Generic Transformational Program Development*

C. Lüth[1], H. Tej[1], and B. Wolff[2]

[1] Bremen Institute of Safe Systems, TZI, FB 3, Universität Bremen
Postfach 330440, 28334 Bremen
{cxl,ht}@informatik.uni-bremen.de
[2] Universität Freiburg, Institut für Informatik
wolff@informatik.uni-freiburg.de

Abstract. This paper describes the modelling of transformational program development inside a tactical theorem prover. The main characteristics of this approach is its genericity, leading to tools suitable for transformational program development in various, different formal methods which are logically embedded into the theorem prover. Combined with a systematic way of building graphical user interfaces, this yields a unifying framework for tool-supported formal program development, with correctness guaranteed by the theorem prover.

1 Introduction

During recent years, the need for formal methods in software development has been recognised increasingly. Along with this recognition, awareness has grown that "there is no single theory for all stages of the development of software" (C. A. R. Hoare [4]), and that formal methods have to be supported by appropriate tools. Thus if formal methods are to gain industrial relevance there is the need for a framework integrating different formal methods and formal methods tools.

The main aim of UniForM project [10] is to develop such a framework. Different formal methods and tools are combined into one universal development environment, with a common user interface and repository management covering the whole of the software life cycle.

The methodology of the UniForM workbench emphasises *transformational program development*. This paper describes our modelling of transformational program development inside a tactical theorem prover (here, Isabelle [12]). This modelling is generic over both the logic and the transformation relation, and together with a logical embedding of a formal method into the theorem prover, we can uniformly describe transformational program development in the context of different formal methods such as Z or CSP. We will sketch a way in which graphical user interfaces for applications based on tactical theorem provers can be built. To tickle the reader's curiosity, we will first give an impression of the Transformation Application System.

* 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.

2 The Transformation Application System at Work

We will now given an example concerning the refinement of CSP process specifications. The example aims at demonstrating the look and feel of the Transformation Application System (TAS). We assume a passing knowledge of CSP or another process calculus such as CCS, but even without any previous knowledge the reader will hopefully be able to form an impression. The development starts out with two processes `COPY1` and `COPY2` running in parallel. Both accept requests, and send responses; one may perhaps think of this as the specification of a concurrent data base server. When implementing such a server, we may not want "real" concurrency, since both processes may access shared resources; so we have to synchronize the two server processes with a central scheduler. This development will be described as a refinement of CSP specifications in the following.

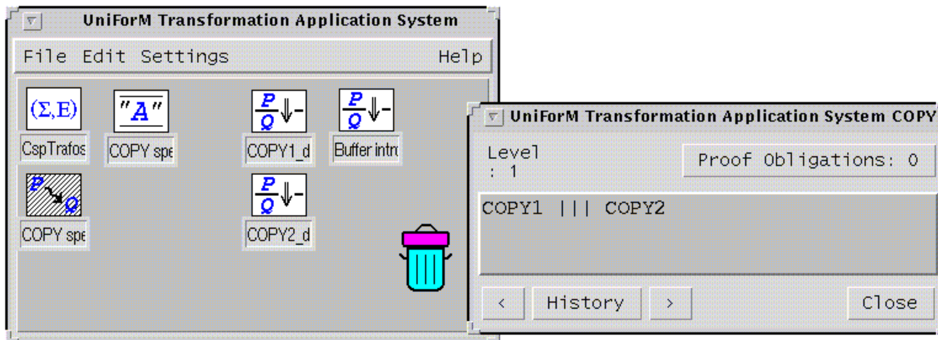

Fig. 1. Initial Configuration

Fig. 1 shows a screenshot of TAS set up for this example. On the *notepad*, the window on the left side, we can see various icons representing objects, such as the theory in which all the CSP transformations live (upper left hand corner), the transformational development in its initial stage (which is shaded, because it is open in the *construction area* below), and a couple of transformations for use later. The notepad only contains transformations which the user has explicitly placed there, not all transformations known in the system, since there would be too many of them.

A transformation is applied by dragging it down into the construction area. To apply a transformation in context, the subterm of the current specification in the construction area is selected with the mouse. After replacing `COPY1` and `COPY2` with their definition, we want to introduce buffer processes. The transformation introducing buffers can be visualised as in Fig. 3, and intuitively replaces the single process `COPY` with two processes `SEND` and `RECV` synchronised via two channels `mid` and `ack`, which are the parameters of this transformation. The applicability condition of this transformation is that the instantiation for `mid` and `ack` are

