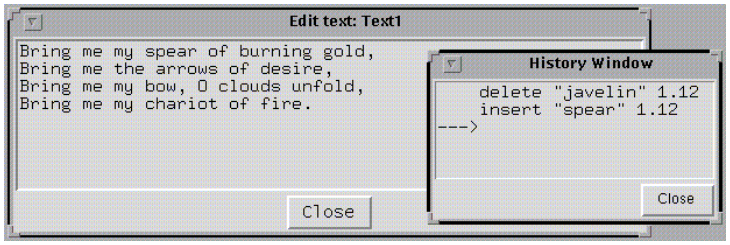


Fig. 2. Construction area and history navigation window.

When closing a construction object, the current value of the construction object is bound to the object which was opened (i.e. to the icon that was double-clicked). The reason for this behaviour is that the notepad would hopelessly clutter up if a new object was created for each step in the history. Objects depending on a closed construction object are marked as outdated.

Note that the replay of the history may fail. For example, the semantics of the `delete "javelin"` operation may be undefined if no `"javelin"` occurs in the text as a consequence of an external change and a reload of the object.

There are more forms of interaction between the notepad window and the construction area or the history window. Objects on the notepad may be dragged on the focus set in the object value field, e.g. replacing the text selected in the focus. *Vice versa*, the selected text of the focus may be extracted and form an object on the notepad.

2.4 `IsaWin` *— A Functional Graphical User Interface for Isabelle*

We will now explain how theorem proving fits into the concepts described in the previous sections by describing our GUI `IsaWin` for the theorem prover Isabelle (Paulson, 1994). Isabelle is just the example at hand; we expect no principal difficulties in developing analogous interfaces for other LCF-style prover based on SML.

2.4.1 Accommodating Basic Theorem Proving into the Notepad

The object types of `IsaWin` are a subset of those provided by Isabelle, as shown on the left of Fig. 3: in the first row two theorems and a theory, in the second row, two different types of rule sets, called simplifier sets and classical rule sets in Isabelle parlance, and an ongoing proof, or more precisely the proof script which we will identify with a proof throughout this paper. Theorems have four modes, they can be introduction rules (as in Fig. 3), elimination rules, destruction rules and equations (which are not shown).

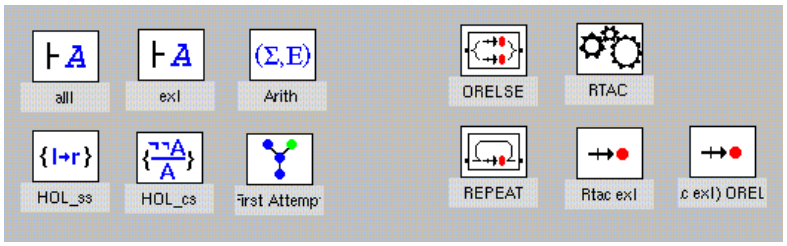


Fig. 3. The Objects of `IsaWin`: to the left, basic objects; to the right, tactical objects

The binary operations in this instance include the forward resolution of two theorems: unifying the conclusion of one theorem with the hypothesis of another one. This corresponds to dropping a theorem object onto another theorem. Note that this may not succeed — the operation is partial. E.g. if the theorem `add_0` : $0 + c = c$ is dropped onto the theorem `sym` : $s = t \Rightarrow t = s$, a new theorem $t = 0 + t$ is produced by forward resolution; but *vice versa* the operation fails. Simplifier sets are sets of rewriting rules. If a theorem is dropped on a simplifier set, a new simplifier set is produced with the theorem included. Classical logics support another type of rule sets. These classical rule sets come in two modes, safe and unsafe, since theorems can be added to a classical rule set in two ways (this distinction makes a difference for a decision procedure of Isabelle, the so-called classical reasoner). If a theorem is dropped on a classical rule set, depending on the mode of the rule set, it is either added as a safe rule or as an unsafe rule.

Reloading an external theory file results in outdating all depending objects, like included theories or theorems depending on them.

2.4.2 Accommodating Tactical Programming into the Notepad

Contrary to a common prejudice against GUI's for theorem provers, it is quite straightforward to embed basic tactic script construction into the notepad metaphor. First, we provide object types corresponding to certain Isabelle types, *tac_op* (for tactical operations), *tactics* and *tacticals*. Second, we provide objects of type *tac_op* corresponding to backward resolution or simplification, basic tactics like proof by assumption as objects of type *tactic*, and the usual connectives *REPEAT*, *THEN* and *ORELSE* on tactics as objects of type *tactical*. Third, we set up the binary

operations by embedding Isabelle's tactical algebra into our world of object types, objects and binary operations. We are now able to construct, for example, an object corresponding to the Isabelle tactic `REPEAT o ((rtac exI) ORELSE' (rtac allI))` which by repeated backward resolution with the quantifier introduction theorems *exI* and *allI* will eliminate an arbitrary sequence of outermost quantifiers on a subgoal. Fig. 3 shows the constructed tactic in the lower left corner, together with tacticals *ORELSE* and *REPEAT*, the tactical operation *RTAC* and the tactic *Rtac exI*.

2.4.3 Accommodating Backward Proof into Construction Objects.

In LCF-style provers, the main proof method is by *backward proof*: if we want to prove a goal ϕ in this style, a *proof state* is initialized with the formula $\phi \Rightarrow \phi$. With a theorem $A \Rightarrow B \Rightarrow \phi$, the proof state can be refined to $A \Rightarrow B \Rightarrow \phi$ by forward resolution. The premises left from the rightmost implication, hence A and B, are called *subgoals*. If as a consequence of further proof steps no subgoals are left, the proof state can be converted into the theorem ϕ.

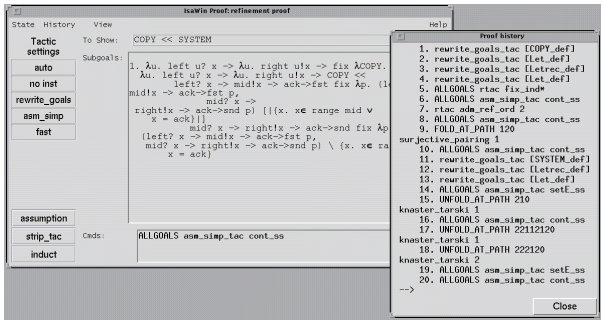


Fig. 4. `IsaWin`'s construction area

It is convenient to declare backward proofs as construction objects and the proof steps performed by tactical operations as their history. When dragging objects from the notepad window to the construction area, the GUI will perform tactical operations, depending on the mode of the dragged object, the settings of the buttons and and the focus set by the user. If a theorem `lemma1` has the mode *introduction rule*, and the focus is set to the second subgoal, the drag&drop gesture will trigger the Isabelle operations `by(rtac lemma1 2)`. Further, dragging a simplifier set down into the construction area will cause Isabelle's rewriting machine to execute the rewrites in it. If there are no subgoals left to be proven, the construction area can be closed to yield a theorem object on the notepad. Figure 4 shows the construction area of the `IsaWin` interface. The most prominent part is the display of the subgoals, and the main goal to be proven.

3 System Design Issues

As mentioned in the introduction, we want to provide a family of user interfaces for different *applications*, let it be for different theorem provers or different tools built on them. Hence, our system architecture has to be *generic*. It depends on an abstract characterisation of the application; this parameter is discussed in Sect. 3.1 and leads to the data model described in Sect. 3.2. This view is complemented by the process model in Sect. 3.3, where the communication of the different components is presented.

3.1 An Abstract View of Functional User Interfaces

At an abstract level, we consider the theorem prover, or the encoded formal method to be an *application* which is a structure with the following characteristics:

- It has *objects*, each of which has *type*. The type determines the possible *modes*, and both determine which operations are applicable to this object, and both were indicated by the object's *icon*.
- There are partial *operations* which can be applied to objects, namely *unary operations* which take exactly one argument, and *binary operations* which take two arguments. Unary operations are selected from the pop-up menu bound to each object, whereas binary operations are triggered by drag&drop.

Thus, an application has a set S of types, a set Ω of operations which have certain arity (i.e. an operation $\omega \in \Omega$ takes exactly one or two arguments of specific types), a set A_s of the possible values of objects of type s, and a way to apply operations ω from Ω to elements of A_s. In other words, an application is given by a signature $\Sigma = (S, \Omega)$, and a partial Σ-algebra A. This separation of the syntax of the application (given by a signature) from its semantics (given by an algebra) is essential in being able to handle replay, as we will see below.

The modes — and similarly, the settings in the construction area — only serve to disambiguate which operation ω is going to be applied. Once the operation has been selected, its evaluation is independent of modes, settings or any other user input.

Technically, we can denote this characterisation by an SML signature `APPL_SIG`, making the generic interface an SML functor which when instantiated with an application yields a graphical user interface for that application; we will elaborate on this in section 4.2 below.

3.2 The Data Model

The metaphor developed in the previous section was based on the representation of values by icons on a notepad, and application of operations on these objects. Representation on the notepad corresponds to *naming* an object— the object can be referred to, and operations can be applied to it.

As an application is given by a signature Σ and a partial Σ-algebra A, the history of an object is given by composition of operations, or in other words by a *term* t

from the term algebra $T_\Sigma(X)$, built over a set X of variables (where the rôle of the variables is taken by the external identifiers). Then, given a mapping of the variables to values in A_s (i.e. a way to evaluate external objects), every term evaluates to an element of A_s (MacLane & Birkhoff, 1967). So in order to be able to replay the construction of an object, every object is represented internally as a pair (a, t) with $a \in A_s, t \in T_\Sigma(X)_s$, where a is the current optional value of the object (if it exists), and t is the history.

Since objects can be referred to, a single object can be used more then once. The data model has to take into account that kind of sharing, since otherwise replay would become unnecessarily expensive. In proof scripts, this sharing is achieved by binding the theorem to an identifier. In our data model, it is implemented by representing all terms representing the history of the objects in a directed acyclic term graph, representing the global data state of the system.

The vertices of the term graph correspond to the pairs (a, t); every vertex may be associated to an icon on the notepad. The edges correspond to operations. If the value a does not exist, the object is called outdated. Recall that outdating can occur in two ways: an external object is changed (i.e. re-evaluated; for example, a file is being reread into the system), or an operation is applied to a construction object.

The notion of history used here is linear, like Archer, Conway and Schneider's script model (Archer *et al.*, 1984). When we go back in the history, there is a sequence of operations which can be applied by going forward again (the pending operations). If after going back we apply a different operation, these pending operations are lost and cannot be referred to anymore. This is a design decision to make navigating the history easy. With the data model, it would be easy to implement a history which is not a linear script, but a graph (like Vitter's US&R model (1984)), where applying new operations is possible while still pending ones are kept in another branch of the history.

3.3 The Process Model

The components of the architecture comprise the notepad and the construction area which have already been introduced above. Additionally, the application may provide communicating components such as a file selector, or a theorem chooser; these typically serve to import external objects into the system. Construction area and notepad are closely coupled, because they exchange values of objects under construction. The notepad and all other components communicate with each other via a clipboard, and with an external environment, exchanging external representations of the history of objects.

Figure 5 shows the process view of the system. The components $S_1, \ldots, S_N$ are the application-specific components; their communication with the environment is optional (hence the dashed arrows). Except for the environment and the clipboard, each component is associated to a widget or a window visualizing its process state in the GUI (construction area and history navigation have one each for convenience).

Our design goal of pervasive direct manipulation is reflected by the communica-

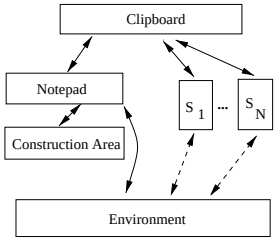


Fig. 5. The Process View of the Architecture Scheme.

tion vertices that connect all components with the clipboard. The arrow pointing into the clipboard represent drag-events (parameterized with the object), while the arrow pointing from the clipboard represent drop-events.

Our design goal of persistence is reflected by the arrows connecting the notepad to the environment. Both of these components have an internal state which we have to be able to save into the environment, and read back from there. The application-specific selector components may have an internal state, and thus may need to communicate with the environment as well.

In Fig. 5, more than one instantiation of the interface can be connected, by sharing the same environment, and by connecting the clipboards. This requires conversion functions between the objects of the different instantiations. This gives us a way to build Formal Software Development Environments as a consequence of the genericity of our architecture. A prototypical implementation of this scheme, centred around tools for the specification language Z, is discussed in (Lüth *et al.*, 1998).

4 Implementation

In this section, we will give an overview of the implementation, briefly touching on all components of the system (Fig. 6) in turn. The system is implemented entirely in Standard ML. The instances discussed throughout the paper are based on the theorem prover Isabelle. Since Isabelle essentially consists of a collection of ML types for objects such as theorems, proofs and rule sets, and ML functions to manipulate these objects, organised into a collection of ML structures and functors, one can conservatively extend Isabelle by writing ML functions, using the abstract datatypes provided by Isabelle, without corrupting the logical core of Isabelle

To implement the graphical user interface, we have developed a functional encapsulation of the interface description and command language Tcl/Tk (Ousterhout, 1994) into Standard ML, called sml_tk (Lüth *et al.*, 1996). This package provides abstract ML datatypes for the Tcl/Tk objects, thus allowing the programmer to use the interface-building library Tk without having to program the control structures of the application in the untyped, interpretative language Tcl.

