RAPPORT DE RECHERCHE

L R I

ISABELLE/CIRCUS : A PROCESS SPECIFICATION AND VERIFICATION ENVIRONMENT

FELIACHI A / GAUDEL M C / WOLFF B

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud –LRI

11/2011

Rapport de Recherche N° 1547

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Isabelle/*Circus* : a Process Specification and Verification Environment

Abderrahmane Feliachi, Marie-Claude Gaudel and Burkhart Wolff

[1] Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
[2] CNRS, Orsay, F-91405, France
{Abderrahmane.Feliachi, Marie-Claude.Gaudel, Burkhart.Wolff}@lri.fr

Résumé *Circus* est un langage de spécification qui permet de spécifier des structures de données et des comportements complexes. Sa sémantique est basée sur le modèle UTP (unifying theories of programming) proposé par Hoare et He.
Nous proposons, sur la base de Isabelle/UTP, notre théorie de la sémantique de UTP en Isabelle/HOL, une sémantique formelle mécanisée, basée sur une intégration superficielle (shallow-embedding) de *Circus* en Isabelle/UTP. Nous dérivons des règles de preuve à partir de cette sémantique et mettons en oeuvre des tactiques qui permettent d'écrire des preuves de raffinement sur des processus *Circus* (impliquant à la fois des données et des comportements complexes).
Afin de faciliter son utilisation, l'environnement de preuve développé supporte une syntaxe très proche de la représentation textuelle de *Circus*.

Abstract The *Circus* specification language combines elements for complex data and behavior specifications, using an integration of Z and CSP with a refinement calculus. Its semantics is based on Hoare and He's unifying theories of programming (UTP).
Based on Isabelle/UTP, our semantic theory of UTP based on Isabelle/HOL, we develop a machine-checked, formal semantics based on a "shallow embedding" of *Circus* in Isabelle/UTP. We derive proof rules from this semantics and implement tactic support that finally allows for proofs of refinement for *Circus* processes (involving both data and behavioral aspects).
This proof environment supports a syntax for the semantic definitions which is close to textbook presentations of *Circus*.

Keywords: *Circus*, denotational semantics, Isabelle/HOL, Process Algebras, Refinement

1 Introduction

Many systems involve both complex (sometimes infinite) data structures and interactions between concurrent processes. Refinement of abstract specifications of such systems into more concrete ones, requires an appropriate formalisation of refinement and appropriate proof support.

There are several combinations of process-oriented modeling languages with data-oriented specification formalisms such as Z or B or CASL; examples are discussed in [3, 9, 16, 13]. In this report, we consider *Circus* [17], a language for refinement, that supports modeling of high-level specifications, designs, and concrete programs. It is representative of a class of languages that provide facilities to model data types, using a predicate-based notation, and patterns of interactions, without imposing architectural restrictions. It is this feature that makes it suitable for reasoning about both abstract and low-level designs.

We present a "shallow embedding" of the *Circus* semantics enabling state variables and channels in *Circus* to have arbitrary HOL types. Therefore, the entire handling of typing can be completely shifted to the (efficiently implemented) Isabelle type-checker and is therefore implicit in proofs. This drastically simplifies the definitions, proofs, and makes the reuse of standardized proof procedures possible. Compared to implementations based on a "deep embedding" such as [18] this drastically improves the usability of the resulting proof environment.

Our representation brings particular technical challenges and contributions concerning some important notions about variables. The main challenge was to represent alphabets and bindings in a typed way that preserves the semantics and improves deduction. We provide a representation of bindings without an explicit management of alphabets. Hpwever, the representation of some core concepts in the unifying theories of programming (UTP) and *Circus* constructs (variable scopes and renaming) became challenging. Thus, we propose a (stack-based) solution that allows the coding of state variables scoping with no need for renaming. This solution is even a contribution to the UTP theory that does not allow nested variable scoping. Some challenging and tricky definitions (e.g. channels and name sets) are explained in this report.

This report is organized as follows. The next section gives an introduction to the basics of our work: Isabelle/HOL, UTP and *Circus* with a short example of a *Circus* process. In section 3, we present our embedding of the basic concepts of the *Circus* language (alphabet, variables ...). We introduce also the representation of the *Circus* actions and process, with an overview of the Isabelle/*Circus* syntax. In section 4, we explain by an example, how Isabelle/*Circus* can be used to write specifications. We give some details on what is happening "behind the scenes" when the system parses each part of the specification. In the last part of this section, we show how to write proofs based on specifications, and give a refinement proof example.