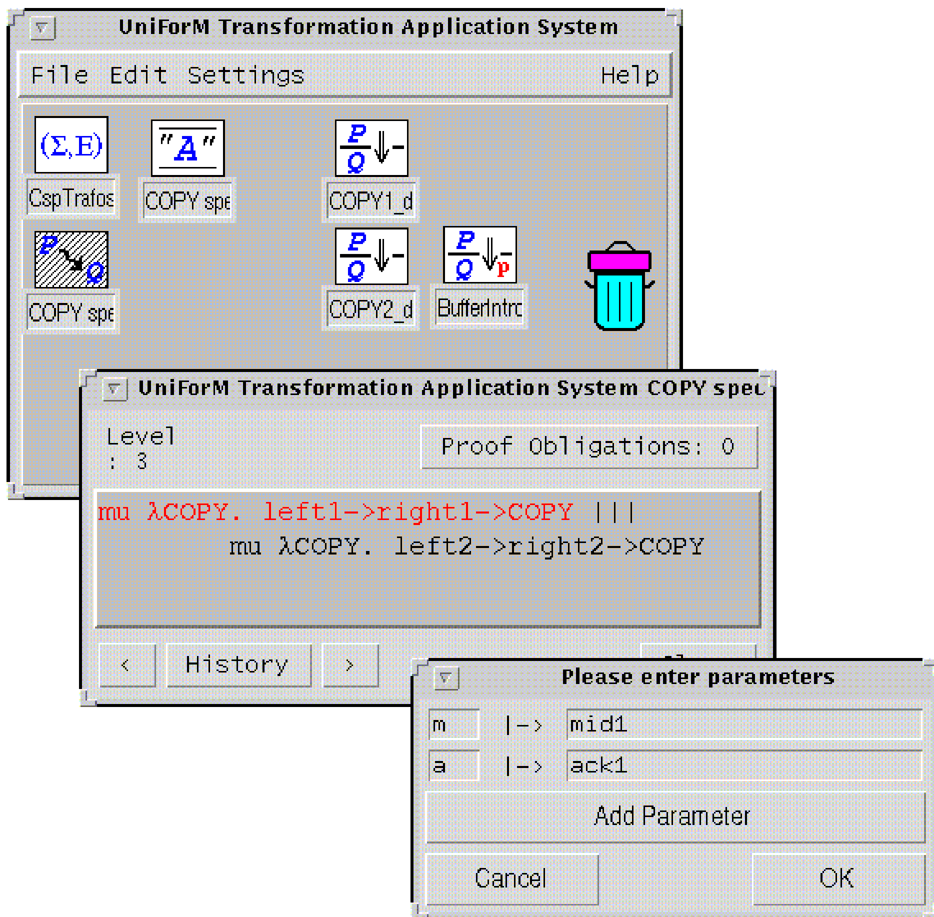

Fig. 2. Applying a transformation with parameters.

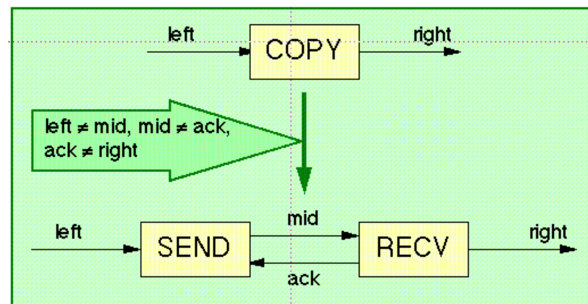

Fig. 3. Transformation: Introducing a buffer

pairwise disjoint from each other and from `left` and `right`. To apply this rule, we mark the relevant subterm, and drag the rule into the construction area. A window pops up querying for the instantiations of the the parameters, which we have to supply (see Fig. 2).

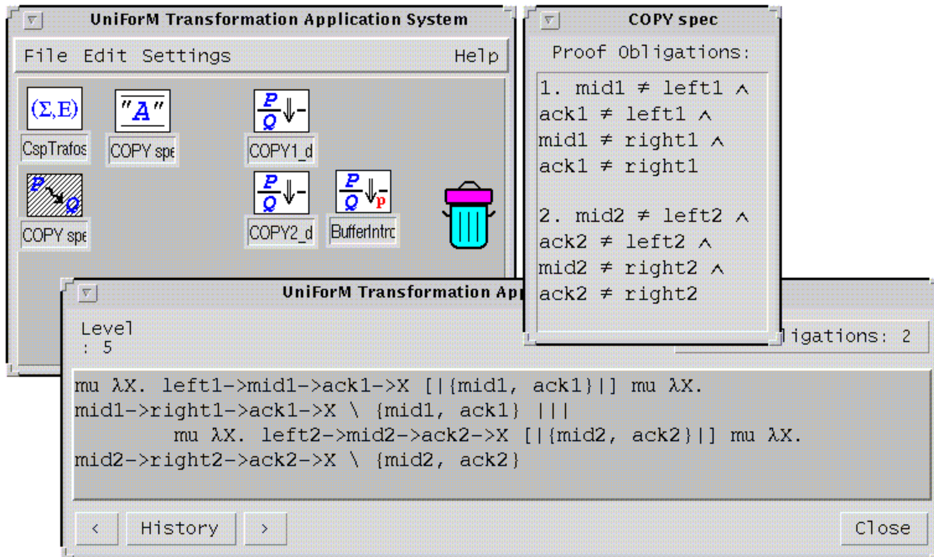

Fig. 4. Displaying proof obligations

We can do the same with the `COPY2` process, and arrive at the stage of development shown in Fig. 4. Note that we now have two proof obligations, which can be displayed by clicking on the button labelled proof obligations. At any stage during the development process, proof obligations can be dragged from this window into the IsaWin window, where they can be proven.