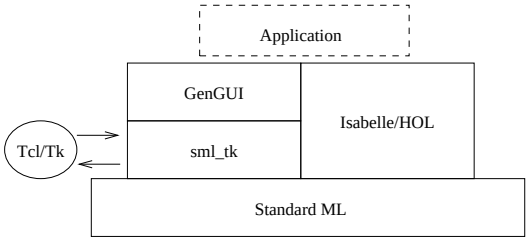


Fig. 6. Module Architecture

4.1 Direct Manipulation of Formulas and Annotation Issues

The problem of representing terms and formulas is ubiquitous in a GUI for a theorem prover. With few exceptions based on a dag-like representation (Kahl, 1998), terms are represented essentially text-based, enriched by mathematical or some graphical notation like square roots or sum signs. In a GUI, there is a potential for novel user interaction such as *query-by-pointing* (clicking on a subterm in order to get information like types) or *prove-by-pointing* (clicking on a subterm to apply a tactic or rewrite) (Bertot & Théry, 1998). Finally, direct manipulation is a straightforward idea enabling the user to drag&drop a subterm within a sum, effecting appropriate applications of associativity and commutativity laws (going back to the system *Theorist*; see also (Bertot, 1997a)) which are, at least in Isabelle, extremely tedious to communicate in command-line style.

In a generic, language independent environment such as Isabelle, a prerequisite of theses interactions is the generation of term annotations that allow to set a focus in the sense of Sect. 2.3, or to *point* in the sense above. In this section, we describe the necessary concepts and their implementation within the Isabelle syntax engine.

First, a mechanism to attach and manage one or more alternative external representations of a syntax to a theory is needed. These *paraphrasings* allow to produce graphical output like $\forall x.P$ instead of the conventional text output `!x.P`. (This mechanism also allows the generation of other documentation formats like LaTeX).

Second, for a smooth transition from Isabelle's text-based ouput to graphical output, we implemented a markup-interpreter as a generic component of sml_tk. It provides a generic parser for an SGML-style notation `<tag> ... </tag>` that binds attributes or ML functions to the subtext marked by the tags; e.g. the graphical output above is obtained from the code `<FONT SYMBOL>\"</FONT>x. P`.

Third, the concept of *annotations* has to be added. Annotations are constant symbols with an external representation that is invisible on the screen and fully transparent to the Isabelle printing macros and printing translations. They are used to generate bindings to specific subtexts. For example, the focus mechanism described above is implemented by surrounding every subterm t with an annotation `<SEL` p`>`t`</SEL>` where p is a representation of the path to the subterm t. The tag `SEL` is bound to a function which given p extracts the subterm t from the proofstate. This annotation has to be transparent to the pretty-printing macros, otherwise e.g. the rewriting from the internal representation `x::y::[]` to the external

representation `[x,y]` will fail. Based on these paths, it is a standard exercise in tactical programming to provide the necessary operations for query-by-pointing and prove-by-pointing.

In summary, a few technical extensions to Isabelle's pretty-printing and parsing machinery are sufficient to make Isabelle support graphical mathematical notation and direct manipulation on terms. These extensions are fully compatible with Isabelle's logical genericity, and fully backwards-compatible with existing syntactic notations.

4.2 The Generic Graphical User Interface GenGUI

The module GenGUI uses the interface description facilities provided by sml_tk to provide a generic graphical user interface. It is independent of Isabelle, and given as a functor

```
functor GenGUI(structure appl: APPL_SIG) : GEN_GUI = ...
```

which returns a graphical user interface for the application `appl`. The abstract characterisation of an application has already been introduced in Sect. 3.1 above; we will now give a sketch of the ML signature `APPL_SIG` describing them. The real signature of course is far more elaborate, containing in particular details about the visual appearance (such as the size of the window, or the particulars of the icons depicting the objects and their locations).

The ML signature can roughly be divided into four parts: typing of the objects, operations and applying them, the construction area and external objects.

For the first part, every object has a type given by `obj_type`; its mode can be changed within the modes of the object's type, as given by `modes`. Objects of type `construction_obj` are construction objects, which can be opened and manipulated in the construction area:

```
signature APPL_SIG =
sig
  type object          (* The type of all objects *)
  eqtype objtype       (* The type of object types *)
  eqtype mode          (* The type of modes *)
  val obj_type         : object -> objtype
  val modes            : objtype-> mode list
  val mode_name        : mode-> string
  val initial_mode     : object-> mode
  val construction_obj : objtype
```

For the second part, there is a type modelling the operations, and an operation with which to apply it. Application is partial, and so the result of an application is either a new object (variant `OK`), or failure (variant `Error`, the string argument is an error message to be displayed):

```
  datatype object_result = OK of object | Error of string
```

```
  type opn
  val apply   : opn* object list-> object_result
  val mon_ops : objtype-> ((object* (opn->unit)-> unit)* string) list
  val bin_ops : (objtype* mode)* (objtype* mode)-> opn option
```

For every object type `mon_ops` gives the unary operations as a list of pairs of functions and strings. The string is the name under which the operation will appear in the pop-up menu; the function implements the operation. It gets passed the actual object as its first argument, and a continuation which is used to apply operations. The reason for passing a continuation is that a unary operation may require further user interaction (e.g. when starting a proof in a theory, we first have to enter some goal to be proven).

The binary operations are given by `bin_ops` and come into effect by drag&drop. For every type and mode of a target object (the one being dropped onto) and type and mode of objects being dropped, this function gives an option of an operation; if this option is empty then no operation is available for this drag&drop situation.

The construction area shows the delicate interplay between the application and GenGUI. Because the generic user interface implements the history and commands such as undo, the application cannot provide these. But since the layout of the construction area is given by the application, there needs to be a way to call functions which navigate the history, in order to bind them to graphical control elements. So we model the construction area by a functor, which takes the history navigation functions given by the signature `HISTORY_SIG` (omitted here), and implements the following export signature:

```
  functor ConArea (structure H : HISTORY_SIG):
    sig val open_area : object*H.history->TkTypes.Widget list
        val drop_ops  : objtype*mode->object list->(opn->unit)->unit
    end
```

The construction area provides the functions `open_area` which takes an object, and its history, and returns a list of widgets making up the construction area. For every type and mode of an object being dropped from the notepad into the construction area, `drop_obs` gives the operation to be applied. Like `mon_ops`, its arguments are the objects and a continuation to allow further user interaction.

The last part of the application deals with external objects. They are referred to by an identifier of type `external_id`. Given such such a reference, we may obtain an object from that by `get_external_obj`. The prime example here are file names; `get_external_obj` loads the contents of the given file. The application may specify dependencies between external objects (see below).

```
  eqtype external_id
  val ext_obj_depends_on : external_id* external_id-> bool
  val get_external_obj   : external_id-> object_result
end
```

The export interface shows a representation of the data model introduced in

Sect. 3.2. The type `obj_label` represents vertices of the term graph. `objects` is a representation of the term graph as a list of pairs (l, e), where l is a label and e an expression, consisting of applied operations, external objects or references to previous labels.

```
signature GEN_GUI= sig
  type obj_label
  datatype obj_hist = External  of external_id
                    | AppliedOp of opn* obj_hist list
                    | Result    of obj_label
  type objects   = (obj_label* obj_hist) list
  type notepad   = (obj_label* TkTypes.Coord) list
  type gui_state = objects* notepad
  val  change_external_obj : external_id-> unit
end
```

The notepad contains representations of vertices in the term graph on the screen, given as pairs of `obj_label` and `Coord`; we only need the coordinates, since the rest of the visual representation will be computed from other information (the type of the object etc.). Then the state of the whole system is given by the term graph and the notepad. Incidentally, this state is represented as a global reference; in Haskell it would be implemented more elegantly as a monad. We do not show the functions used to control the start and restart of the GenGUI, but importantly there is a function `change_external_id` by which an external application can signal that the value of an external object has changed. GenGUI then reevaluates the corresponding external object, and all external objects depending on it (as specified by `ext_obj_depends_on`), and moreover outdates all objects constructed from these external objects.

Note how the functional nature of the interface is reflected in the typing: all operations, given by `mon_ops`, `bin_ops` and `drop_ops`, can only produce new objects. The application cannot delete objects.

As a final detail, the clipboard is implemented by sharing a common structure `CLIPBOARD`, which exports two functions, `get: unit-> obj_hist` and `put: obj_hist-> unit`; if (and only if) `put` is called with the history of an object, the next call to `get` will return this history.

5 A Different Instantiation of the Generic Architecture

In this section we will demonstrate how instantiations of our generic architecture can be used to build a special purpose tool by encapsulating a formal method into Isabelle. The tool will be the transformation system `TAS` similar in spirit to window inferencing (Grundy, 1991) as realized for example in the system TkWinHOL (Långbacka *et al.*, 1995), and related to systems such as Prospectra (Hoffmann & Krieg-Brückner, 1993).

5.1 Concepts of TAS

In this section, we will briefly sketch the basic principles of modelling transformational program developments in an LCF-style prover, following the lines of Kolyang, Santen and Wolff (1996a). A transformational development can be described as a sequence of correctness-preserving refinement steps

$$SP_1 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

One can abstractly view the SP_i as arbitrary formulae and $\rightsquigarrow$ as a transitive, reflexive and monotone relation. Every development step $SP_i \rightsquigarrow SP_{i+1}$ is given by applying *transformation rules*, ranging from simple logical rules to complex ones that convert a certain design pattern into an algorithmic scheme, such as *Global Search* or *Divide & Conquer*.

The basic idea of the Transformation Application System TAS is to separate the *logical core* of a transformation from the pragmatics of its application, its *tactical sugar*, driving the concrete application in a development context. A logical core theorem has the following general form

$$\forall P_1, \ldots, P_n \,.\, A \Rightarrow I \rightsquigarrow O$$

where $P_1, \ldots, P_n$ are the *parameters* of the rule, A the *applicability condition*, I the *input pattern* and O the *output pattern*. By proving the logical core theorem, a transformation is proven correct. When applying a transformation, the applicability conditions result in *proof obligations* which are proven externally, by other interfaces to Isabelle (like IsaWin) or by decision procedures (e.g. model-checkers).

The Transformation Application System is designed to hide this implementation in the prover from the user. Since the proof obligations can be deferred to a later stage, the user of a transformation system can concentrate on the main design decisions of transformational program development: which transformation to apply, and how to instantiate its parameters.

5.2 TAS *as an Instantiation of the Generic GUI*

We will show how to set up TAS as an application in the sense of Sect. 3.1 above. We have to define object types, and operations. The construction objects of TAS will be transformational program developments, with a history of the transformation rules which have been applied. The object types are transformational program developments, transformation rules with none, some or all of their parameters instantiated, parameter instantiations, texts, and theories. Fig. 7 shows a screen shot of TAS with some objects on the notepad, and a transformational development currently open in the construction area. The operations include instantiating a transformation rule by dropping an instantiation on a transformation rule, and applying a transformation rule by dragging it into the construction area.

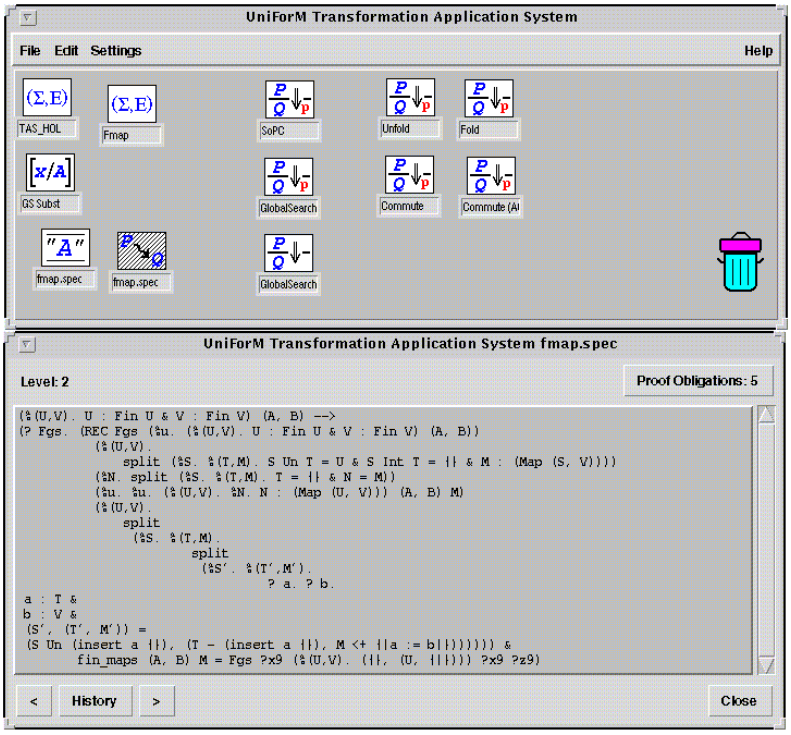


Fig. 7. The Graphical User Interface of `TAS`

6 Evaluation, Related Work and Conclusions

In this final section, we will discuss a metric evaluation of `IsaWin`, briefly review related work and close with a summary of our results and an outlook on future work.

6.1 Metric Evaluation of `IsaWin`

Card, Moran, and Newell (1983) proposed the *goals*, *operators*, *methods* and *selection rules* (GOMS) model and related it to the *keystroke-level model* (KLM). They postulate that the users formulate goals (e.g. prove lemma) and subgoals (e.g. push operator outermost) which they achieve by using methods (press key, move mouse, recall theorem name, etc.). The selection rules are the control structures for choosing among several methods available for accomplishing a goal — statistical assumptions about the deviation of these choices form the basis for a translation into the keystroke-level model. KLM attempts to predict performance times for error-free expert performance of tasks by summing up the time for key-stroking, pointing, drawing, thinking, and waiting for the system. Kieras and Polson (1985), and Elkerton and Palmiter (1991) refined the approach.

The original model, but to a lesser extent also its successors, "concentrate on expert users and error-free performance, and place less emphasis on learning, problem solving, error handling, subjective satisfaction and retention" (Shneiderman, 1998).

Given these fundamental constraints, it is not clear that the GORM model and its variants apply to theorem proving. Of course, what can be done at least is a rough comparison on the KLM-level of `IsaWin` interaction with pure command-line interaction, perhaps supported by a text-based editor with a short-cut facility. The proof script in Fig. 4 is generated by 47 elementary user-interactions like *set focus*, *drag-object*, etc. This contrasts substantially with the interaction necessary to produce a proof script of 233 characters by a command-line interface, even assuming editor short cuts. Further, the counterpart of proof-by-pointing and query-by-pointing in command-line interfaces, require the construction of substitutions, which is both tedious and extremely error-prone. The situation is also favourable for `IsaWin` if we take replay into account, which allows a much finer analysis of which proofs are affected by a change than the conventional rerunning of scripts which fails at the first problem.