2 Background

2.1 Isabelle, HOL and Isabelle/HOL

Isabelle [11] is a generic theorem prover implemented in SML. It is based on the so-called "LCF-style architecture", which makes it possible to extend a small trusted logical kernel by user-programmed procedures in a logically safe way. New object logics can be introduced to Isabelle by specifying their syntax and semantics, by deriving its inference rules from there and program specific tactic support for the object logic. Isabelle is based on a typed λ-calculus including a Haskell-style type-system including type-classes (e.g. in $\alpha :: \text{order}$, the type-variable ranges over all types that posses a partial ordering.)

Higher-order logic (HOL) [7, 1] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \Rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x \bullet P\, x$ and $\exists x \bullet P\, x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is an instance of Isabelle with higher-order logic. It provides a rich collection of library theories like sets, pairs, relations, partial functions lists, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative, i. e. logically safe definitions. Setups for the automated proof procedures like `simp`, `auto`, and arithmetic types such as `int` are provided.

2.2 Advanced Specification Constructs in Isabelle/HOL

Constant definitions. In its easiest form, constant definitions are definitional logical axioms of the form $c \equiv E$ where c is a fresh constant symbol not occurring in E which is closed (both wrt. variables and type variables). For example:

```
definition upd::(α⇒β)⇒α⇒β⇒(α⇒β)     ("_⦇_ := _⦈")
where      upd f x v ≡ λ z. if x=z then v else f z
```

The pragma `("_⦇ _ := _⦈")` for the Isabelle syntax engine introduces the notation `f⦇x:=y⦈` for `upd f x y`. Moreover, some elaborate preprocessing allows for recursive definitions, provided that a termination ordering can be established; such recursive definitions are thus internally reduced to definitional axioms.

Type definitions. Types can be introduced in Isabelle/HOL by different ways. The most general way to safely introduce new types is the type definition using `typedef` construct. This allows one to introduce a type as a non-empty subset of an existing type. More precisely, the new type is specified to be isomorphic to this non-empty subset. For instance:

```
typedef mytype = "{x::nat. x < 10}"
```

This definition requires that the set is non-empty: $\exists$`x. x`$\in$`{x::nat. x<10}`, which is easy to prove in this case:

```
by (rule_tac x = 1 in exI, simp)
```

where *`rule_tac`* is a tactic that applies an introduction rule and `exI` corresponds to the introduction of the existential quantification.

In the same way, the `datatype` command allows one to define inductive datatypes. This command introduces a datatype using a list of *constructors*. For instance, a logical compiler is invoked for the following introduction of the type `option`:

```
datatype α option  = None | Some α
```

which generates the underlying type definition and derives distinctness rules and induction principles. Besides the *constructors* `None` and `Some`, the following match-operator and his rules are also generated:

case x of None $\Rightarrow$ ... | Some a $\Rightarrow$ ...

Extensible records. Isabelle/HOL's support for *extensible records* is of particular importance for our work. Record types are denoted, for example, by:

```
record T = a::T1
           b::T2
```

which implicitly introduces the record constructor (|`a:=e1,b:=e2`|) and the update of record r in field a, written as `r`(|`a:= x`|). Extensible records are represented internally by cartesian products with an implicit free component δ, i.e. in this case by a triple of the type `T1` $\times$ `T2` $\times \delta$. The third component can be referenced by a special selector `more` available on extensible records. Thus, the record `T` can be extended later on using the syntax:

```
record ET = T + c::T3
```

The key point is that theorems can be established, once and for all, on `T` types, even if future parts of the record are not yet known, and reused in the later definition and proofs over `ET`-values. Using this feature, we can model the effect of defining the alphabet of UTP processes incrementally while maintaining the full expressivity of HOL wrt. the types of `T1`, `T2` and `T3`.

2.3 *Circus* and its UTP Foundation

Circus is a formal specification language [17] which integrates the notions of states and complex data types (in a Z-like style) and communicating parallel processes inspired from CSP. From Z, the language inherits the notion of a schema used to model sets of (ground) states as well as syntactic machinery to describe pre-states and post-states; from CSP, the language inherits the concept of *communication events* and typed communication channels, the concepts of deterministic and non-deterministic choice (reflected by the process combinators $P \,\Box\, P'$ and $P \sqcap P'$), the concept of concealment (hiding) $P \backslash A$ of events in A occurring in in the evolution of process P. Due to the presence of state variables, the *Circus* synchronous communication operator syntax is slightly different frome CSP: $P \;[\![\; n \mid c \mid n' \;]\!] P'$ means that P and P' communicate via the channels mentioned in c; moreover, P may modify the variables mentioned in n only, and P' in n' only, n and n' are disjoint name sets.

Moreover, the language comes with a formal notion of refinement based on a denotational semantics. It follows the failure/divergence semantics [14], (but coined in terms of the UTP [12]) providing a notion of execution trace `tr`, refusals `ref`, and divergences. It is expressed in terms of the UTP [10] which makes it amenable to other refinement-notions in UTP. The semantics allows for a rich set of algebraic rules for specifications and their transitions to program models.