As the Isabelle/CSP notation can become fairly lengthy, we can visualise the transformational development in the style of Fig. 3; Fig. 5 shows the double application of the buffer-introducing transformation just described. This visualisation does not show the hiding operators, though, and can hence be misleading.

Now follow two transformations which prepare the specification for the transformation introducing the scheduling process. The first of these transforms the specification from two synchronized buffer processes consisting of a sender and receiver process each to one buffer process consisting of two synchronized sender and receiver processes; the second pushes all hiding operators to the outside. The scheduling process itself consists actually of two processes, one to synchronize the received messages, and one to synchronize the outgoing acknowledgements. Applying the transformation to introduce the first of these shown in Fig. 7. The transformation has applicability conditions and parameters (viz, the names of the

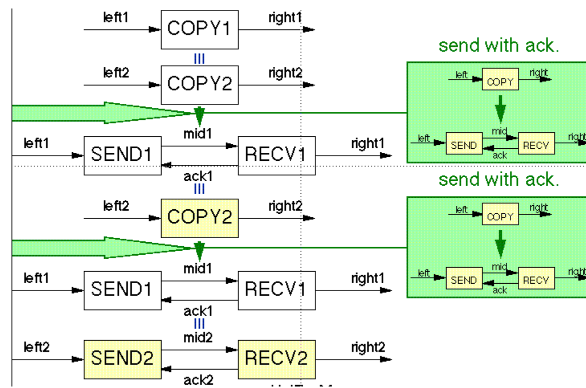

Fig. 5. Stepwise Transformational Development

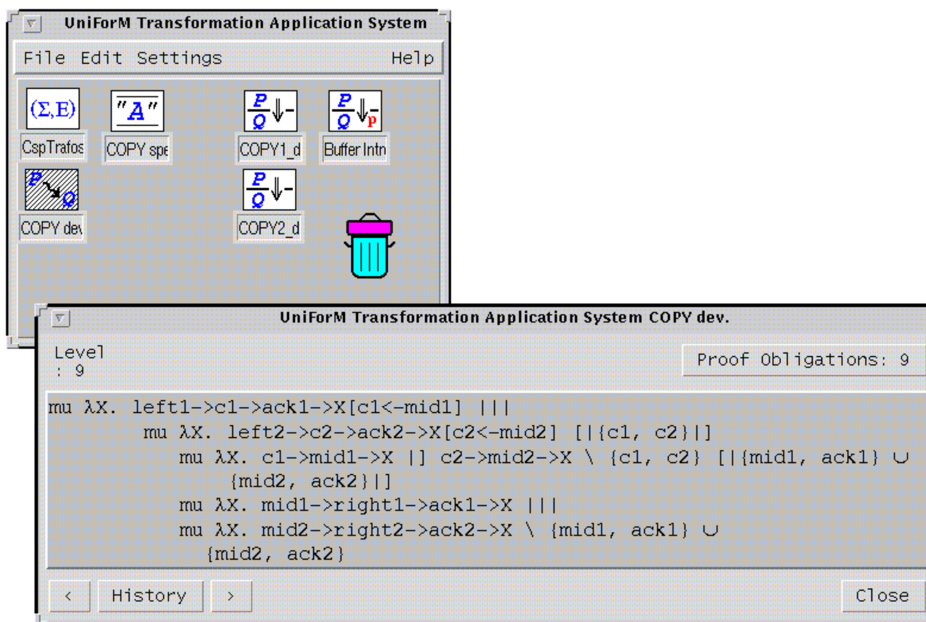

Fig. 6. The final stage of the development

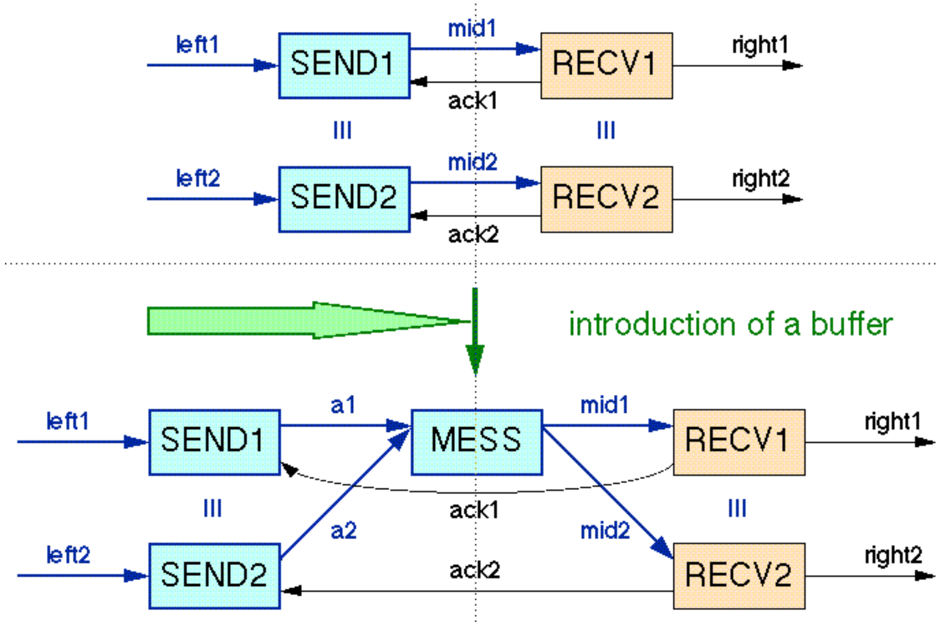

Fig. 7. Transformation: Synchronized Buffer

freshly introduced channels `a1` and `a2` and the usual disjoint conditions. Application of this transformation twice yields the final stage of the development, shown as a screenshot in Fig. 6. The development can be displayed with a hypertext browser (Fig. 9). Furthermore, it can be closed and turned into a new transformation rule — i.e. we can abstract from the particular development. Fig. 8 shows the relevant menu being activated, and the new transformation rule has already appeared in the lower left corner.