Of course, the value of such taxonomic data is strictly limited, also because it ignores issues such as the flexibility of a rich tactical language. At present, claims like "GUI's improve productivity over command-line editors in some formal method" can not be founded on such data, although some first studies (Jackson, 1997) suggest this, at least for a particular prover and GUI. It may actually be the case that a GUI precisely because it is easier to use does not encourage purposeful planning to the extent which is necessary for the successful use of a theorem prover (Merriam & Harrison, 1997). Then again, it may be that a GUI makes the alternatives the user faces clearer and easier to invoke (Bornat & Sufrin, 1998). The question remains open until more systematic studies have been conducted; for `IsaWin`, the prototypical status of the implementation has until now precluded such studies.

6.2 Related Work

6.2.1 Generic Architectures and Abstract GUI Descriptions

Design patterns have recently received a lot of attention in the field of object-oriented programming (Gamma *et al.*, 1990; Cooper, 1998). Also motivated by reusability, some techniques (e.g. templates roughly corresponding to functors) are similar to our generic system architecture. However, important aspects of these patterns are described completely informal, resulting in a sometimes intransparent mixture of meta-language, `C++` code and pragmatics. In contrast, work on "architecture styles" (Abowd *et al.*, 1993; Allen & Garlan, 1994) aims at a fully formal description of generic architectures. However, for the moment, the emphasis of this research lays on foundation, description and analysis and less on implementation. Hence, we consider this work as complementary.

In the HCI literature, there is a large body of work applying formal methods, for the modelling of GUI's, based on temporal logic, Z or process algebras; see (Dix *et al.*, 1998) for a survey. Interface components can be described as processes exchanging events in a process algebra like CSP (Dix *et al.*, 1998, pp 320). A similar modelling in CSP could be done for our generic system architecture; then even the dialogue behaviour of the application can be described and specified formally.

6.2.2 GUI's for Theorem Provers

GUI's for computer algebra systems such as Maple, Mathematica or MuPad all offer mathematical editing facilities and some of them even direct manipulation of formulae (e.g. rewrite by drag&drop). Typically, this kind of direct manipulation is only available without genericity. These systems are built for a fixed syntax (with emphasis on arithmetics or differential equations), a fixed logic and on the basis of a non-generic system architecture. This also holds for the special purpose theorem prover CADiZ (Toyn, 1996).

In contrast, most recent theorem proving environments are generic, and some also offer proof support for direct manipulation. JAPE (Bornat & Sufrin, 1996) is generic in the logic and offers an interface with different styles of proof layout, graphical pretty-printing, and supports proof by direct manipulation, so-called "gestures". It is a lightweight prover, which has not been used yet to encapsulate a formal method. JAPE's gestures are similar to CtCoq (Bertot & Bertot, 1996), where they are called proof-by-pointing, but the basic idea remains the same. CtCoq is based on powerful prover, Coq, which unlike Isabelle is not generic, and moreover supports graphical output which can be configured by the user at runtime, script-based replay, and further direct manipulation like rewriting by drag&drop, Coq, which on the other hand is not generic.

CtCoq is actually part of a larger initiative, in spirit similar to ours, to provide generic interfaces for a family of provers (Bertot & Théry, 1998). The generic interface is implemented using the Centaur system (Borras *et al.*, 1988). In contrast to our architecture, the system is distributed (prover and interface can run on different machines) and heterogeneous (prover and interface need not be implemented in the same language). In our view, despite practical advantages, this does not lead to a better system architecture; and the close interaction between interface and prover possible because both are implemented in the same language leads to better support of direct manipulation and in particular, replay.

6.3 Results

In this paper, we have demonstrated how ideas of functional programming applied to user interface design gives rise to a new *functional* visualization metaphor, the notepad. The metaphor serves as vehicle to make the data structures of these provers accessible to pervasive direct manipulation.

The notepad allows for abstract manipulation of *values* represented by *icons*. The functional paradigm allows the recording of the construction history of every object, which is the key for a systematic replay.

A fundamental design decision in the implementation was to use the Tk toolkit, encapsulated into Standard ML by sml_tk. The encapsulation sml_tk helped us to survive the evolution of Tk in the recent years while taking advantage of its portability. As table 1 shows, sml_tk is largest chunk of code. Building on that, TAS and IsaWin can be kept fairly compact. To put these statistics into context, pure Isabelle has about 17500 lines of ML code. The code as produced by the

Module	Code size (lines of SML)
sml_tk	9900
GenGUI	2600
IsaWin	4800
TAS[a]	4500

[a] TAS and IsaWin share about 1400 lines of code.

Table 1. *Size of Code*

Standard ML of New Jersey compiler offers satisfactory response times, but shows a voracious appetite for memory. A running system will need at least 32 MB of memory, and to compile TAS and IsaWin 64 MB or more are required (on a Sun SPARC or UltraSPARC workstation). These numbers are mostly due to Isabelle and rise substantially with elaborate encodings like HOL-CSP; sml_tk itself compiles on far smaller machines.

Implementing the interface in a typed language with powerful modularization concepts rather than an untyped scripting language like Tcl or Lisp results in a clean, *generic* system architecture. We have presented two instantiations of this architecture, the interface `IsaWin` for the theorem prover Isabelle, and the transformation system `TAS`. As a consequence, we expect that our interface components can reused for a certain range of similar applications. In particular, this gives a blueprint for the construction of tools with graphical user interface for formal methods encoded into a theorem prover. We have in turn instantiated `TAS` with CSP and Z, two prominent formal methods for which encodings into Isabelle have been developed.

In a restricted area of interaction with Isabelle, our instantiations seem to substantially facilitate user interaction. This holds in particular for point-and-query, point-and-prove interactions and for global replay activities.

6.4 Future Work

`TAS` and `IsaWin` are still prototypical user interfaces that still need work in details. We would like to allow cut-copy-paste manipulation of the history; in particular the conversion of selected parts of the history to tactic objects would pave the way for powerful techniques of interactive reuse. Further, goals and substitutions are presently read as standard text and parsed via Isabelle's parsing machinery. This should be extended by a suitable mixture with structure-oriented editing facilities as in CtCoq, or mouse-supported input as in JAPE.

A far more involved subject is to scale up the systematic *replay* towards automatic *reuse* of former proof attempts. The most evolved replay and reuse techniques we are aware of are realized in the KIV-system (Reif *et al.*, 1997). KIV also provides direct manipulation on the history and moreover automatic support of reuse by detecting unaffected subparts of the proof which can still be used after failed replay. The

authors claim that the productivity of this system is essentially due to its reuse techniques (Reif & Stenzel, 1992).

However, this feature is based on a very specialized logic. Extending it for a generic theorem prover on the one hand and embedding it into our generic notion of history will represent a substantial challenge, but we believe that the deep incorporation of history both on the system level and on the generic interface level provides a good starting point.

References

Abowd, G., Allen, R., & Garlan, D. (1993). Using style to understand descriptions of software architecture. *Proceedings ACM SIGSOFT'93.* ACM Press.

Allen, R., & Garlan, D. (1994). Formalizing architectural connection. *Proceedings 16th conf. software engineering.* ACM Press.

Archer, J. E., Conway, R., & Schneider, F. B. (1984). User recovery and reversal in interactive systems. *ACM trans. on progr. languages and systems.*, **6**(1), 1– 10.

Bertot, J., & Bertot, Y. (1996). The CtCoq experience. *In:* (Merriam, 1996).

Bertot, Y. (1997a). Direct manipulation of algebraic formulae in interactive proof systems. *In:* (Bertot, 1997b).

Bertot, Y. (ed). (1997b). *User interfaces for theorem provers UITP'97.* INRIA Sophia Antipolis. Electronic proceedings at `http://www.inria.fr/croap/events/uitp97-papers.html`.

Bertot, Y., & Théry, L. (1998). A generic approach to building user interfaces for theorem provers. *Journal for symbolic computation*, **25**(2), 161– 194.

Bornat, R., & Sufrin, B. (1996). Jape's quiet interface. *In:* (Merriam, 1996).

Bornat, R., & Sufrin, B. (1998). Using gestures to disambiguate unification. *User interfaces for theorem provers UITP'98.*

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., & Pascual, V. (1988). Centaur: the system. *3rd symposion on software development environments.* (also as INRIA Report No. 777).

Bowen, J. P., & Gordon, M. J. C. (1994). Z and HOL. *Pages 141– 167 of:* Bowen, J. .P., & Hall, J. A. (eds), *Z users workshop.* Workshops in Computing. Springer.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer interaction.* Lawrence Erlbaum Associates.

Cooper, J. W. (1998). Using design patterns. *Comm. of the ACM.*, **41**(6), 65– 68.

Dix, A., Finley, J., Abowd, G., & Beale, R. (1998). *Human-computer interaction.* Prentice-Hall.

Elkerton, J., & Palmiter, S. (1991). Designing help using a GOMS model: an information retrieval evaluation. *Human factors*, **33**(2), 185– 204.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1990). *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley.

Gordon, M. J. C., & Melham, T. M. (1993). *Introduction to HOL: A theorem proving environment for higher order logics.* Cambridge University Press.

Grundy, J. (1991). Window inference with the HOL system. M. Archer, J. J. Joyce, Leveitt, K. N., & Windley, P. J. (eds), *International workshop on the HOL theorem proving system and its applications.* IEEE Computer Society Press.

Hoffmann, B., & Krieg-Brückner, B. (1993). *PROSPECTRA: program development by specification and transformation.* LNCS, no. 690. Springer.

Jackson, M. (1997). A pilot study of an automated theorem prover. *In:* (Bertot, 1997b).

Kahl, W. 1998 (Feb.). *The Higher Order Programming System — user manual for HOPS.* Universität der Bundeswehr München. URL `http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/`.

Kieras, D. E., & Polson, P. G. (1985). An approach to the formal analysis of user complexity. *International journal of man-machine studies*, **22**, 365– 394.

Kolyang, Santen, T., & Wolff, B. (1996a). Correct and user-friendly implementations of transformation systems. *Pages 629– 648 of:* Gaudel, M. C., & Woodcock, J. (eds), *Formal methods europe FME'96.* LNCS, no. 1051. Springer.

Kolyang, Santen, T., & Wolff, B. (1996b). A structure preserving encoding of Z in Isabelle. *Pages 283 – 298 of:* von. Wright, J., Grundy, J., & Harrison, J. (eds), *Theorem proving in higher order logics.* LNCS, no. 1125. Springer.

Långbacka, T., Ruksenas, R., & v. Wright, J. (1995). TkWinHOL: A tool for doing window-inferencing in HOL. *Pages 245– 260 of: Higher order logic theorem proving and its applications.* LNCS, no. 971. Springer.

Lüth, C., Westmeier, S., & Wolff, B. (1996). *sml_tk: Functional programming for graphical user interfaces.* Tech. rept. 7/96. Universität Bremen. See also the sml_tk home page at `http://www.informatik.uni-bremen.de/~cxl/sml_tk/`.

Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., & Wolff, B. (1998). Hol-Z in the UniForM-workbench – a case study in tool integration for Z. *Pages 116–134 of:* Bowen, J. P., Fett, A., & Hinchey, M. G. (eds), *ZUM'98: The Z formal specification notation.* LNCS, vol. 1493. Springer.

MacLane, S., & Birkhoff, G. (1967). *Algebra.* New York: Macmillan.

Merriam, N. (ed). (1996). *User interfaces for theorem provers UITP '96.* Technical Report. University of York. Electronic proceedings available at `http://www.cs.york.ac.uk/~nam/uitp96/proceedings.html`.

Merriam, N., & Harrison, M. (1997). What is wrong with GUIs for theorem provers? *In:* (Bertot, 1997b).

Nipkow, Tobias, & von Oheimb, David. (1998). Java$_{light}$ is type-safe — definitely. *Pages 161–170 of: Proc. 25th ACM symp. principles of programming languages.* ACM Press.

Ousterhout, J. K. (1994). *Tcl and the Tk toolkit.* Addison-Wesley.

Paulson, L. C. (1994). *Isabelle - a generic theorem prover.* LNCS, no. 828. Springer.

Regensburger, F. (1994). *HOLCF: Eine konservative Einbettung von LCF in HOL.* Ph.D. thesis, Technische Universität München.

Reif, W., & Stenzel, K. (1992). *Reuse of proofs in software verification.* Technischer Bericht 26/92. Universität Karlsruhe, Fachbereich Informatik.

Reif, W., Schellhorn, G., & Stenzel, K. (1997). Proving system correctness with KIV. *Pages 859– 862 of:* Bidoit, M., & Dauchet, M. (eds), *TAPSOFT '97: Theory and practice of software development.* LNCS, no. 1214. Springer.

Shneiderman, B. (1982). The future of interactive systems and the emergence of direct manipulation. *Behaviour and information technology*, **1**(3), 237– 256.

Shneiderman, B. (1998). *Designing the user interface.* 3 edn. Addison-Wesley.

Tej, H., & Wolff, B. (1997). A corrected failure-divergence model for CSP in Isabelle/HOL. *Pages 318–337 of:* Fitzgerald, J., Jones, C. B., & Lucas, P. (eds), *Formal Methods Europe FME '97.* LNCS, no. 1313. Springer.

Thimbleby, H. (1990). *User interface design.* ACM Press Frontier Series. Addison-Wesley.

Toyn, I. (1996). Formal reasoning in Z using CADiZ. *In:* (Merriam, 1996).

Vitter, J. S. (1984). US&R: A new framework for redoing. *IEEE software*, **1**(4), 39– 52.