A simple *Circus* specification is `FIG`, the fresh identifiers generator given in fig. 1:

$[ID]$

channel req
channel $ret, out : ID$

process $FIG \mathrel{\widehat=}$ **begin**
state $S == [\, idS : \mathbb{P}\, ID \,]$
$Init \mathrel{\widehat=} idS := \emptyset$

Out

ΔS
$v! : ID$

$v! \notin idS$
$idS' = idS \cup \{v!\}$

Remove

ΔS
$x? : ID$

$idS' = idS \setminus \{x?\}$

• $Init$; **var** $v : ID$ •
$(\mu\ X \bullet (req \rightarrow Out\ ;\ out!v \rightarrow Skip\ \Box\ ret?x \rightarrow Remove)\ ;\ X)$

end

Fig. 1. The Fresh Identifiers Generator in (Textbook) *Circus*

Predicates and Relations. The UTP is a semantic framework based on an alphabetized relational calculus. An *alphabetized predicate* is a pair (*alphabet*, *predicate*) where the free variables appearing in the predicate are all in the alphabet, e.g. $(\{x, y\}, x > y)$. As such, it is very similar to the concept of a *schema*

in Z. In the base theory Isabelle/UTP of this work, we represent alphabetized predicates by sets of (extensible) records, e.g. `{A. x A > y A}`.

An *alphabetized relation* is an alphabetized predicate where the alphabet is composed of input (undecorated) and output (dashed) variables. In this case the predicate describes a relation between input and output variables, for example $(\{x, x', y, y'\}, x' = x + y)$ which is a notation for: `{(A,A').x A' = x A + y A}`, which is a set of pairs, thus a relation.

Standard predicate calculus operators are used to combine alphabetized predicates. The definition of these operators is very similar to the standard one, with some additional constraints on the alphabets.

Designs and processes. In UTP, in order to explicitly record the termination of a program, a subset of alphabetized relations is introduced. These relations are called *designs* and their alphabet should contain the special boolean observational variable `ok`. It is used to record the start and termination of a program. A UTP design is defined as follows in Isabelle:

$$\texttt{(P} \vdash \texttt{Q)} \equiv \lambda\ \texttt{(A,A').}\ \texttt{(ok A} \wedge \texttt{P (A,A'))} \longrightarrow \texttt{(ok A'} \wedge \texttt{Q (A,A'))}$$

Following the way of UTP to describe reactive processes, more observational variables are needed to record the interaction with the environment. Three observational variables are defined for this subset of relations: `wait`, `tr` and `ref`. The boolean variable `wait` records if the process is waiting for an interaction or has terminated. `tr` records the list (trace) of interactions the process has performed so far. The variable `ref` contains the set of interactions (events) the process may refuse to perform. These observational variables defines the basic alphabet of all reactive processes called "`alpha_rp`".

Some healthiness conditions are defined over `wait`, `tr` and `ref` to ensure that a process satisfies some properties [5] (see table 2). Four healthiness conditions, $H1$ to $H4$, are defined to characterize designs and three other ones, $R1$, $R2$ and $R3$, for reactive processes.

Finally, a CSP process is a UTP reactive process that satisfies two additional healthiness conditions called $CSP1$ and $CSP2$ (all well-formedness conditions are summarized in table 2). A process that satisfies $CSP1$ and $CSP2$ is said to be CSP healthy.

3 Isabelle/*Circus*

The Isabelle/*Circus* environment allows the representation of processes in a syntax which is close to the textbook presentations of *Circus* (see Fig. 2). Similar to other specification constructs in Isabelle/HOL, this syntax is "parsed away", i. e. compiled into an internal representation of the denotational semantics of *Circus*, which is a formalization in form of a shallow embedding of the (essentially untyped) paper-and-pencil definitions by Oliveira et al. [12], based on UTP. *Circus* actions are defined as CSP healthy reactive processes.

$H1$: A design may not make any prediction on variable values until the program has started.
`P = λ(A,A'). ok A →P (A,A')`
$H2$: A design may not require non-termination.
`P(A,A'(|ok:=False|))→P(A,A'(|ok:=True|))`
$H3$: If the precondition of a design is satisfiable, its postcondition must be satisfiable too.
`P = P ;; Π`
$H4$: Exclude miracle design.
`P ;; true = true`
$R1$: The execution of a reactive process never undoes any event that has already been performed.
`P = P ∧ λ (A,A'). tr A ≤ tr A'`
$R2$: The behaviour of a reactive process is oblivious to what has gone before.
`P = λ(A,A'). P(A(|tr:=[]|),A'(|tr:=(tr A' - tr A)|))`
$R3$: Intermediate stable states do not progress.
`P = Πrea ◁ wait o fst ▷ P`
$CSP1$: Extension of the trace is the only guarantee on divergence.
`P = P ∨ (λ (A,A'). ¬ ok A ∧ tr A ≤ tr A')`
$CSP2$: A process may not require non-termination.
`P = P ;; J`
`J = λ(A,A'). (ok A →ok A') ∧ tr A' = tr A ∧ wait A' = wait A`
`∧ ref A' = ref A ∧ more A' = more A`

Where:

- `;;` is the sequential composition operator over relations,
- Π is the relational Skip,
- Π`rea` is the Skip reactive process,
- `◁ ▷` is the conditional operator over relations,
- `o` is the HOL functional composition operator,
- `fst` returns the first element of a pair.