3 Transformational Program Development in Isabelle

In this section, we will give a brief introduction to the LCF prover Isabelle, and describe how a transformational development like the one presented in Sect. 2 can be realized inside Isabelle with the Transformation Application System. It should be pointed out that the technicalities described in this section all remain hidden from the actual user of the system.

3.1 LCF Theorem Provers and Isabelle

The family of LCF theorem provers originate from the seminal Edinburgh LCF system by Milner, Wadsworth and Gordon in the 70's [3]. Their main characteristics are the way they are embedded into the functional programming language ML, and their flexibility due to so-called tactics.

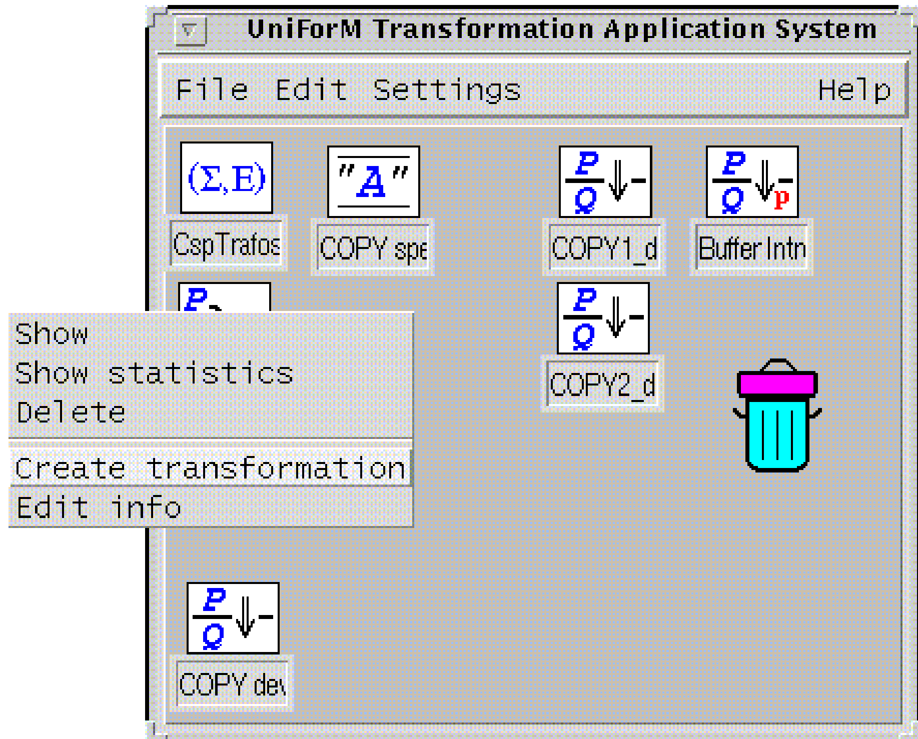

Fig. 8. Development abstraction: creating new transformations

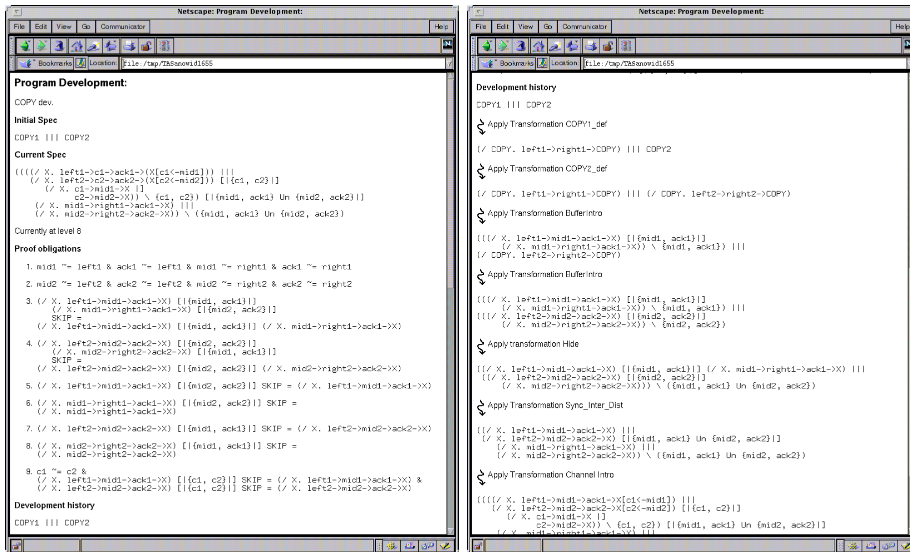

Fig. 9. Displaying developments

An LCF prover essentially consists of a collection of ML modules and functions. The user types ML expressions or ML programs, which are evaluated and their result printed. This design obviously leaves something to be desired as far as user-friendliness is concerned, but it is very powerful and extendible. Using specific functions provided by the prover, the user can program *tactics* as ML programs, allowing a higher degree of proof automation.