Part IV

Selected Papers: Validation by Case Studies

A Verification Approach for Applied System Security

Achim D. Brucker[1], **Burkhart Wolff**[2]

[1] Information Security, ETH Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland, e-mail: brucker@inf.ethz.ch
[2] Universität Freiburg, George-Köhler-Allee 52, D-79110 Freiburg, Germany, e-mail: wolff@informatik.uni-freiburg.de

The date of receipt and acceptance will be inserted by the editor

Abstract. We present a method for the security analysis of realistic models over off-the-shelf systems and their configuration by formal, machine-checked proofs. The presentation follows a large case study based on a formal security analysis of a CVS-Server architecture.

The analysis is based on an abstract architecture (enforcing a role-based access control), which is refined to an implementation architecture (based on the usual discretionary access control provided by the POSIX environment). Both architectures serve as a skeleton to formulate access control and confidentiality properties.

Both the abstract and the implementation architecture are specified in the language Z. Based on a logical embedding of Z into Isabelle/HOL, we provide formal, machine-checked proofs for consistency properties of the specification, for the correctness of the refinement, and for security properties.

Keywords: verification, security, access control, refinement, POSIX, CVS, Z

1 Introduction

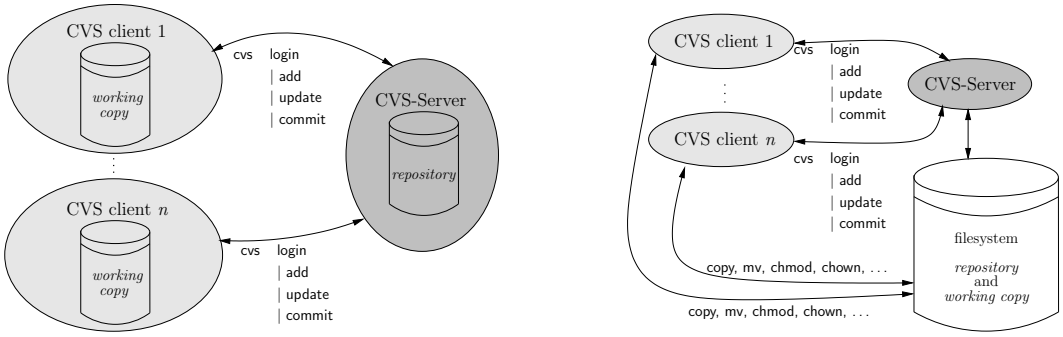
These days, the *Concurrent Versions System* (CVS) is a widely used tool for version management in many industrial software development projects, and plays a key role in open source projects usually carried out by highly distributed teams [3,5,4]. CVS provides a central database (the *repository*) and means to synchronize local modifications of partial copies (the *working copies*) with the repository. The repository can be accessed via a network; this requires a security architecture establishing authentication, access control and non-repudiation. A further complication of the CVS security architecture stems from the fact that the administration of authentication and access control is done via CVS itself; i.e. the authentication table is accessed and modified via standard CVS operations.

This work emerged from our own experiences with setting up a CVS-Server for more than 80 users worldwide. Besides overcoming a number of security problems (e.g. [13]), we had to develop an improved CVS-Server configuration described in [1] meeting two system design requirements: first, we had to provide a configuration of a CVS-Server that enforces a role-based access control [16]; second, we had to develop an "open CVS-Server architecture", where the repository is part of the shared filesystem of a local network and the server is a regular process on a machine in this network. While such an architecture has a number of advantages, the correctness and trustworthiness of the security mechanisms become a major concern. Thus, we decided to apply formal modeling and analysis techniques to meet the challenge.

In this paper, we present the method we developed for analyzing the security problems of complex systems such as the CVS-Server and its configuration. As a result, we provide the following contributions:

1. a modeling technique that we call *architectural modeling*, which has an abstraction level in-between the usual behavioral modeling used in protocol analysis and code verification,
2. a technique to use system architecture models for defining security requirements,
3. the presentation of the mapping from security requirements to concrete security technologies as a data refinement problem,
4. mechanized proof-techniques for refinements and security properties over system transitions, and
5. reusable models for widely used security technologies.

In particular, we provide means to model a certain type of security policies, and show how security analysis can be performed not only on the abstract, but also on the concrete level.



(a) "system" (b) "implementation"

Fig. 1. The different CVS-Server architectures

The paper is organized as follows: After introducing some background material, e.g. CVS, our chosen specification formalism Z and the architectural modeling style, we present the model of the abstract system architecture. We proceed with the model of the POSIX filesystem as an infrastructure for the implementation architecture, and present the implementation architecture itself. Then, we describe the refinement relation between the system architecture and the implementation architecture, and the analysis of security properties at the different layers based on formal proofs in an interactive theorem prover.

2 Background

2.1 The CVS Operations

For the purpose of this paper, it is sufficient to mention only the most common CVS commands (initiated by the client). These are: login for client authenticating, add for registering files or directories for version control, commit for transferring local changes to the repository, and update for incorporating changes from the repository (e.g. fetching the latest version from the repository) into the working copy. Additionally, CVS provides functionality for accessing the history, for branching, for logging information which is out of the scope of this paper, and a mechanism for conflict resolution (e.g. merging the different versions) which is only modeled as an abstract operation. Further, in order to facilitate both the refinement and the security analysis, we will include in our CVS model a operation which is strictly speaking not part of CVS but of the operating system: the operation modify. This operation models changes of the working copy, e.g. by editing a file.

2.2 Z and Isabelle/HOL-Z

As our specification formalism, we chose Z [19] for the following reasons: first, Z fits our modeling problem since the complex states of our components suggest to use a formalism with rich theories for data-structures. Second, syntax and semantics of Z are specified in an ISO-standard [9]; for future standardization efforts of operating system libraries (e.g. similar to the POSIX [20] model in Sec. 3.3.2), Z is therefore a likely candidate. Third, Z comes with a data-refinement notion [19, p. 136], which provides a correctness notion of the underlying "security technology mapping" between the two architectures and a means to compute the proof obligations. We assume a rough familiarity with Z (the interested reader is referred to excellent textbooks on Z such as [19, 21]).

As our modeling and theorem-proving environment, we chose Isabelle/HOL-Z [2], which is an integrated documentation, type-checking, and theorem-proving environment for Z specifications, which is built on top of Isabelle/HOL. Isabelle [11] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church's *higher-order logic* (HOL) [8], a classical logic with equality. Isabelle/HOL-Z is a conservative embedding of Z into HOL (which is semantically isomorphic to Z). As a result, Isabelle/HOL-Z combines up-to-date theorem proving technology with a widespread, standardized specification formalism, and powerful documentation facilities.

2.3 Architectural Modeling

As a means to identify conceptual entities of the problem domain and to structure the overall specification, we found it useful to describe the *architecture* of the system on several abstraction layers. Following Garlan and

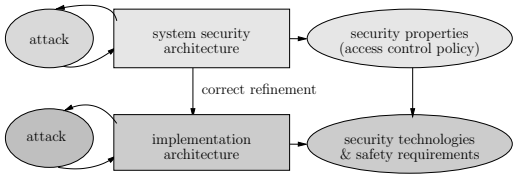


Fig. 2. Refining Security Architectures

Shaw's approach [7,18], architectures are composed by *components* (such as *clients*, *servers* or *stores* like the filesystem) and *connectors* (like *channels*, *shared variables*, etc). In this terminology it is straight forward to make the mentioned architectures more precise (as implementation architecture, we present the intended "open server architecture", see Fig. 1). We assume for each operation (such as add) a shared variable as connector that keeps all necessary information that goes to and from the components. This paves the way to formalize this architecture by describing the transition relation of the combined system by the parallel composition of the local transition relations of the components synchronized over the corresponding shared variable. Since such transition relations can be represented in Z [9] by *operation schemas*, we can thus define, for example:

$$CVS\_add = Client\_add \\ \wedge Server\_add \setminus add_{\text{shared variable}}$$

where $\wedge$ is the schema-conjunction and $\setminus$ the hiding operator (i.e. an existential quantifier). Throughout this paper, we will only present combined operation schemas and model properties over the transitive closure of their transition relations.

2.4 Architecture Refinement

When analyzing security architectures one can separate an *abstract security architecture* (see Sec. 3.2), which is merely a framework for describing the security requirements, from an *implementation architecture* (see Sec. 3.3) where a mapping to security mechanisms is described (see Fig. 2). By connecting the abstract and the concrete layer formally, it is possible to reason about safety and security properties on the abstract level. Such a connection between abstract and more concrete views on a system and their semantic underpinning is well-known under the term *refinement*, and security technology mappings can be understood as a special case of this. Various refinement notions have been proposed [21,14]; in our setting, we chose to use only a simple data refinement notion following Spivey [19].

2.5 Security Models vs. Security Technologies

Many security models distinguish between *objects* (e.g. data) and *subjects* (e.g. users). Using *role-based access control* (RBAC) [16] one assigns each subject at least one role (e.g. the role "administrator"), and access of objects is granted or denied by the role a subject is acting in. Further, roles can be hierarchically ordered, e.g. subjects in the role "administrator" are allowed to do everything other roles are allowed to. Our CVS-Server uses such a *hierarchic RBAC* model.

In an RBAC model, the decision, which roles may have access to which objects is done during system design and cannot be changed by regular users. In contrast, in a *discretionary access control* (DAC) model, every object belongs to a specific subject (its *owner*), and the owner is allowed to change the access policies at any time, hence "discretionary". For example a DAC implementation that also allows grouping users is the Unix/POSIX filesystem layer [20] access control.

Based on a DAC that supports groups, one can 'implement' an RBAC model by a special setup [15]. We use a similar technique to implement a hierarchic RBAC model for our CVS-Server on top of the POSIX filesystem layer, which is described in Sec. 3.3.3. However, we will analyze the concrete form in which DAC is implemented in POSIX and not a conceptual model thereof.

3 The CVS-Server Case Study

The Z specification of the CVS-Server [1] consists of more than 120 pages, and the associated proof scripts are about 13000 lines of code. The organization of the Z-sections follows directly the overall scheme presented in Fig. 3. The Z-sections *AbsState* and *AbsOperations* are describing the abstract system architecture of the client and the server components. The Z-section *SysConsistency* contains the consistency conditions (conservatism of axiomatic definitions, definedness of applications, non-blocking operation schemas) of the system architecture. This is mirrored at the implementation architecture level by the structures *FileSystem*, *CVS-Server*, and *ImplConsistency*. The Z-section *Refinement* contains the usual abstraction predicates relating the abstract and the concrete states, and also the proof obligations for this refinement. The security properties, together with the corresponding proof obligations, are defined in the sections *SysArchSec* and *ImplArchSec*.

3.1 Entities of the Security Model

Following the standard RBAC model, we introduce abstract types for CVS clients (users) *Cvs_Uid*, permissions *Cvs_Perm* (which are isomorphic to roles in our setting), and CVS passwords *Cvs_Passwd* used to authenticate a CVS client for a permission:

$$[Cvs\_Uid, Cvs\_Perm, Cvs\_Passwd]$$

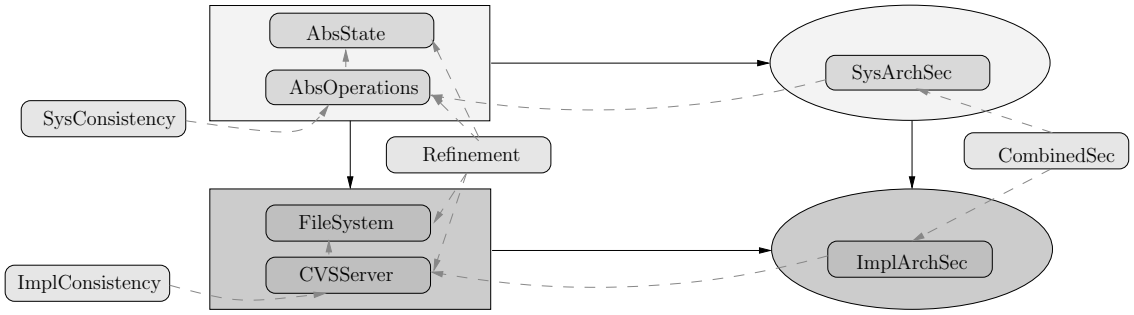


Fig. 3. Organizing the Specification into Z-sections

Permissions are hierarchically organized by the reflexive and transitive relation *cvs_perm_order* (over permissions *Cvs_Perm*) with *cvs_adm* as greatest element:

$$
\begin{array}{l}
cvs\_adm, cvs\_public : Cvs\_Perm \\
cvs\_perm\_order : Cvs\_Perm \leftrightarrow Cvs\_Perm \\
\hline
cvs\_perm\_order = cvs\_perm\_order^* \\
\forall\, x : Cvs\_Perm \bullet (x, cvs\_adm) \in cvs\_perm\_order \\
\forall\, x : Cvs\_Perm \bullet (cvs\_public, x) \in cvs\_perm\_order \\
\forall\, x : Cvs\_Perm \bullet (x \neq cvs\_adm) \Rightarrow \\
\qquad (cvs\_adm, x) \notin cvs\_perm\_order \\
\forall\, x : Cvs\_Perm \bullet (x \neq cvs\_public) \Rightarrow \\
\qquad (x, cvs\_public) \notin cvs\_perm\_order \\
\forall\, x : Cvs\_Perm \bullet \exists\, y : Cvs\_Perm \bullet \\
\qquad (x, y) \in cvs\_perm\_order
\end{array}
$$

We turn now to the security entities and mechanisms of the CVS-Server and the clients: first we have to model the working copies and the repositories as maps assigning *abstract names Abs_Name* to *data Abs_Data* (both types are abstract in our model):

$$
\begin{array}{l}
[Abs\_Name, Abs\_Data] \\
ABS\_DATATAB \equiv Abs\_Name \nrightarrow Abs\_Data
\end{array}
$$

A CVS-Server provides an authorization table, which is used to control access within the repository. The server stores for each file in the repository the required permission. These tables are modeled as follows:

$$
\begin{array}{ll}
AUTH\_TAB & \equiv Cvs\_Uid \times Cvs\_Passwd \\
 & \quad \nrightarrow Cvs\_Perm \\
ABS\_PERMTAB & \equiv Abs\_Name \nrightarrow Cvs\_Perm
\end{array}
$$

Clients possess in their working also a table that assigns to each abstract name a CVS client and another map that associates each CVS client to the password previously used during the cvs login procedure. The interplay of these tables will be discussed later, here we just define them:

$$
\begin{array}{ll}
ABS\_UIDTAB & \equiv Abs\_Name \nrightarrow Cvs\_Uid \\
PASSWD\_TAB & \equiv Cvs\_Uid \nrightarrow Cvs\_Passwd
\end{array}
$$

3.2 The System Architecture

In this section, we give a brief overview on how we model the system architecture, which is divided into: the state of the server (including the repository), the state of the client (including the working copy), and a set of CVS operations working over both of them.