Table 2. UTP Healthiness conditions

In the UTP representation of reactive processes we have given in a previous paper [8], the process type is generic. It contains two type parameters that represent the channel type and the alphabet of the process. These parameters are very general, and they are instantiated for each specific process. This could be problematic when representing the *Circus* semantics, since some definitions rely directly on variables and channels (e.g assignment and communication). In this section we present our solution to deal with this kind of problems, and our representation of the *Circus* actions and processes.

In the following, we describe the foundation as well as the semantic definition of the process operators of *Circus*. A distinguishing feature of *Circus* processes are explicit state variables which do not exist in other process algebras like, e.g., CSP. These can be:

- *global* state variables, i. e. they are declared via alphabetized predicates in the `state` section, or Z-like Δ operations on global states that generate alphabetized relations, or

Process ::= **circusprocess** Tpar* name = PParagraph* **where** Action
PParagraph ::= AlphabetP | StateP | ChannelP | NamesetP | ChansetP | SchemaP
| ActionP
AlphabetP ::= **alphabet** [vardecl$^+$]
vardecl ::= name :: type
StateP ::= **state** [vardecl$^+$]
ChannelP ::= **channel** [chandecl$^+$]
chandecl ::= name | name type
NamesetP ::= **nameset** name = [name$^+$]
ChansetP ::= **chanset** name = [name$^+$]
SchemaP ::= **schema** name = SchemaExpression
ActionP ::= **action** name = Action
Action ::= **Skip** | **Stop** | Action ; Action | Action □ Action | Action ⊓ Action
| Action \ chansetN | var := expr | guard & Action | comm → Action
| **Schema** name | ActionName | μ var • Action | **var** var • Action
| Action ⟦ namesetN | chansetN | namesetN ⟧ Action

Fig. 2. Isabelle/*Circus* syntax

– *local* state variables, i. e. they are result of the variable declaration statement **var** var • Action. The scope of local variables is restricted to Action.

On both kind of state variables, logical constraints may be expressed.

3.1 Alphabets and Variables

In order to define the set of variables, the *Circus* semantics describes the alphabet of its components, be it on the level of alphabetized predicates, alphabetized relations or actions. We recall that these items are represented by sets of records or sets of pairs of records, following the idea that an alphabet is used to establish a "binding" of variables to values. The *alphabet of a process* is defined by extending the reactive process alphabet (cf. Section 2.3) with the corresponding variable names and types. Considering the example *FIG*, where the global state variable *idS* is defined, this is reflected in Isabelle/Circus by the extension of the process alphabet by this variable, i.e. by the extension of the Isabelle/HOL record:

```
record α alpha = α alpha_rp +  idS :: ID set
```

This introduces the record type `alpha` that contains the observational variables of a reactive process, plus the variable `idS`. Note that our *Circus* semantic representation allows "built-in" bindings of alphabets in a typed way. Moreover, there is no restriction on the associated HOL type. However, the inconvenience of this representation is that variables cannot be introduced "on the fly"; they must be known statically i.e. at type inference time. Another consequence is that a "syntactic" operation such as variable renaming has to be expressed as a "semantic" operation that maps one record type into another.

Updating and accessing global variables. Since the alphabets are represented by HOL records, i.e. a kind binding "*name* ↦ *value*", we need a certain

infrastructure to access data in them and to update them. The Isabelle representation as records gives us already two functions (for each record) "select" and "update". The "select" function returns the value of a given variable name, and the "update" functions updates the value of this variable. Since we may have different HOL types for different variables, a unique definition for select and update cannot be provided. There is an instance of these functions for each variable in the record. The name of the variable is used to distinguish the different instances: for the select function the name is used directly and for the update function the name is used as a prefix e.g. for a variable named "x" the names of the *select* and *update* functions are respectively `x` of type α and `x_update`.