The main contemporary LCF provers are the HOL system [2] and Isabelle [12], the latter of which is used in our work, since it is more flexible, and more powerful (offering built-in tools for automatic proof such as a rewriter, and a so-called classical reasoner).

The two basic proof methods in Isabelle are *forward resolution* and *backward resolution*. Forward resolution is a way to derive new theorems: if we have two theorems $\frac{P}{Q}$ and $\frac{R}{S}$, and we can find substitutions σ, τ such that $\sigma(Q) = \tau(R)$, then we can derive (by the associativity of the implication) the new theorem $\frac{\sigma(P)}{\tau(S)}$. Backward resolution drives the proof activity. If we wish to prove a goal P, then backward resolution with a theorem $\dfrac{Q_1, \ldots, Q_n}{Q}$ means that we find a substitution σ which when applied to Q yields P, $\sigma(Q) = P$. To prove P, we now have to prove $\sigma(Q_1), \sigma(Q_1), \ldots, \sigma(Q_n)$. These new goals are called *subgoals*. The main way to prove the subgoals is by proving them separately. Isabelle keeps track of them, and displays them as a list of numbered subgoals rather than one conjunction. Isabelle will assist in finding the substitutions mentioned above, but this involves higher-order unification which is in general undecidable, so it is not guaranteed that Isabelle will find such a substitution if it exists.

3.2 Transformational Program Development

In general, transformational development is described by a sequence of specifications

$$SP_1 \rightsquigarrow \ldots \rightsquigarrow SP_n \tag{1}$$

In a full transformational development SP_1 is the requirement specification and SP_n the executable specification (from which a program can be generated), but a transformational development may describe any subsequence of this as a single step in the overall development process, deriving a more refined specification or program from a more general one.

In this framework, we consider the SP_i to be arbitrary formulae of higher-order logic, and development steps $SP_i \rightsquigarrow SP_{i+1}$ are described by a relation $\rightsquigarrow$ called the *transformation relation*. The transformation relation can be any transitive-reflexive relation; additional monotonicity allows transformations to be applied inside a context (see below). The transformation relation can be thought of as the refinement relation underlying the transformational development, i.e. $SP \rightsquigarrow SP'$ iff SP' is a refinement of SP. The transformational development above amounts to *proving* formula (1) from preconceived rules called *transformations* (below we will show examples of these). Steps in the proof correspond to applying transformation rules.

A transformation for a transformation relation $\rightsquigarrow$ is given by a *logical core theorem* of the following general form:

$$\forall P_1, \ldots, P_n \,.\, A \Rightarrow I \rightsquigarrow O \tag{2}$$

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*.

The transformational development is started by creating an initial proof state

$$1.\ SP_1 \rightsquigarrow ?Z$$

where $?Z$ is the Isabelle notation for a so-called *meta variable* representing the final result. Meta variables are free variables in a term subject to unification, and can be thought of as named "holes" in the term.

The transformation given by the core theorem 2 is *applied* by performing the following sequence of tactical operations: first, a resolution with the transitivity of $\rightsquigarrow$ is carried out. This leads to a proof state with two subgoals:

$$\begin{array}{l} 1.\ SP_1 \rightsquigarrow ?Y \\ 2.\ ?Y \rightsquigarrow ?Z \end{array}$$

where $?Y$ is the new intermediate specification and $?Z$ remains the ultimate target of the development. In the second step, $?Y$ is substituted by the transformed specification SP_1: by forward resolution of the logical core theorem 2 with the elimination rules for the universal quantifier and the implication, one obtains the logical core theorem in a form where the variables $P_1, \ldots, P_n$ bound by the universal quantifier are substituted by meta variables $?P_1, \ldots, ?P_n$

$$\frac{A'}{I' \rightsquigarrow O'} \tag{3}$$

We now find a substitution σ such that $\sigma(I') = SP_1$, and resolve the transformed core theorem 3 with subgoal 1. The unification of the conclusion of 3 and subgoal 1 yields a substitution for the meta-variable $?Y$, which is the transformed program $SP_2 \stackrel{def}{=} \sigma(O')$; applying it to the applicability conditions yields the proof obligation $L \stackrel{def}{=} \sigma(A')$. The proof obligations will appear as a new subgoal, since they also need to be proven to make the transformation sound; in fact, the applicability condition is typically a conjunction $A = A_1 \wedge \ldots \wedge A_n$, so $L = \sigma(A'_1 \wedge \ldots \wedge A'_n) = \sigma(A'_1) \wedge \ldots \wedge \sigma(A'_n)$ is a conjunction as well, which will show up as n subgoals. The proof state after applying the transformation thus reads

$$\begin{array}{ll} 1. & \sigma(A'_1) \\ \vdots & \\ n. & \sigma(A'_n) \\ n+1. & SP_2 \rightsquigarrow ?Z \end{array}$$

The application of the next transformation will be focused on subgoal $n+1$ and so forth. This way, transformational developments can be represented within

the infrastructure of Isabelle, allowing browsing and copying developments and abstract operations on them.