It is a distinguishing feature of a CVS-Server to store the authentication data inside the repository such that it can be accessed and modified with CVS operations. This implies certain formal prerequisites: we require an abstract name *abs_cvsauth* to be associated with data, which can be converted into an authentication table via a postulated function *authtab*.

$$
\begin{array}{l}
abs\_cvsauth : Abs\_Name \\
abs\_auth\_of : Abs\_Data \nrightarrow AUTH\_TAB \\
abs\_data\_of : AUTH\_TAB \rightarrow Abs\_Data \\
authtab : ABS\_DATATAB \nrightarrow AUTH\_TAB \\
\hline
\mathsf{ran}(abs\_data\_of) \subseteq \mathsf{dom}(abs\_auth\_of) \\
\forall\, x : \mathsf{dom}\ abs\_auth\_of \bullet \\
\qquad abs\_data\_of(abs\_auth\_of\ x) = x \\
\forall\, x : AUTH\_TAB \bullet \\
\qquad abs\_auth\_of(abs\_data\_of\ x) = x \\
\forall\, r : ABS\_DATATAB \bullet abs\_cvsauth \in \mathsf{dom}(r) \Rightarrow \\
\qquad authtab(r) = abs\_auth\_of(r\ abs\_cvsauth)
\end{array}
$$

Modeling the server's state as a Z schema is straight forward. The state contains the repository *rep* and the map *rep_permtab* containing the required permissions for each file. Accessing the authentication table inside *rep* will require to have the role *cvs_adm*. *RepositoryState* is modeled as follows:

RepositoryState

$$
\begin{array}{l}
rep : ABS\_DATATAB \\
rep\_permtab : ABS\_PERMTAB \\
\hline
abs\_cvsauth \in \mathsf{dom}\ rep \\
\mathsf{dom}\ rep = \mathsf{dom}\ rep\_permtab \\
rep\_permtab(abs\_cvsauth) = cvs\_adm \\
rep(abs\_cvsauth) \in \mathsf{dom}\ abs\_auth\_of
\end{array}
$$

The state of the client component contains the working copy *wc*, the *wc_uidtab* assigning a CVS client to each file and a password table *abs_passwd* with credentials (passwords) used in previous CVS login operations (*abs_passwd* models the file .cvspass). Thus, for any data in the working copy and whenever an access to it may be processed, an individual role may be generated and validated by the server with respect to its current repository state. Further, there is a set of abstract names *wfiles* which is used as filter in update and commit operations. This filter corresponds to the concept of the *working directory* in the implementation, i.e. the effects of these operations are restricted to files stored within the working directory:

ClientState

$$wc : ABS\_DATATAB$$
$$wc\_uidtab : ABS\_UIDTAB$$
$$abs\_passwd : PASSWD\_TAB$$
$$wfiles : \mathbb{P}\, Abs\_Name$$

In the following, we define the abstract CVS operations that model combined state transitions of the client and the repository. Due to space reasons, we only present login and commit.

The login operation simply stores the authentication data on the client-side. This is used to authenticate a CVS user for a permissions of the client. The Δ and Ξ notation is used in Z to import the schemas in two variants; one variant as a copy, the other by replacing all variables by corresponding stroked variables (e.g. wc') describing the successor state. The Ξ also introduces equalities enforcing that the components of the previous state are equal to the post state components.

abs_login

$$\Delta ClientState$$
$$\Xi RepositoryState$$
$$passwd? : Cvs\_Passwd$$
$$uid? : Cvs\_Uid$$
$$(uid?, passwd?) \in \mathsf{dom}(authtab\ rep)$$
$$abs\_passwd' = abs\_passwd \oplus \{uid? \mapsto passwd?\}$$
$$wc' = wc$$
$$wc\_uidtab' = wc\_uidtab$$
$$wfiles' = wfiles$$

The commit (ci) operation usually takes a set of files as arguments (here denoted by *files*?). The case that no arguments may be passed is modeled by the possibility to set *files*? to the set of all files *ABS_NAME*.

Now we address the core of our hierarchic RBAC model of the system architecture, the *has_access* predicate. As a prerequisite, we define the shortcut is_valid_in for checking that a CVS client, together with a credential (password), represents a valid role with respect to the current repository:

$$\_\ \mathsf{is\_valid\_in}\ \_ : (Cvs\_Uid \times Cvs\_Passwd) \leftrightarrow ABS\_DATATAB$$
$$\forall\, role : Cvs\_Uid;\ pwd : Cvs\_Passwd;\ rep : ABS\_DATATAB \bullet$$
$$(role, pwd)\ \mathsf{is\_valid\_in}\ rep \Leftrightarrow (role, pwd) \in \mathsf{dom}(authtab(rep))$$

Further, the *has_access* predicate ensures is_valid_in and that the permissions resulting from these credentials are sufficient to access the requested file according to the role hierarchy:

$$has\_access\ \_ : \mathbb{P}(ABS\_PERMTAB \times ABS\_DATATAB \times PASSWD\_TAB \times Abs\_Name \times Cvs\_Uid)$$
$$\forall\, rep\_pt : ABS\_PERMTAB;\ rep : ABS\_DATATAB;\ pwtb : PASSWD\_TAB;\ file : Abs\_Name;\ role : Cvs\_Uid \bullet$$
$$has\_access(rep\_pt, rep, pwtb, file, role) \Leftrightarrow (role, pwtb(role))\ \mathsf{is\_valid\_in}\ rep \wedge (rep\_pt(file), authtab(rep)(role, pwtb(role))) \in cvs\_perm\_order$$

The commit operation consists of the construction of a new repository rep' and a new table with required permissions $rep\_permtab'$ which were constructed via the override operator $\oplus$ from previous states of these tables. For rep', three cases can be distinguished: (i) either a file in the repository does not occur in the working copy, then it is unchanged, or (ii) it occurs in the working copy but not in the repository, then it is copied provided a valid permission is available in the *wc_uid_tab* of the working copy, or (iii) the file exists both in working copy and repository, then the working copy file overrides the repository file whenever the client has access:

abs_ci

$$\Xi ClientState$$
$$\Delta RepositoryState$$
$$files? : \mathbb{P}\, Abs\_Name$$
$$(wfiles \cap files?) \subseteq \mathsf{dom}\ wc$$
$$rep' = rep \oplus (\{n : wfiles \cap files? \mid n \notin \mathsf{dom}\ rep \wedge n \in \mathsf{dom}\ wc\_uidtab \wedge (wc\_uidtab(n), abs\_passwd(wc\_uidtab\ n))\ \mathsf{is\_valid\_in}\ rep\} \lhd wc) \oplus (\{n : wfiles \cap files? \mid n \in \mathsf{dom}\ rep \wedge n \in \mathsf{dom}\ wc\_uidtab \wedge has\_access(rep\_permtab, rep, abs\_passwd, n, wc\_uidtab(n))\} \lhd wc)$$
$$rep\_permtab' = rep\_permtab \oplus \{n : wfiles \cap files? \mid n \notin \mathsf{dom}\ rep \wedge n \in \mathsf{dom}\ wc\_uidtab \wedge (wc\_uidtab(n), abs\_passwd(wc\_uidtab\ n)) \in \mathsf{dom}(authtab\ rep) \bullet n \mapsto authtab(rep)(wc\_uidtab(n), abs\_passwd(wc\_uidtab\ n))\}$$

The table *rep_permtab′* is extended by permissions for files that are new in the repository (based on the permissions used for committing these files). Further, the table *wc_uid_tab* is updated by the add operation, which we omit here.

In addition to these abstract models of the CVS operations, we provide a modify operation which explicitly models interactions of users with their files via modifying the files of the working copy of the client state.

3.3 The Implementation Architecture

The implementation architecture of CVS-Server is intended to model realistically the security mechanisms used to achieve the security goals formalized in the previous system architecture. Therefore, it captures the relevant operating system environment methods, i.e. POSIX methods in our case, for accessing files and changing their access attributes. We derived our POSIX model by formalizing the specification documents [20] and detailed system descriptions [6] and by validating it by carefully chosen tests and by inspections of critical parts of the system sources. In this POSIX model, the *CVS Filesystem* will be embedded, i.e. a repository is described as some area in the filesystem, where file attributes are set in a suitable way.

3.3.1 Modeling Basic Data Structures

We declare basic abstract sorts for POSIX user IDs, group IDs, data (file contents left abstract in this model), elementary filenames and file paths.

$$[Uid, Gid, Data, Name]$$
$$Path \equiv \mathrm{seq}\, Name$$

We assume a static table *groups* that assigns to each user a set of groups he belongs to. We also describe a special user ID *root*, modeling the system administrator. As we will show later, all security goals can only be achieved for all users except *root*, because root is allowed to do (almost) everything.

$$groups : Uid \to \mathbb{P}\, Gid$$
$$root : Uid$$

3.3.2 Modeling the POSIX Filesystem Access Control

Within POSIX, every file belongs to a unique pair of owner (user) and group, and file access is divided into access by the *user* (owner), the *group* or *other* (world). The POSIX *discretionary access control* (DAC) distinguishes access for reading (r), writing (w), and executing (x). We also model the "set group id" (sg) on directories, which affects the default group of newly created files within that directory (see [6] for more technical details about the Unix/POSIX DAC):

$$Perm ::= ru \mid wu \mid xu \mid rg \mid wg \mid xg \mid ro \mid wo \mid xo \mid sg$$

The filesystem consists of a map from a file path to file content (which is either *Data* for regular files or *Unit* for directories[1]) and of file attributes (assigning to each file or directory the permissions[2], the user ID of the owner and the group it belongs to). Our concept of file attributes may easily be extended by adding new components to its records.

$$Unit ::= Nil$$
$$FILESYS\_TAB \equiv Path \nrightarrow (Data + Unit)$$
$$FILEATTR \equiv [perm : \mathbb{P}\, Perm;\ uid : Uid;\ gid : Gid]$$
$$FILEATTR\_TAB \equiv Path \nrightarrow FILEATTR$$

We use type sums for modeling the *FILESYS_TAB* which are not part of the Z standard. Type sums can simulate enumerations in Z free type definitions on the fly. The two functions $Inl : X \to (X + Y)$ and $Inr : Y \to (X + Y)$ are provided for building type sums.

For testing if a directory contains a specific entry (either a file or a directory) we provide the function is_in. Further, we provide functions that test for regular files (is_file_in) and for directories (is_dir_in); their definitions are straight forward:

$$\_ \ \mathsf{is\_in}\ \_ : Path \leftrightarrow (Path \nrightarrow (Data + Unit))$$
$$\_ \ \mathsf{is\_dir\_in}\ \_ : Path \leftrightarrow (Path \nrightarrow (Data + Unit))$$
$$\_ \ \mathsf{is\_file\_in}\ \_ : Path \leftrightarrow (Path \nrightarrow (Data + Unit))$$
$$\forall\, fs : (Path \nrightarrow (Data + Unit));\ f : Path \bullet (f \ \mathsf{is\_in}\ fs) \Leftrightarrow f \in \mathsf{dom}\, fs$$
$$\forall\, fs : (Path \nrightarrow (Data + Unit));\ d : Path \bullet (d \ \mathsf{is\_dir\_in}\ fs) \Leftrightarrow (d \ \mathsf{is\_in}\ fs) \wedge (\exists\, u : Unit \bullet fs(d) = Inr(u))$$
$$\forall\, fs : (Path \nrightarrow (Data + Unit));\ f : Path \bullet (f \ \mathsf{is\_file\_in}\ fs) \Leftrightarrow (f \ \mathsf{is\_in}\ fs) \wedge \neg\, (f \ \mathsf{is\_dir\_in}\ fs)$$

At this point we are ready to model the filesystem state, which mainly describes the map of (name) paths to their attributes. As mentioned, we require that all defined paths must be "prefix-closed", i.e. all prefix paths must be defined in the filesystem (thus constituting a tree) and point to directories.

FileSystem

$$files : FILESYS\_TAB$$
$$attributes : FILEATTR\_TAB$$
$$\forall\, p : \mathsf{dom}\, files \bullet (p = \langle\rangle) \vee (front(p) \ \mathsf{is\_dir\_in}\ files)$$
$$\mathsf{dom}\, files = \mathsf{dom}\, attributes$$

In addition to the filesystem state, we introduce a state schema *ProcessState* for client related information,

[1] We do not consider *special files*, like devices, named pipes or process files.