Since a variable is characterized essentially by these functions, we define a general type (synonym) called `var` which represents a variable as a pair of its select and update function (in the underlying state σ).

```
types (β, σ) var = "(σ ⇒ β) * ((β ⇒ β) ⇒ σ ⇒ σ)"
```

For a given alphabet (record) of type σ, `(`β`, the type `σ`)var` represents the type of the variables whose value type is β in this alphabet. One can then extract the select and update functions from a given variable with the following functions:

```
definition select :: "(β, σ) var ⇒σ ⇒ β"
  where select f ≡ (fst f)

definition update :: "(β, σ) var ⇒ β ⇒ σ ⇒ σ"
  where update f v ≡ (snd f) (λ _ . v)
```

Finally, we introduce a function called `VAR` to implement a syntactic translation of a variable name to an entity of type `var`.

```
syntax "_VAR" :: "id ⇒(β, σ) var" ("VAR _")
translations VAR x => (x, _update_ name x)
```

Note that in this syntactic translation rule, `_update_ name x` stands for the concatenation of the string `_update_` with the content of the variable `x`; the resulting `_update_x` in this example is mapped to the field-update function of the extensible record `x_update` by a default mechanism. On this basis, the assignment notation can be written as usual:

```
syntax
  "_assign" :: "id ⇒(σ ⇒ β) ⇒(α, σ) action" ("_ ‘:=‘ _")
translations
  "x ‘:=‘ E"  => "CONST ASSIGN (VAR x) E"
```

and mapped to the *semantics* of the program variable `(x,x_update)` together with the universal `ASSIGN` operator defined in Section 3.3.

Updating and accessing local variables. In *Circus*, local program variables can be introduced on the fly, and their scopes are explicitly defined, as can

be seen in the *FIG* example. In textbook *Circus*, nested scopes are handled by variable renaming which is not possible in our representation due to the implicit representation of variable names. Instead, we represent local program variables by global variables, using the `var` type defined above, where selection and update involve an explicit stack discipline. Each variable is mapped to a list of values, and not to one value only (as for state variables). Entering the scope of a variable corresponds to adding a new value as the head of the corresponding values list. Leaving a variable scope corresponds to removing the head of the values list. The select and update functions correspond to selecting and updating the head of the list.

Note that this encoding scheme requires to make local variables lexically distinct from global variables; local variable instances are just distinguished from the global ones by the stack discipline. The results in dynamic scoping which is required by the operational semantics.

3.2 Synchronization infrastructure: Name sets and channels.

Name sets. An important notion, used in the definition of parallel *Circus* actions, is name sets as seen in Section 2. A name set is a set of variable names, which is a subset of the alphabet. This notion cannot be directly expressed in our representation since variable names are not explicitly represented. Its definition is a bit tricky and relies on the characterization of the variables in our representation. As for variables, name sets are defined by their functional characterization. Name sets are only used in the definition of the binding merge function *MSt*: $\forall\, v \bullet (v \in ns1 \Rightarrow v' = (1.v)) \land (v \in ns2 \Rightarrow v' = (2.v)) \land (v \notin ns1 \cup ns2 \Rightarrow v' = v)$.

The disjoint name sets $ns1$ and $ns2$ are used to determine which variable values (extracted from local bindings of the parallel components) are used to update the global binding of the process. A name set can be functionally defined as a binding update function, that copies values from a local binding to the global one. For example, a name set *NS* that only contains the variable x can be defined as follows in Isabelle/Circus:

```
definition NS lb gb ≡ x_update (x lb) gb
```

where `lb` and `gb` stands for local and global bindings, `x` and `x_update` are the select and update functions of variable `x`. Then the merge function can be defined by composing the application of the name sets to the global binding.

Channels. Reactive processes interact with the environment via synchronizations and communications. A synchronization is an interaction via a channel without any exchange of data. A communication is a synchronization with data exchange. In order to reason about communications in the same way, a datatype *channels* is defined using the channels names as constructors. For instance, in:

```
datatype channels = chan1 | chan2 nat | chan3 bool
```

we declare three channels: `chan1` that synchronizes without data , `chan2` that communicates natural values and `chan3` that exchanges boolean values.

This definition makes it possible to reason globally about communications since they have the same type. A drawback is that the channels may not have the same type: in the above example the types of `chan1`, `chan2` and `chan3` are respectively of types `channels`, `nat` ⇒`channels` and `bool` ⇒`channels`. However, in the definition of some *Circus* operators, we need to compare two channels, and one can't compare for example `chan1` with `chan2` since they don't have the same type. A solution would be to compare `chan1` with (`chan2 v`). The types are equivalent in this case, but the problem remains because comparing (`chan2 0`) to (`chan2 1`) will state inequality just because the communicated values are not equal. We can of course define an inductive function over the datatype `channels` to compare channels, but this is only possible when all the channels are known *a priori*.