The hard part with real-life design transformations (such as Global Search [8] or Split of Postcondition [5]) is finding the right instantiation of the rule's parameters. In these cases, the instantiation σ above will most likely not be found automatically by Isabelle's unification, but users will have to supply instantiations $R_1, \ldots, R_n$ for the parameters $?P_1, \ldots, ?P_n$ (after careful thought on their part), from which Isabelle will be able to find the correct σ by unification. On the other hand, for simple transformations, e.g. those based on folding or unfolding an equation, Isabelle will most likely find the substitution σ if it exists. Note that the unification of I' with SP_1 can fail, which means that the transformation is not applicable here.

To apply a transformation in a context, the transformation relation needs to be monotone with respect to that particular context. In higher-order logic, contexts are represented as λ-abstractions; so e.g. $C[SP]$ (the specification SP in the context C) is represented as $(\lambda x.Cx)SP$. When applying transformation 3 to SP *in the context* C, a new subgoal is generated which requires $\rightsquigarrow$ to be monotone with respect to C:

$$n+1.\ ?S \rightsquigarrow ?T \Longrightarrow C(?S) \rightsquigarrow C(?T)$$
$$n+2.\ SP_2 \rightsquigarrow ?Z$$

This subgoal will either be resolved automatically by Isabelle's rewriting (by applying congruence rules for single operators with respect to $\rightsquigarrow$ top-down), or if it cannot be resolved, the transformation in context fails. This means that C contains an operation with respect to which $\rightsquigarrow$ is not monotone (a typical example here is the process refinement ordering for CSP processes from Sect. 2, which is not monotone with respect to hiding). One might have thought of requiring the transformation relation $\rightsquigarrow$ to be monotone with all contexts (i.e. being a congruence), but this turns out to be too strong a requirement, excluding useful transformation relations.

The sequence of steps just described forms the basic *tactical sugar* of the transformation. The tactical sugar can vary in many respects; e.g. it might contain standard proof procedures which remove trivial proof obligations, or it may use more sophisticated, semantic matching techniques [13]. Hence, a transformation rule is given by a core theorem and the tactical sugar governing its application. The same core theorem can give rise to more than one transformation rule by endowing it with different tactical sugar.

3.3 Abstraction and Reuse

A development is *closed* by resolving with the reflexivity law $?A \rightsquigarrow ?A$ which simply unifies the current result SP_i of the transformational development with the final result $?Z$. (Note that this does not imply that the specification is executable in any sense, it just allows termination of the transformational development.) Once a development is closed, we can extract a theorem from it, which can be the core theorem for a new transformation rule, where the unproven proof obligations would

appear as application conditions; hence, we can *abstract* from specific developments to more general ones.

The Transformation Application System is designed to hide the internal tactical steps, the existence of meta variables and other Isabelle technicalities etc. from the user. It comes with a graphical user interface described below. Users of the Transformation Application System will not have to worry about the details of how the transformational process is implemented within Isabelle — in fact, they will not need to have any knowledge of Isabelle at all. Since the proof of side conditions can be deferred to a later stage, users can concentrate on the main design decisions of transformational program development: which transformation to apply, and how to instantiate its parameters.

3.4 Examples for Transformation Relations and Rules

The methodology described here is generic in two respects: firstly, over the logic employed, and secondly over the transformation relation. This means that the same system (i.e. TAS) can be used for transformational program development in different formal methods, provided there exists a logical embedding of the formal method into Isabelle. Two main logical embeddings have been developed in the course of the UniForM project, for the process calculus CSP [15] and for the specification language Z [9], besides "cheap" embeddings such as transformational development in higher-order logic itself, or the Bird-Meertens calculus. Although developing a logical embedding for real-life formal methods such as these is by no means a routine task, we still believe it is worth the effort, for the benefit of being able to use it as much as for the benefit of validating the formal method itself– during the development of the formal embedding of CSP, a small but persistent error in the theory of CSP was found which went undetected for years.

The most simple transformation relation, available in all embeddings, is equality. A theorem $s = t$ then gives rise to two transformations which fold ($s \leadsto t$) or unfold ($t \leadsto s$) the equation. The equation can be as basic as commutativity of an operation, or as complex as the global search transformation given in [8].

A more sophisticated example of a transformation relation is the refinement process ordering from CSP, where a process Q refines a process P if roughly speaking Q can do everything P can and diverges less often. This refinement relation is an example of a relation which is not monotone with respect to contexts, failing for particular hiding operations.