[2] The terms *attributes* and *permissions* are used interchangeably.

namely the current user and group ID, the client's umask (which is used to set the initial file attributes on new files) and current working directory ($wdir$). The working directory is often used as an implicit parameter to filesystem and CVS operations:

$$
\begin{array}{|l}
\textit{ProcessState} \\
\hline
uid : Uid \\
gid : Gid \\
umask : \mathbb{P}(Perm \setminus \{sg\}) \\
wdir : Path \\
\hline
\end{array}
$$

As a prerequisite for describing functions that do modifications on the file system, we need to model the POSIX DAC in detail. Therefore we first introduce a function has_attrib, which decides whether the attributes (read, write and execute) of a file are set with respect to a specific user (and the groups he is a member of). Within this function, a crucial detail of the POSIX access model is formalized, namely that file access is checked by *sequentially* testing the following conditions (leading to an overall failure if the first condition fails):

1. If the user owns the file, he can only access the file if the access attributes for users grant access.
2. If the user is a member of the group owning the file, he can only access the file if the access attributes for the group grant access.
3. Last, the access attributes for others are checked.

These requirements may lead to some unexpected consequences, e.g. assume a user u being member of the group g and owner of a file with the permissions $(\!|\ perm == \{rg, ro\}, uid == u, gid == g\ |\!)$. Curiously, file access will be denied for him, while granted for all others in his group, because the rights specified for the user precede the rights given for the group.

$$
\begin{array}{|l}
has\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB \\
\qquad\qquad \times Perm \times Perm \times Perm) \\
\hline
\forall\, uid : Uid;\ fa : FILEATTR\_TAB; \\
\quad pu, pg, po : Perm \bullet \forall\, p : \mathsf{dom}(fa) \bullet \\
has\_attrib(uid, p, fa, pu, pg, po) \Leftrightarrow \\
\quad ((uid = root) \lor \\
\quad (\forall\, m : \mathbb{P}\, Perm;\ diruid : Uid;\ dirgid : Gid \mid \\
\qquad (\!|\ perm \equiv m, uid \equiv diruid, gid \equiv dirgid\ |\!) \\
\qquad\quad = fa(p) \bullet \\
\qquad\quad (diruid = uid \land pu \in m) \lor \\
\qquad\quad (diruid \neq uid \land dirgid \in groups(uid) \\
\qquad\quad \land\ pg \in m) \lor \\
\qquad\quad (diruid \neq uid \land dirgid \notin groups(uid) \\
\qquad\quad \land\ po \in m)))
\end{array}
$$

Based on has_attrib we introduce shortcuts for checking read, write and execute attributes (e.g. has_w_attrib) of files and directories as well as definitions for checking the read, write and execute access (e.g. has_w_access).

$$
\begin{array}{|l}
has\_w\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
has\_r\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
has\_x\_attrib\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
\hline
\forall\, uid : Uid;\ p : Path;\ fa : FILEATTR\_TAB \bullet \\
\quad has\_w\_attrib(uid, p, fa) \\
\qquad \Leftrightarrow has\_attrib(uid, p, fa, wu, wg, wo) \\
\quad \ldots
\end{array}
$$

$$
\begin{array}{|l}
has\_w\_access\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
has\_r\_access\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
has\_x\_access\_ : \mathbb{P}(Uid \times Path \times FILEATTR\_TAB) \\
\hline
\forall\, uid : Uid;\ p : Path;\ fa : FILEATTR\_TAB \bullet \\
\quad has\_w\_access(uid, p, fa) \Leftrightarrow \\
\qquad (\forall\, pref : Path \mid pref\ \mathsf{prefix}\ (front\ p) \bullet \\
\qquad has\_x\_attrib(uid, pref, fa)) \\
\qquad \land\ has\_w\_attrib(uid, front\ p, fa) \\
\quad \ldots
\end{array}
$$

As an example for our approach to specify POSIX operations, we present the (shortened) file remove specification [20], which corresponds to unlink():

> *The unlink() function shall fail and shall not unlink the file if:*
> - *A component of path does not name an existing file or path is an empty string.*
> - *Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the directory entry to be removed.*

This text is formalized by a Z operation schema rm as follows: The first condition in the body is common for most filesystem operations and requires the path of the file must be a valid one in the filesystem table. The second condition requires that the client has write permissions on the file and the working directory ("*the directory containing the directory entry to be removed*"), which is checked via the has_w_access predicate:

$$
\begin{array}{|l}
rm \\
\hline
\Delta FileSystem \\
\Xi ProcessState \\
u? : Name \\
\hline
(wdir \frown \langle u? \rangle)\ \mathsf{is\_file\_in}\ files \\
has\_w\_access(uid, wdir, attributes) \\
\quad \land\ has\_w\_access(uid, wdir \frown \langle u? \rangle, attributes) \\
files' = \{wdir \frown \langle u? \rangle\} \lhd files \\
\quad \land\ attributes' = attributes \\
\hline
\end{array}
$$

The definitions for the remaining filesystem operations are similar, see [1] for details.

3.3.3 Mapping CVS Access Control onto POSIX DAC

We turn now to a crucial aspect of the implementation of the security goals by security mechanisms provided from

standard POSIX DAC: any CVS role will be mapped to a particular pair of system *owner* and a set of system *group*s. This mapping has the consequence of an inheritance mechanism for generating default roles when creating new objects in the repository. Additionally, there is a mechanisms to "down-scale" and "up-scale" the permissions in the repository for the CVS administrator (not described here).

For every CVS operation, the server determines the CVS role according to the client's CVS ID and password. These roles are then mapped to POSIX user and group IDs, and these are compared to the file attributes of the files and directories the operations operates on. This translation is done by the two functions $cvsperm2uid$ and $cvsperm2gid$.

$$
\begin{array}{l}
cvsperm2uid : Cvs\_Perm \rightarrowtail Uid \\
cvsperm2gid : Cvs\_Perm \rightarrowtail Gid \\
users : \mathbb{P}\ Uid \\
\hline
root \notin \mathsf{ran}\ cvsperm2uid \\
\mathsf{ran}\ cvsperm2uid \cap users = \varnothing \\
\mathsf{ran}\ cvsperm2gid \cap \bigcup \{x : users \bullet groups(x)\} = \varnothing \\
\mathsf{ran}\ cvsperm2uid \lhd groups = \{x : Cvs\_Perm \bullet \\
\quad cvsperm2uid\ x \mapsto \\
\qquad \{c : Cvs\_Perm \mid (c, x) \in cvs\_perm\_order \bullet \\
\qquad\quad cvsperm2gid\ c\}\}
\end{array}
$$

It is important to notice that CVS IDs (Cvs_Uid) are independent of POSIX IDs (Uid) and that the POSIX IDs which are used by CVS are disjoint from "normal" POSIX user IDs, i.e. it is impossible to login with such a special POSIX ID.

From these distinctness constraints follows that the POSIX system administrator and the CVS administrator may be different. Moreover, we require that the group table (administrated by the system administrator and nobody else) is compatible with cvs_perm_order. These requirements have to be assured during installation of a CVS server.

The CVS repository is a subtree of the normal filesystem; its root is denoted by the absolute path cvs_rep and all paths inside the repository are relative to the root cvs_rep. Further, the administrative files of CVS are stored in the $CVSROOT$ directory, which is a subdirectory of cvs_rep, and the file that contains all authentication information is called $cvsauth$ and is located inside $CVSROOT$.

$$
\begin{array}{l}
cvs\_rep : Path \\
CVSROOT : Name \\
cvsauth : Name \\
auth\_of : Data \nrightarrow AUTH\_TAB \\
data\_of : AUTH\_TAB \rightarrow Data \\
\hline
\mathsf{ran}\ data\_of \subseteq \mathsf{dom}\ auth\_of \\
\forall\, x : \mathsf{dom}\ auth\_of \bullet data\_of(auth\_of\ x) = x \\
\forall\, x : AUTH\_TAB \bullet auth\_of(data\_of\ x) = x
\end{array}
$$

3.3.4 Modeling the CVS Filesystem

A major design decision for our specification is to enrich the *FileSystem* state by new state components relevant to CVS, or more precisely, the combined client/server component of CVS. In CVS, working copies contain specific attributes assigned to the files; we restrict ourselves to security relevant attributes, i.e. the CVS client ID and password, and the path *rep* where the file is located in the repository. This information is kept in an own table implicitly associated to the working copies.

$$
\begin{array}{ll}
CVS\_ATTR & \equiv [rep : Path;\ f\_uid : Cvs\_Uid] \\
CVS\_ATTR\_TAB & \equiv Path \nrightarrow CVS\_ATTR
\end{array}
$$

Due to the space reasons, we only show some requirements of the combined POSIX and CVS filesystem:

- working copies and the repository are distinct areas of the filesystem.
- the repository contains a special directory that contains the administrative data of CVS. Certain restrictive access permissions must be ensured to this directory and its contents to preserve the system integrity.
- requirements on file attributes within the repository:
  - since the owners of files must be POSIX user IDs that are disjoint from "regular" POSIX user IDs, and the group IDs must be legal with respect to the CVS role hierarchy. This guarantees that regular users only have the rights described by the file attributes for others. Thus, our initial invariant for the base directory of the repository implies that such a user cannot do anything, using only POSIX operations, within the repository.
  - read, write and execute permissions are the same for user and group. Together with our group setup this ensures that the initial CVS role and all roles with higher precedence have the same rights to access that file.

  These invariants are formally described in the axiomatic definition:

$$
\begin{array}{l}
attr\_in\_rep\ \_ : \mathbb{P}\ FileSystem \\
attr\_in\_root\ \_ : \mathbb{P}\ FileSystem \\
attr\_outside\_root\ \_ : \mathbb{P}\ FileSystem \\
\hline
\forall\, fs : FileSystem \bullet attr\_in\_rep(fs) \Leftrightarrow \\
\quad (\forall\, p : \mathsf{dom}\ fs.files \mid (cvs\_rep\ \mathsf{prefix}\ p) \bullet \\
\quad (((fs.attributes\ p).uid) \in \mathsf{ran}\ cvsperm2uid \\
\quad \wedge\ ((fs.attributes\ p).gid) \\
\qquad \in groups((fs.attributes\ p).uid) \wedge \\
\quad (ru \in ((fs.attributes\ p).perm) \Leftrightarrow rg \\
\qquad \in (fs.attributes\ p).perm) \wedge \\
\quad (wu \in ((fs.attributes\ p).perm) \Leftrightarrow wg \\
\qquad \in (fs.attributes\ p).perm) \wedge \\
\quad (xu \in ((fs.attributes\ p).perm) \Leftrightarrow xg \\
\qquad \in (fs.attributes\ p).perm))) \\
\quad \ldots
\end{array}
$$

We turn now to a formal description of the repository *within* the filesystem. This invariant of the system is captured in the state schema *Cvs_FileSystem*:

Cvs_FileSystem

$$
\begin{array}{l}
FileSystem;\ wcs\_attributes : CVS\_ATTR\_TAB \\
cvs\_passwd : PASSWD\_TAB \\
\hline
\mathsf{dom}\ wcs\_attributes \subseteq \mathsf{dom}\ files \\
(cvs\_rep \frown \langle CVSROOT, cvsauth\rangle\ \mathsf{is\_file\_in}\ files) \\
attributes(cvs\_rep) = \\
\lbag\ perm \equiv \{ru, wu, xu, xg, sg\}, \\
\quad uid \equiv cvsperm2uid(cvs\_adm), \\
\quad gid \equiv cvsperm2gid(cvs\_public)\ \rbag \\
attr\_in\_rep(\theta FileSystem) \\
((attributes(cvs\_rep \frown \langle CVSROOT\rangle)).gid) \\
\quad = cvsperm2gid(cvs\_adm) \\
attr\_in\_root(\theta FileSystem) \\
\quad \land\ attr\_outside\_root(\theta FileSystem)
\end{array}
$$

Additionally to *rep_attributes*, we impose similar requirements for the administrative area of the repository by the predicate *attr_in_root*. Further, we describe in the predicate *attr_outside_root* the requirements for the data in the repository, i.e. files that are subject to version control. Both axiomatic definitions are omitted here.

Now we have established a basis for the operations on the combined POSIX and CVS environment. As in Sec. 3.2, we present the login and commit operations in order to compare the two different architecture levels.

Before we describe the operations of the CVS-Server we need to model the access to the CVS authentication table (*get_auth_tab*) that is part of the $cvs\_rep \frown CVSROOT$ directory and underlies the standard access discipline of CVS-Server. In particular, the authentication table is only modifiable by the CVS administrator, but not by any other client of the system.

$$
\begin{array}{l}
get\_auth\_tab : FILESYS\_TAB \to AUTH\_TAB \\
\hline
\quad \ldots
\end{array}
$$

The login operation updates the variable *cvs_passwd*, provided that for the combination of user ID and password the authentication will succeed.

cvs_login

$$
\begin{array}{l}
\Delta Cvs\_FileSystem \\
\Xi ProcessState \\
cvs\_uid? : Cvs\_Uid \\
cvs\_pwd? : Cvs\_Passwd \\
\hline
(cvs\_uid?, cvs\_pwd?) \in \mathsf{dom}(get\_auth\_tab\ files) \\
cvs\_passwd' = cvs\_passwd \\
\qquad \oplus \{cvs\_uid? \mapsto cvs\_pwd?\} \\
wcs\_attributes' = wcs\_attributes \\
\theta FileSystem = \theta(FileSystem)'
\end{array}
$$

In the commit operation, the current working directory *wdir* can be restricted by the parameter *p?* to just one file or directory. All files below *p?* for which the client has access will be committed. We use the function *cutPath* to remove a given prefix from a path.

$$
\begin{array}{l}
cutPath : (Path \times Path) \nrightarrow Path \\
\hline
\forall\, a, b, c : Path \bullet\ cutPath(a, b) = c \Leftrightarrow a = b \frown c
\end{array}
$$

In contrast to the system architecture specification we also must determine the POSIX file attributes of the files. The particularity of the update and the commit operation is the use of *rep_access* which computes the paths into the repository to which the client has read access according to his CVS role.

$$
\begin{array}{l}
rep\_access : Cvs\_FileSystem \to Path \to \mathbb{P}\, Path \\
\hline
\forall\, cfs : Cvs\_FileSystem;\ p : Path \bullet \\
\quad rep\_access(cfs)(p) = \{q : Path \mid p\ \mathsf{prefix}\ q \\
\qquad \land\ cvs\_rep \frown q \in \mathsf{dom}\ cfs.files \\
\qquad \land\ (\exists\, idpwd : cfs.cvs\_passwd \bullet \\
\qquad\quad idpwd \in \mathsf{dom}(get\_auth\_tab(cfs.files)) \\
\qquad \land\ (has\_r\_access(cvsperm2uid(\\
\qquad\quad get\_auth\_tab(cfs.files)(idpwd)), \\
\qquad\quad cvs\_rep \frown q, cfs.attributes) \\
\qquad \lor\ (has\_x\_access(cvsperm2uid(\\
\qquad\quad get\_auth\_tab(cfs.files)(idpwd)), \\
\qquad\quad cvs\_rep \frown q, cfs.attributes) \\
\qquad \land\ cvs\_rep \frown q\ \mathsf{is\_dir\_in}\ cfs.files))\}
\end{array}
$$

The schema *cvs_ci* (see Fig. 4) models the commit command. We require that the client has read access for the file or directory in the current working directory and sufficiently high-ranked role to modify the repository.