Thus, we add some constraint to the generic channels type: we require the `channels` type to implement a function `chan_eq` that tests the equality of two channels. Fortunately, Isabelle/HOL provides a construct for this kind of restriction: the type classes (sorts) seen in the first section. We define a type class (interface) `chan_eq` that contains a signature of the `chan_eq` function.

```
class chan_eq =
  fixes chan_eq :: "α ⇒α ⇒ bool"
begin end
```

Concrete channels type must implement the interface (class) " `chan_eq`" that can be easily defined for this concrete type. Moreover, one can use this class to add some definition that depends on the channel equivalence function. For example, a trace equivalence function can be defined as follows:

```
fun tr_eq where
  tr_eq [] [] = True | tr_eq xs [] = False | tr_eq [] ys = False
| tr_eq (x#xs) (y#ys) = if chan_eq x y then tr_eq xs ys else False
```

This function will be applicable for traces of elements whose type belongs to the sort `chan_eq`.

3.3 Actions and Processes

The *Circus* actions type is defined as the set of all the CSP healthy reactive processes. The type `(α,σ)relation_rp` is the reactive process type where α is of `channels` type and σ is a record extensions of `action_rp`, i. e. the global state variables. On this basis, we can encode the concept of a process for a family of possible state instances. We introduce the vital type `action` via the type-definition:

```
typedef(Action)
 (α::chan_eq,σ) action = {p::(α,σ)relation_rp. is_CSP_process p}
proof - {...} qed
```

As mentioned before, a type-definition introduces a new type by stating a set. In our case it is the set of reactive processes that satisfy the healthiness-conditions for CSP-processes, isomorphic to the new type.

Technically, this construct introduces two constants definitions `Abs_Action` and `Rep_Action` respectively of type `(α,σ) relation_rp ⇒(α,σ) action` and `(α,σ)action ⇒(α,σ)relation_rp` as well as the usual two axioms expressing the bijection `Abs_Action(Rep_Action(X))=X` and `is_CSP_process p ⟹ Rep_Action(Abs_Action(p))=p` where `is_CSP_process` captures the healthiness conditions.

Every *Circus* action is an abstraction of an alphabetized predicate. Below, we introduce the definitions of all the actions and operators using their denotational semantics. We must provide for each action, the proof that this predicate is CSP healthy. In this section we show all *Circus* basic actions and operators definitions. We also show how a whole *Circus* process is represented in the UTP framework. The environment contains the definitions of all the *Circus* operators shown in the next section.

Moreover, the environment contains a proof for a theorem stating that every reactive design — based on the above and the subsequent definitions — is CSP healthy.

Basic actions. `Stop` is defined as a reactive design, with a precondition `true` and a postcondition stating that the system deadlocks and the traces are not evolving.

```
definition
Stop ≡  Abs_Action (R (true ⊢λ(A, A'). tr A' = tr A ∧ wait A'))
```

`Skip` is defined as a reactive design, with a precondition *true* and a postcondition stating that the system terminates and all the state variables are not changed. We represent this fact by stating that the `more` field is not changed, since this field is mapped to all the state variables. Recall that the `more`-field is a tribute to our encoding of alphabets by extensible records and stands for all future extensions of the alphabet (e.g. state variables).

```
definition Skip ≡  Abs_Action (R (true ⊢ λ (A, A'). tr A' = tr A
                                    ∧ ¬ wait A' ∧ more A = more A'))
```

The universal assignment action. In the previous section 3.1, we described already how global and local variables were represented by access- and updates functions introduced by fields in extensible records. In these terms, the "lifting" to the assignment action in *Circus* processes is straightforward:

```
definition
  ASSIGN::"(β, σ) var ⇒(σ ⇒ β) ⇒(α::ev_eq, σ) action"
where
  ASSIGN x e ≡ Abs_Action (R (true ⊢ Y))
```

```
where
 Y = λ(A, A'). tr A' = tr A ∧ ¬ wait A' ∧
                more A' = (assign x (e (more A))) (more A)
```

where `assign` is the projection into the update operation of a semantic variable described in section 3.1.

Internal and External Choice. For the internal choice operator the semantics is quite simple. It is defined as the relational disjunction of the two actions.

```
definition ndet (infixl "⊓") where
P ⊓ Q ≡  Abs_Action ((Rep_Action P ) ∨(Rep_Action Q))
```

The external choice semantics is more complicated, it is defined in our environment as follows:

```
definition det (infixl "□") where
P □ Q ≡  Abs_Action(R (X ⊢ Y))
where
X =  ¬(Spec F F (Rep_Action P)) ∧¬ (Spec F F (Rep_Action Q))
and
Y = (Spec T F (Rep_Action P)) ∧ (Spec T F (Rep_Action Q))
          ◁ λ (A, A'). tr A = tr A' ∧ wait A' ▷
    (Spec T F (Rep_Action P)) ∨ (Spec T F (Rep_Action Q))
```

where the operation *Spec* is defined as follows:

```
definition Spec x y P = λ(A, A'). P (A(|wait := y|), A'(|ok := x|))
```