For Z, the transformation relation could be model inclusion (i.e. $SP_1 \leadsto SP_2$ iff. $Mod(SP_1) \supseteq Mod(SP_2)$) or refinement of specifications. In both cases, the open question under investigation at the moment is the formulation of appropriate higher-level transformations, which help the user to guide the development process in the right way.

4 Tool Support

As demonstrated in Sect. 2, the Transformation Application System comes with a graphical user interface. This interface has been designed in a flexible and generic

way, allowing to quickly construct graphical user interfaces for any formal method embedded into Isabelle. In this section, we will briefly sketch how this is achieved.

4.1 System Architecture: Generic and Open

The tools are implemented with a highly generic and open system design, building entirely on well-documented, public domain systems. We will here only briefly sketch the implementation; a more detailed description can be found in [6] and [7].

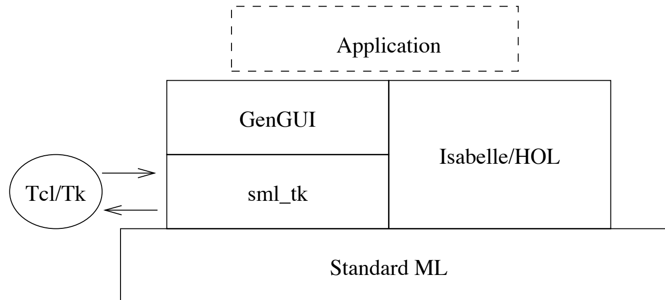

Fig. 10. System Architecture

The system is entirely implemented in Standard ML (SML) (see Fig. 10), because one can extend Isabelle conservatively by writing ML functions, using the abstract datatypes provided by Isabelle. ML's typing discipline and structuring mechanisms protect the theorem-proving core of Isabelle from being logically corrupted, and provide a closely coupled and safe (in particular, typed) interaction with Isabelle. Moreover, one can take advantage of SML's powerful structuring mechanisms to obtain a highly generic and open system architecture.

To implement the graphical user interface, we are using the interface description and command language Tcl/Tk, encapsulated into Standard ML by the sml_tk package (also developed at the University of Bremen). 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 interface in the untyped, interpretative language Tcl. Detailed information on sml_tk can be found in [11], or at the sml_tk home page (`http://www.informatik.uni-bremen.de/~cxl/sml_tk/`).

The generic graphical user interface GenGUI builds on the interface description facilities provided by sml_tk to provide a generic graphical user interface. It is implemented as a functor (a parameterised module)

```
functor GenGUI(structure appl: APPL_SIG ) = ...
```

which provides a graphical user interface for any application described by the signature `APPL_SIG`.

4.2 Visual Appearance

A consequence of this approach is that all tools obtained by instantiating GenGUI have a uniform visual appearance. Their main window always consists of two areas: the *notepad* in the upper part, and the *construction area* in the lower part. The notepad contains icons representing the objects, which can be dragged, moved and dropped onto each other, whereas the construction area allows a more refined manipulation of an object's internals.

One prominent instantiation of GenGUI is the Transformation Application System already demonstrated above.A different instantiation yields a graphical user interface for Isabelle itself, called IsaWin.

4.3 IsaWin— a Graphical User Interface for Isabelle

IsaWin is an interface to Isabelle in its own right as well as the tool to prove the proof obligations arising from transformational developments using TAS, or even the correctness of the transformations of TAS.