4 Formal Analysis

A formal model, even if successfully type-checked, is in itself not a value of its own: it must be validated, e.g. by testing techniques or by formal proof activities as in our approach. In this section, we present a formal consistency check of the specifications, and we show that the implementation architecture is, in a formal sense, a refinement of the abstract system architecture. We specify and prove security properties of the type "no combination of user-commands will enable a user to write into the repository, except he has the required access rights".

4.1 Checking the Consistency

Two types of "sanity checks" are useful and have been carried out with HOL-Z [2] routinely:

– definedness checks for all applications of partial functions in their context, as undefined applications usu-

$$
\begin{array}{l}
\textit{cvs\_ci} \\
\Delta \mathit{Cvs\_FileSystem} \\
\Xi \mathit{ProcessState} \\
p? : \mathit{Path} \\
\hline
\mathit{has\_r\_access}(\mathit{uid}, \mathit{wdir} \frown p?, \mathit{attributes}) \\
\mathit{wdir} \in \mathsf{dom}\ \mathit{wcs\_attributes} \\
\mathit{files}' = \mathit{files} \oplus \{q : \mathit{rep\_access}(\theta \mathit{Cvs\_FileSystem})((\mathit{wcs\_attributes}\ \mathit{wdir}).\mathit{rep} \frown p?) \mid \\
\qquad \mathit{has\_r\_access}(\mathit{uid}, \mathit{wdir} \frown \mathit{cutPath}(q, (\mathit{wcs\_attributes}\ \mathit{wdir}).\mathit{rep}), \mathit{attributes}) \bullet \\
\qquad\qquad \mathit{cvs\_rep} \frown q \mapsto \mathit{files}(\mathit{wdir} \frown \mathit{cutPath}(q, (\mathit{wcs\_attributes}\ \mathit{wdir}).\mathit{rep}))\} \\
\mathit{attributes}' = \mathit{attributes} \oplus \{q : \mathit{rep\_access}(\theta \mathit{Cvs\_FileSystem})((\mathit{wcs\_attributes}\ \mathit{wdir}).\mathit{rep} \frown p?) \mid \\
\qquad \mathit{has\_r\_access}(\mathit{uid}, \mathit{wdir} \frown \mathit{cutPath}(q, (\mathit{wcs\_attributes}\ \mathit{wdir}).\mathit{rep}), \mathit{attributes}) \bullet \\
\qquad\qquad \mathit{cvs\_rep} \frown q \mapsto (\!|\ \mathit{perm} \equiv \{\mathit{ru}, \mathit{rg}\}, \\
\qquad\qquad\qquad \mathit{uid} \equiv \mathit{cvsperm2uid}(\mathit{get\_auth\_tab}(\mathit{files})((\mathit{wcs\_attributes}\ q).\mathit{f\_uid}, \\
\qquad\qquad\qquad\qquad \mathit{cvs\_passwd}((\mathit{wcs\_attributes}\ q).\mathit{f\_uid}))), \\
\qquad\qquad\qquad \mathit{gid} \equiv \mathit{cvsperm2gid}(\mathit{get\_auth\_tab}(\mathit{files})((\mathit{wcs\_attributes}\ q).\mathit{f\_uid}, \\
\qquad\qquad\qquad\qquad \mathit{cvs\_passwd}((\mathit{wcs\_attributes}\ q).\mathit{f\_uid}))) \\
\qquad\qquad |\!)\} \\
\mathit{wcs\_attributes}' = \mathit{wcs\_attributes} \\
\mathit{cvs\_passwd}' = \mathit{cvs\_passwd}
\end{array}
$$

Fig. 4. The specification of the commit command (implementation architecture)

ally indicate that some part of the precondition of a schema context is missing, and
- checking the state invariant of all operation schemas; in particular, we require that in a schema, all syntactic preconditions (i.e. the conjuncts in the predicate part that contain occurrences of variables without stroke "′" and "!" suffix) suffice to show that a successor state exists.

Violating these conditions does not result in logical inconsistencies but in unprovable statements or operation definitions with undesired semantical effects.

4.2 Establishing the Refinement

To prove that the concrete implementation architecture correctly implements the abstract system architecture, we have to define an abstraction schema R which relates the components of the abstract state to the components of the concrete state. In particular, we must map abstract names and data to paths and files in the sense of the POSIX filesystem, and the working copies and repositories of the abstract model must be related to certain areas of the filesystem, the authentication tables must be related, the user must not be *root* (the refinement simply does not work otherwise) and the file attributes in the concrete filesystem must be convertible along the mapping discussed in Sec.3.3.3.

Due to limited space, we will only show two constraints of R formally. As a prerequisite, let us define a function $\mathit{Rname2path}$, which maps abstract names, to file paths in the implementation model. One constraint is that *abs_cvsauth* is mapped to the right path and that the authentication tables in both models are equal:

$$
\begin{aligned}
\mathit{Rname2path}(\mathit{abs\_cvsauth}) &= \mathit{cvs\_rep} \\
&\quad \frown \langle \mathit{CVSROOT}, \mathit{cvsauth} \rangle \\
\mathit{authtab}(\mathit{rep}) &= \mathit{get\_auth\_tab}(\mathit{files})
\end{aligned}
$$

The last constraint we present here enforces the abstract working copy to have a counterpart in the implementation working copy:

$$\mathit{Rname2path}(|\mathsf{dom}\ \mathit{wc}|) = \mathsf{dom}\ \mathit{wcs\_attributes}$$

To verify the refinement relation R, following Spivey in [19], we must prove two refinement conditions for each operation on the abstract state and its corresponding operation on the concrete state: Condition (a) ensures that a concrete operation terminates whenever their corresponding abstract operation is guaranteed to terminate, condition (b) ensures that the state after the concrete operation represents one of those abstract states in which the abstract operation could terminate.

As an example of the refinement, we show the instantiation of conditions (a) and (b) for the CVS login operation. The refinement conditions, though, as defined in [19], assume that both operations have the same input parameters, but since we define them differently in our two models, we introduce an additional schema Asm, which is used to insert further assumptions into the refinement proofs (the effect could also have been achieved by a suitable renaming):

Asm

$passwd?, cvs\_pwd? : Cvs\_Passwd$
$uid?, cvs\_uid? : Cvs\_Uid$

$passwd? = cvs\_pwd?$
$uid? = cvs\_uid?$

In the case of the login operation, these assumptions are simple since the parameters are of the same type but differ in name. Instantiating condition (a) and (b) for the login operation and adding the assumption schema *Asm* leads to the following two proof obligations:

$$
\begin{aligned}
&login_a \equiv \forall\, ClientState;\ RepositoryState;\\
&\quad ProcessState;\ Cvs\_FileSystem;\\
&\quad passwd?, cvs\_pwd? : Cvs\_Passwd;\ uid?,\\
&\quad cvs\_uid? : Cvs\_Uid \bullet\\
&\qquad Asm \land \mathsf{pre}\ abs\_login \land R \Rightarrow \mathsf{pre}\ cvs\_login\\
&login_b \equiv \forall\, ClientState;\ RepositoryState;\\
&\quad ProcessState;\ Cvs\_FileSystem;\\
&\quad ProcessState';\ Cvs\_FileSystem';\ passwd?,\\
&\quad cvs\_pwd? : Cvs\_Passwd;\ uid?\\
&\quad cvs\_uid? : Cvs\_Uid \bullet\\
&\qquad Asm \land \mathsf{pre}\ abs\_login \land R \land cvs\_login\\
&\qquad\quad \Rightarrow (\exists\, ClientState';\ RepositoryState' \bullet\\
&\qquad\qquad R' \land abs\_login)
\end{aligned}
$$

The obligations for the other operations are defined analogously. So far, we proved these obligations formally for the refinement of login, add and update. These proofs considerably helped us to identify subtle side-conditions in our model and thus to get our real CVS configuration "right".

4.3 Security Properties in Architecture Layers

Specifying the security properties motivates a Z-section for the system architecture and one for the implementation architecture, both containing a classical *behavioral* specification. In *SysArchSec* we investigate security properties of the system architecture. In *ImplArchSec* we investigate the same properties and additional ones that are specific to the implementation architecture.

4.3.1 The General Scheme of Security Properties

As an interface between the operation schemas of the two architecture layers and the behavioral part allowing to specify safety properties, we convert suitably restricted operation schemas of both system layers into explicit relations over the underlying state. The purpose of these restrictions is to provide a slot for side-conditions that are related to the security model and not the functional model described in the previous sections:

$$
\begin{aligned}
rop_1 &= op_1 \land R_1\\
&\dots\\
rop_n &= op_n \land R_n
\end{aligned}
$$

where each rop_i represents the operation schema op_i constrained by the restriction schema R_i. Further the schema disjunction *step* represents the overall step relation of the system, which is converted into a transitively closed relation *trans*:

$$
\begin{aligned}
\text{step} &= rop_1 \lor \dots \lor rop_n\\
\text{trans} &= \{step \mid (\theta state, \theta state')\}^*
\end{aligned}
$$

In the literature, three types of properties can be distinguished: One may formalize properties over the *set of reachable states*, the *set of possible transitions* or the set of *possible sequences of states (traces)* of a system. While the first two types are only sufficient for classical safety invariants ("something bad will never happen"), the latter two allow for the specification of liveness properties ("eventually something good will happen"). The general scheme for properties over reachable states and possible transitions for safety properties and the schema for liveness properties looks as follows:

$$
\begin{aligned}
\mathrm{SP_{RS}} &= \forall \sigma : \mathrm{trans}(\!|\mathrm{init}|\!) \bullet P\sigma\\
\mathrm{SP_{RT}} &= \forall(\sigma, \sigma') : \mathrm{init} \lhd \mathrm{trans} \bullet P(\sigma, \sigma')\\
\mathrm{LP_{RT}} &= \forall(\sigma, \sigma') : \mathrm{init} \lhd \mathrm{trans} \bullet\\
&\qquad \exists(\sigma'', \sigma''') : \mathrm{trans} \bullet P(\sigma, \sigma', \sigma'', \sigma''')
\end{aligned}
$$

Note that the reachable states are restricted via the existential image operator or the domain restriction to the states (respectively transitions) reachable from the set of initial states init.

4.3.2 An Instance of the General Scheme: *RBAC_write*

We will exemplify the scheme SP_{RT} for a crucial security property, namely "the user may write in the repository only if he has RBAC-permissions", which we will call *RBAC_write* in the following. Moreover, we will outline the inductive proof.

As a prerequisite, we postulate two arbitrary sets *knows* and *invents*; a client "knows" a set of pairs of roles and passwords, and "invents" only files from a given set of pairs from names to data. We assume *invents* to be closed under the *merge*-operation left abstract in our model.[3] On this basis, we define a *security policy*, by providing suitable restrictions $op_i R$ for the system operations.[4] For example, we restrict the add operation to elements in the domain of the *invents*-set, we assume login being restricted to roles and passwords the client *knows* set, the modify operation and add being restricted to data the client "invents". While these restrictions have a more technical nature, a more conceptual restriction of *abs_ci* is as follows: in the role *cvs_adm*, the authentication table may only be altered such that rights are

[3] This is very similar to the concept of abstract crypt-functions and the closures *analz*, *synth* and *parts* in [12]; see discussion

[4] In practice, such security policies may be based on voluntary self-restrictions of users or enforced by administrative means.

withdrawn, not granted. A typical restriction looks as follows:

$$\begin{aligned} abs\_loginR \equiv\ & abs\_login \\ & \wedge [cvs\_uid? : Cvs\_Uid;\ passwd? : \\ & \quad Cvs\_Passwd \mid \\ & \quad (cvs\_uid?, passwd?) \in Aknows] \end{aligned}$$

Now we define the *step*-relation and its transitive closure of the system architecture layer:

$$\begin{aligned} step &\equiv abs\_loginR \vee abs\_addR \vee abs\_ciR \\ &\quad \vee abs\_modifyR \vee abs\_up \vee abs\_cd \\ AbsState &\equiv ClientState \wedge RepositoryState \\ trans &\equiv \{step \bullet (\theta AbsState, \theta AbsState')\}^* \end{aligned}$$

Finally, for constructing the proof goal $RBAC\_write$, we instantiate the P in our schema SP_{RT} by:

$$\begin{array}{|l} rbac\_write\_ : \ldots \\ \hline \forall\, rep, rep' : ABS\_DATATAB; \\ \forall\, rptab' : ABS\_PERMTAB \bullet \\ \quad rbac\_write(rep, rep', rptab') \Leftrightarrow \\ \qquad (\forall f : \mathsf{dom}\ rep' \bullet \\ \qquad\quad (rep(f) \neq rep'(f) \\ \qquad\qquad \wedge (f, rep'(f)) \in invents) \\ \qquad\quad \Rightarrow (\exists m : knows \bullet \\ \qquad\quad (rptab'(f), authtab(rep')(m)) \\ \qquad\qquad \in cvs\_perm\_order)) \end{array}$$

This property reads as follows: whenever there is a change in the repository, and the changed file stems from the users *invents*-set, the user must have valid permissions according to the *RBAC*-model. We observe that *rbac_write* is *true* whenever the repository does not change, i.e. $rbac\_write(r, r, rt)$ holds.