Guarded Actions. A guarded action is an action that can be executed if the guard value is *true* only, otherwise it stops. A guard is defined as a predicate over the action variables and the semantics of the guarded action is defined as follows:

```
definition guard ("_ & _") where
g & P ≡  Abs_Action(R (X ⊢ Y))
where
X = (g o more o fst) ⟶¬ (Spec F F (Rep_Action P))
and
Y = ((g o more o fst) ∧ (Spec T F (Rep_Action P))) ∨
    (¬ (g o more o fst) ∧ (λ (A, A'). tr A' = tr A ∧ wait A'))
```

Given this definition, it is easy to prove that if the value of the guard is *false* the action will stop. The proof of this property is given by the following:

```
lemma "false & P = Stop"
by (simp add: Guard_def Stop_def csp_defs design_defs utp_defs
                                                        rp_defs)
```

Sequencing. The sequence operator is defined using the UTP sequential composition operator: the semantics of composing two actions is given by the relational composition of their corresponding relations.

```
definition seq (infixl ";") where
P ; Q ≡ Abs_Action (Rep_Action P ;; Rep_Action Q)
```

Schema Expressions. In order to define the semantics of schema expressions, the function `Pre` is introduced. This function verifies that the precondition of a schema expression is *true* before applying the schema operation. This function is defined by ignoring the output alphabet of the schema, since the precondition depends only on the input variables.

```
definition Pre ::"'α relation ⇒'α predicate" where
Pre sc ≡ λ A. ∃ A'. sc (A, A')
```

The semantics of schema expressions is then:

```
definition
Schema sc ≡ Abs_Action(R (X ⊢ Y))
where
X = λ(A, A'). (Pre sc) (more A)
and
Y = λ(A, A'). sc (more A, more A') ∧ ¬ wait A' ∧ tr A = tr A'
```

Communications. The definition of prefixed actions is based on the definition of a special relation `do_I`. In the *Circus* denotational semantics, various forms of prefixing were defined. In our theory, we define one general form, and the other forms are defined as special cases.

```
definition do_I c x P ≡  X  ◁ wait o fst ▷  Y
where
X = (λ (A, A'). tr A = tr A' ∧ ((c ‘ P) ∩ ref A') = {})
and
Y = (λ (A, A'). hd ((tr A') - (tr A)) ∈ (c ‘ P) ∧
     (c (select x (more A))) = (last (tr A')))
```

where `c` is a channel constructor, `x` is a variable (of `var` type) and `P` is a predicate. The `do_I` relation gives the semantics of an interaction: if the system is ready to interact, the trace is unchanged and the waiting channel is not refused. After performing the interaction, the new event in the trace corresponds to this interaction.

The semantics of the whole action is given by the following definition:

```
definition Prefix c x P S ≡Abs_Action(R (X ⊢ Y)) ; S
where
X = true
and
Y =  do_I c x P ∧ (λ (A, A'). more A' = more A)
```

where c is a channel constructor, x is a variable (of type var), P is a predicate and S is an action. This definition states that the prefixed action semantics is given by the interaction semantics (do_I) composed with the semantics of the continuation (action S).

Different types of communication are considered:

- Inputs: the communication is done over a variable.
- Constrained Inputs: the input variable value is constrained with a predicate.
- Outputs: the communications exchanges only one value.
- Synchronizations: only the channel name is considered (no data).

The semantics of these different forms of communications is based on the general definition above.

```
definition read c x P ≡ Prefix c x true P
definition write1 c a P ≡ Prefix c (λs. a s, (λ x. λy. y)) true P
definition write0 c P ≡ Prefix (λ_.c) (λ_._, (λ x. λy. y)) true P
```

where read, write1 and write0 corresponds respectively to *input*, *output* and *synchronization*, *constrained input* corresponds to the general definition.

We configure the Isabelle syntax-engine such that it parses the usual communication primitives and gives the corresponding semantics:

```
translations
  c ? p →P      == CONST read c (VAR p) P
  c ? p : b →P  == CONST Prefix c (VAR p) b P
  c ! p →P      == CONST write1 c p P
  a → P         == CONST write0 (TYPE(_)) a P
```

Hiding. The hiding operator is interesting because it depends on a channel set. This operator `P \ cs` is used to encapsulate the events that are in the channel set cs. These events become no longer visible from the environment. The semantics of the hiding operator is given by the following reactive process:

```
definition
Hide ::"[(α, σ) action , α set] ⇒(α, σ) action" (infixl "\")
where
P \ cs ≡ Abs_Action( R(λ (A, A').
          ∃ s. (Rep_Action P)(A, A'(|tr :=s, ref := (ref A') ∪ cs|))
             ∧ (tr A' - tr A) = (tr_filter (s - tr A) cs))); Skip
```