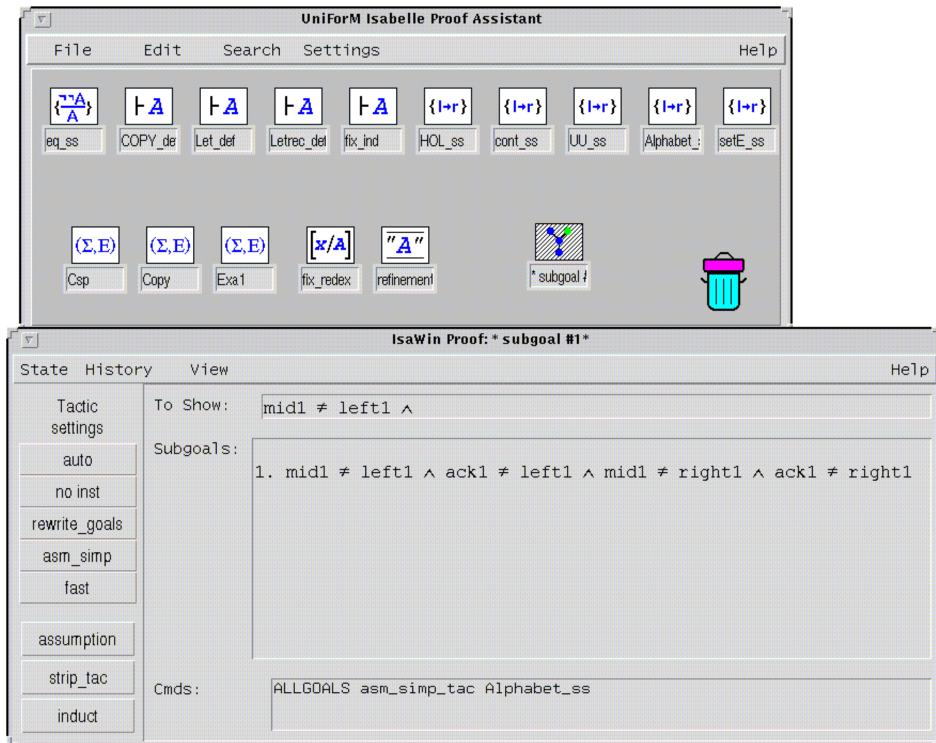

Fig. 11. IsaWin

Fig. 11 shows a screen shot of IsaWin. The icons in the notepad represent theorems, proofs, two types of rule sets, and theories (collections of type declarations, theorems and rule sets). In the construction area, proofs are carried out (here, one of the proof obligations from Sect. 2 is currently being proven); once a proof is finished, it can be turned into a theorem. The operations include backward resolution by dropping a theorem onto a proof, forward resolution by dropping a theorem onto a theorem, or rewriting by dropping a rule onto a proof.

5 Conclusions, Related and Future Work

The transformational approach to program development has a long tradition, starting from the Munich CIP Project [1]. During the PROSPECTRA project [5], a system has been implemented that enabled the formalisation of transformation rules and their use during the software development process; however, this system was severely hampered by its unstructured design and limited reasoning power, defects which we aimed to remedy by using a powerful prover and a programming language with powerful structuring concepts.

In KIDS [14], programs are developed by transforming *problem specifications* to programs. First, high-level transformations such as global search are used to transform the problem specification to an inefficient program which is then optimised by low-level transformations. In KIDS, there is no way to check the soundness of the implemented transformations. Here our approach offers a complementary aspect to KIDS since we can prove the correctness of the transformation before applying it, and discharge the resulting proof obligations in the Isabelle system.

The main emphasis during development has been put on a clear and generic system architecture rather than bells and whistles. Having achieved the former, we are going to concentrate on the latter, and are going to implement extensions such as better error handling, pretty printing using mathematical notations and focusing (applying a transformation rule or an Isabelle tactic to a subterm of the current goal, leading to the concept of a generic focus) in the near future. Further, instantiations of our framework for use with Z are currently being developed.

In summary, we have demonstrated a system which implements transformational program development in a flexible and generic way. The system can be accommodated to suit a wide variety of formal methods, provided the formal method in question provides a sufficiently rigorous mathematical foundation so it can be embedded into a theorem prover.

The main advantages of our approach, besides this considerable flexibility, are that because the logical embedding is constructed as a conservative extension of the theorem prover, it is guaranteed to be consistent, and since the transformation rules are based on theorems in this logic, they are guaranteed to be correct. Further, representing the whole transformational development as an object inside the logic offers interesting perspectives in proof and development reuse and abstraction; the development abstraction shown at the end of Sect. 2 should only be considered a first, tentative step in that direction.

References

1. F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, R. Gnatz, F. Gei selbrech tinger, E. Hansel, B. Krieg-Brückner, A. Laut, T. A. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. The Wide Spectrum Language CIP-L.* LNCS 183. Springer Verlag, 1985.
2. M. J. C. Gordon and T. M. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logics.* Cambridge University Press, 1993.
3. M. J. C. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer Verlag, 1979.
4. C. A. R. Hoare. How did software get so reliable without proof? In M. C. Gaudel and J. Woodcock, editors, *Formal Methods Europe*, LNCS 1051, pages 1–17. Springer Verlag, 1996.
5. B. Hoffmann and B. Krieg-Brückner. *Program Development by Specification and Transformation.* LNCS 690. Springer Verlag, 1993.
6. Kolyang, C. Lüth, T. Meier, and B. Wolff. Generating graphical user-interfaces in a functional setting. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 59–66. University of York, 1996.
7. Kolyang, C. Lüth, T. Meier, and B. Wolff. Generic interfaces for transformation systems and interactive theorem provers. In J. Peleska B. Berghammer, B. Buth, editor, *International Workshop on Tool Support for Validation and Verfication*, number 1 in BISS Monograph, Bremen, May 1996.
8. 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*, LNCS 1051, pages 629– 648. Springer Verlag, 1996.
9. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle. In J. von. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 283 – 298. Springer Verlag, 1996.
10. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. UniForM Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995. `http://www.informatik.uni-bremen.de/~uniform/`
11. C. Lüth, S. Westmeier, and B. Wolff. sml_tk: Functional programming for graphical user interfaces. Technical Report 7/96, Universität Bremen, 1996.
12. L. C. Paulson. *Isabelle - A Generic Theorem Prover.* Number 828 in LNCS. Springer Verlag, 1994.
13. H. Shi. A semantic matching algorithm: Analysis and implementation. In W. Penczek and A. Szalas, editors, *Proc. Mathematical Foundations of Computer Science '96*, number 1113 in LNCS, pages 517– 528, 1996.
14. D. R. Smith. KIDS — a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024– 1043, 1991.
15. 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.