4.3.3 A Proof-Outline

We will now present an exemplary proof (performed with HOL-Z) for $RBAC\_write$. The initial proof goal stating that $RBAC\_write$ holds is refined by unfolding elementary definitions and simplification of Z notation to the following proof state:

$$\begin{aligned} [\![& \sigma_0 = (abs\_passwd, rep, rep\_permtab, wc, \\ & \quad wc\_uidtab, wfiles); \\ & \sigma_1 = (abs\_passwd', rep', rep\_permtab', wc', \\ & \quad wc\_uidtab', wfiles'); \\ & AbsState\sigma_0; \\ & AbsState\sigma_1; \\ & (\sigma_0, \sigma_1) : \{step \bullet (\sigma_0, \sigma_1)\}^* \\ ]\!] \Longrightarrow & \\ & \sigma_0 : init \\ & \quad \Rightarrow rbac\_write(rep, rep', rep\_permtab') \end{aligned}$$

Over this implication, we can now apply an induction rule over the transitive closure:

$$\frac{(a, b) \in r^* \qquad \begin{array}{c}[x \in \mathsf{dom}\ r] \\ \vdots \\ P\ x\ x\end{array} \qquad \begin{array}{c}[y \in \mathsf{ran}\ r] \\ \vdots \\ P\ y\ y\end{array} \qquad \left[\begin{array}{c} P\ x\ y, \\ (x, y) \in r^*, \\ (y, z) \in r \end{array}\right] \\ \vdots \\ P\ x\ z}{P\ a\ b}$$

This leads to two base cases and the induction step; both base cases are trivially true due to observation $rbac\_write(r, r, rt)$. Now the induction steps, which looks after some massage as follows, remains to show:

$$\begin{aligned} [\![& \ldots \\ & \sigma_{00} = (abs\_passwdx, repx, rep\_permtabx, wcx, \\ & \quad wc\_uidtabx, wfilesx); \\ & \sigma_{01} = (abs\_passwdy, repy, rep\_permtaby, wcy, \\ & \quad wc\_uidtaby, wfilesy); \\ & \sigma_{10} = (abs\_passwdz, repz, rep\_permtabz, wcz, \\ & \quad wc\_uidtabz, wfilesz); \\ & \sigma_{00} : init \Rightarrow rbac\_write(repx, repy, rep\_permtaby); \\ & (\sigma_{00}, \sigma_{01}) : \{step \bullet (\sigma_0, \sigma_1)\}^*; \\ & (\sigma_{00}, \sigma_{10}) : \{step \bullet (\sigma_0, \sigma_1)\} \\ ]\!] \Longrightarrow & \sigma_{00} : init \\ & \quad \Rightarrow rbac\_write(repx, repz, rep\_permtabz) \end{aligned}$$

Here, the point of proof refinement is the assumption $(\sigma_{00}, \sigma_{01}) : \{step \bullet (\sigma_0, \sigma_1)\}^*$, which can be decomposed via the definition of *step* into a disjunction of schemas, where the input variables are existentially quantified. A generic tactic strips away the disjunctions and the existential quantifiers in the assumption. The result is a case split over all operations of the system architecture and universally quantified input parameters of all operations under consideration. Now, the observation is crucial that all operations except *abs_ci* do not change the repository, and, as a consequence of observation $rbac\_write(r, r, rt)$, imply the truth of the step. We can therefore focus on the case *abs_ci*:

$$\begin{aligned} [\![& \ldots \\ & (\sigma_{00}, \sigma_{01}) : \{step \bullet (\sigma_0, \sigma_1)\}^*; \\ & rbac\_write(repx, repy, rep\_permtaby); \\ & abs\_ci(abs\_passwdy, abs\_passwdz, filesq, repy, repz, \\ & \quad rep\_permtaby, rep\_permtabz, wcy, wcz, \\ & \quad wc\_uidtaby, wc\_uidtabz, wfilesy, wfilesz) \\ & (\sigma_{00}) : init; \\ ]\!] \Longrightarrow & rbac\_write(repx, repz, rep\_permtabz) \end{aligned}$$

This is the core part of an invariance proof: the system made a transition from an initial system state (with *repx*) to another (with *repy*) performing an arbitrary combination of operations *and* the system behaved well (i.e. $rbac\_write(repx, repy, rep\_permtaby)$). Now a commit operation (*abs_ci*) occurs, and the question is if the

resulting state (with *repz*) will also fulfill our safety property.

The core of this subproof is, of course, a case distinction following the definition of *abs_ci* shown in Sec. 3.2: a file may be

1. in the repository *and not* in the working copy: then *abs_ci* will change nothing,
2. in the working copy *and not* in the repository: then *abs_ci* will only change the latter if the current credentials are is_valid_in which implies *write_correct* as the *rep_permtab* was changed accordingly,
3. both in the working copy *and* the repository: then *abs_ci* will only change the file in the repository if the current credentials allow for *has_access* which implies *write_correct*.

The interested reader may note that the overall scheme of the proof follows the structure of the general scheme of the property descriptions, which allows for automated tactic support that copes with Z-related technicalities, the choice of the inductions, the decomposition of the specification and the systematic derivation of state components remaining invariant. Obviously, there is a high potential of automation for this type of proofs, such that the proof developer may be guided rather automatically to the critical questions in the induction step.

4.3.4 Other Examples

The verification of the analogous property *RBAC_read* is straight forward; files in the working copy of a client are either invented by him (via the operation modify) or stem from the repository, where the client knows a password to obtain sufficient permissions.

An important, but quite obvious liveness property in the LP_{RT}-scheme is *RBAC_do_write*: Provided the client has access, it can change a file arbitrarily and perform operations leaving the repository changed accordingly; the proof immediately boils down to *abs_ci* which is designed to fulfill this property. At first sight, *RBAC_do_write* looks very similar to *RBAC_write*, however, note that both properties are independent: one could model an absolutely secure CVS-Server that never changes the repository. Such a model trivially fulfills *RBAC_write*, but is ruled out by *RBAC_do_write*.

So far, *RBAC_write* is formalized for a single-user client/server setting. Extending the analysis to a multi-user client/server model only requires simple modifications in the definition of the *step*-relation; via renaming of the working copies and the *invents* and *knows*-sets, instances of *abs_ci*, *abs_up* and *modify* for each client with individual working copy can be generated. Adding suitable restrictions (e.g. *invents* and *knows*-sets must be pairwise disjoint), *RBAC_write* and similar properties remain valid.

It is well-known that security properties are usually not preserved under refinement (see discussion later). The reason is that *implementing* one security architecture by another opens the door to *new* types of attacks on the implementation architecture that can be completely overlooked on the abstract level. For example, on the implementation architecture, it is possible to realize an attack on the repository by combinations POSIX commands such as *rm* and *setumask* etc (see Sec.3.3.2). In principle, our method can be applied for this type of analysis of the implementation architecture as well. In this setting, the *step*-relation and the *init* is defined as:

$$
\begin{aligned}
step_{impl} \equiv\ & rm \lor setumask \lor \cdots \lor chmod \\
& \lor cvs\_login \lor \cdots \lor cvs\_update \\
init_{impl} \equiv\ & ConcState \\
& \land [wcs\_attributes : CVS\_ATTR\_TAB \mid \\
& \quad wcs\_attributes = \varnothing]
\end{aligned}
$$

Although the proofs on the implementation architecture have the same structure as on the system architecture, they are far more complex since concepts such as paths, the distinction between files and directories, and their permissions are involved. Moreover, they require new side-conditions (for example, the refinement can only be established for the case that the user is not root) which were systematically introduced by the abstraction predicate R.

On the other hand, the higher degree of detail on the implementation architecture makes a formalization of new types of security properties possible: For example, since the crucial concept *directory* is present on the implementation level and since the existence of files can only be established by having access to all parent directories of a file, one can express confidentiality properties such as "the user can not find out that a file with name x exists in some directory of the repository" on this level.

5 Conclusion

5.1 Discussion

We demonstrate a method for analyzing the security in off-the-shelve system components thus made amenable to formal, machine-based analysis. The method proceeds as follows: First, specify the system architecture (as a framework for formal security properties), second, specify the implementation architecture (validated by inspecting informal specifications or testing code), third, set up the security technology mapping as a refinement, and fourth, prove refinements and security properties by mechanized proofs. The demonstration of the method follows a case study of a security problem for a real system, the CVS client/server architecture. We believe that the method is applicable for a wider range of problems such as mission-critical e-commerce applications or e-government applications.

The core of our approach is based on the presentation of the security technology mapping as data refinement problem. In general, it has been widely recognized that security properties can not be easily refined — actually, finding refinement notions that preserve security properties is a hot research topic [10,17]. However, standard refinement proof technology has still its value here since it checks that abstract security requirements are indeed achieved by a mapping to concrete security technology, and that implicit assumptions on this implementation have been made explicit. Against implementation specific attacks, we believe that specialized security property refinement techniques will be limited to restricted aspects. For this problem, in most cases the answer will be an analysis on the implementation level, possibly by reusing results from the abstract level.

In our approach, the analysis is based on interactive theorem proving while security analysis is often based on model-checking techniques for logics like LTL, the μ-calculus or process algebras like CSP. While these techniques offer a high degree of automation, they possess well-known and obvious limitations: the state-space must usually be finite and in practice be very small, and the analysis tends to be infeasible for many models, in particular those imposed by system specifications. As a consequence, proof engineers tend to develop oversimplified and unsystematically abstracted system models. In contrast, in our approach technical concerns like the size of the system state-space, aesthetic concerns like naturalness of the modeling (in our example, we use architectural modeling) or methodological needs like realistic treatments of system specifications do not represent fundamental obstacles to the analysis. In particular the latter paves the way for the reuse of standard system models like POSIX. Moreover, we have the full flexibility of Z and HOL to express security properties at need.

5.2 Related Work

Sandhu and Ahn described in [15] a method for embedding role-based access control with the discretionary access control provided by standard Unix systems. Our model used this construction for providing the static roles, but extended it to a dynamic model.

Wenzel developed a specification of the basic Unix functionality, which was done in Isabelle/HOL and is part of the actual Isabelle [11] distribution. On the file system part, only a simple access model, not supporting groups and the concepts of set-id bits, is formalized.

Our behavioral analysis is based on the same foundations as Paulson's inductive method for protocol verification [12]. Beyond the obvious difference, that Paulson research focus is on analysis (the language of protocols is deliberately small and restrictive) and not on modeling, technical differences consist merely in some details: Paulson uses specialized induction schemes which are automatically derived from the protocol-rules; these are considered as inductive rules defining the set of system traces. In contrast, we use standard induction over transitive relations, which leads to a different organization of the specification and the security properties and leads to different tactic support.

5.3 Future Work

In our opinion, amazingly little work has been addressed to the specification of the POSIX interface; due to its often not intuitive features, its importance for security implementations and its high degree of reuse, this is a particularly rewarding target. We believe that our formalization is a starting point for a comprehensive, more complete model of the filesystem related commands.

Clearly, the formal proofs established so far do not represent a *complete* analysis of the (real) CVS-Server. Many more security properties can be formulated, and, by setting up different operation restrictions R_i, "best-practice" security policies can be formally investigated. Moreover, in order to make implementation level security analysis more feasible, it could be highly rewarding to develop techniques and methods to reuse (abstract) system level proofs on the more concrete levels.

Acknowlegements

We would like to thank Nicole Rauch for many valuable discussions. Harald Hiss provided in his Diplomarbeit most of the proof-work.

References

1. A. D. Brucker, F. Rittinger, and B. Wolff. A CVS-Server security architecture — concepts and formal analysis. Technical Report 182, Albert-Ludwigs-Universität Freiburg, 2002.
2. A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
3. P. Cederqvist et al. *Version Management with CVS*, 2000. `http://www.cvshome.org/docs/manual/`.
4. `http://www.cvshome.org`.
5. K. Fogel. *Open source development with CVS*. The Coriolis Group, 1999.
6. Æ. Frisch. *Essential System Administration*. O'Reilly, 1995.
7. D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.
8. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
9. Z formal specification notation — syntax, type system and semantics, 2002. ISO/IEC 13568:2002.

10. J. Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (FME)*, volume 2021 of *LNCS*. Springer Verlag, 2001.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.
12. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
13. `http://www.cvshome.org/dev/security9706.html`.
14. A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.
15. R. Sandhu and G.-J. Ahn. Decentralized group hierarchies in UNIX: An experiment and lessons learned. In *National Information Systems Security Conference*, pages 486–502, 1998.
16. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.
17. T. Santen, M. Heisel, and A. Pfitzmann. Confidentiality-preserving refinement is compositional — sometimes. In *ESORICS*, volume 2502 of *LNCS*, pages 194–211. Springer Verlag, 2002.
18. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
19. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.
20. *The Single UNIX Specification Version 3*. The Open Group and IEEE, 2002. This standard superseeds the "Single UNIX Specification Version 2" (Unix 98) and the "IEEE Standard 1003.1-2001" (POSIX.1).
21. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall International Series in Computer Science. Prentice Hall, 1996. `http://www.usingz.com/`.

Author Index

B

Brucker, Achim. D. . 47, 65, 181, 205, 269

K

Karlsen,Einar 225
Kolyang 143

L

Lüth, Christoph 163, 225, 245

R

Rauch, Nicole 123
Rittinger, Frank 47

S

Santen,Thomas 143

T

Tej,Haykal 27

W

Westmeier,Stefan 225
Wolff, Burkhart 47, 65, 123, 181, 205, 269
Wolff,Burkhart . . 1, 27, 143, 163, 225, 245