The definition uses a filtering function tr_filter that removes from a trace the events whose channels belong to a given set. The definition of this function is based on the function chan_eq we defined in the class chan_eq. This explains the presence of the constraint on the type of the action channels in the hiding definition, and in the definition of the filtering function below:

```
fun tr_filter::"a::chan_eq list ⇒a set ⇒a list" where
  tr_filter [] cs = []
```

```
| tr_filter (x#xs) cs = (if (¬ chan-in_set x cs)
                           then (x#(tr_filter xs cs))
                            else (tr_filter xs cs))
```

where the `chan-in_set` function checks if a given channel belongs to a channel set using `chan_eq` as equality function.

Parallel Composition. The parallel composition of actions is one of the most important definitions in our environment. It involves two important notions presented in the last section which are name sets and channel sets. As explained in Sections 2 and 3.2, name sets are used in the state merge function *MSt* that merges the final values of the local states into the global one. The name sets are defined as state update functions that can be composed to build the global state by the *MSt* function. On this basis, we define the *MSt* function in Isabelle/Circus as follows:

```
definition
MSt s1 s2 = (λ (S, S'). (S' = s1 S)) ;; (λ (S, S'). (S' = s2 S))
```

where `s1` and `s2` are disjoint name sets.

The second important notion in this definition is channel set. As explained in section 3.2, channels are defined as datatype constructors. As channels are usually defined over different types, channel sets cannot be directly defined since the types of elements may be not the same (as explained in section 3.2). To avoid this problem, we use the communications type `channels` as the type of elements in the channel sets. Thus, in the case of constructors communicating values, we apply them to some dummy values to obtain a value of type `channels`. One can define, for instance, the channel set `cs = {chan1, chan2(Some x)}`, and define a new membership function over this channel set using the function `chan_eq` to check if a channel belongs to some given channel set.

Given these definitions, the parallel composition operator is stated as follows:

```
definition Par ("_ ⟦_ | _ | _ ⟧_") where
A1 ⟦ ns1 | cs | ns2 ⟧ A2 ≡  Abs_Action(R (X ⊢ Y))
where
X = (λ (S, S'). ¬∃tr1 tr2. (Spec F F (Rep_Action A1) ;;
(λ (S, S'). tr1 = tr S)) (S, S') ∧(Spec F (wait S) (Rep_Action A2) ;;
(λ (S, S'). tr2 = tr S)) (S, S') ∧(tr_filter tr1 cs) = (tr_filter tr2 cs)
 ∧ ¬ ∃ tr1 tr2. (Spec F (wait S) (Rep_Action A1) ;;
(λ (S, S'). tr1 = tr S)) (S, S') ∧(Spec F F (Rep_Action A2) ;;
(λ (S, S'). tr2 = tr S)) (S, S') ∧(tr_filter tr1 cs) = (tr_filter tr2 cs))
and
Y = (λ (S, S'). (∃ s1 s2. (λ (A, A'). (Spec T F (Rep_Action A1) (A, s1))
   ∧ Spec T F (Rep_Action A2) (A, s2)) ;; M_par s1 ns1 s2 ns2 cs) (S, S'))
```

where `A1` and `A1` are *Circus* actions, `ns1` and `ns2` are name sets defined as update functions over the state of the actions `A1` and `A2`. Finally, `cs` is a channel set defined over the communications type of the actions `A1` and `A2`.

The environment contains also the definitions of `tr_filter`, `M_par` and some other functions used in these definitions.

Recursion. To represent the recursion operator "μ" over actions, we use the universal least fix-point operator "*lfp*" defined in the HOL library for lattices and we follow again [12]. The use of least fix-points in [12] is the most substantial deviation from the standard CSP denotational semantics, which requires Scott-domains and complete partial orderings. The operator *lfp* is inherited from the "*Complete Lattice class*" under some conditions, and all theorems defined over this operator can be reused. In order to reuse this operator, we have to show that the least-fixpoint over functionals that enrich pairs of failure - and divergence trace sets monotonely, produces an `action` that satisfies the CSP healthiness conditions. This consistency proof for the recursion operator is the largest contained in the Isabelle/*Circus* library.

In order to reuse the *lfp* operator and its inherited proofs, we must prove that the *Circus* actions type defines a complete lattice. This leads to prove that the actions type belongs to the "*Complete Lattice class*" of HOL. Since type classes in HOL are hierarchic, we provide a proof in three steps. First, we prove that the *Circus* actions type forms a lattice by instantiating the HOL "*Lattice class*". In the second step, we prove that actions type instantiates a subclass of lattices called "*Bounded Lattice class*". The last step is to prove the instantiation from the "*Complete Lattice class*". The details of these proofs are not given here.

```
instantiation action :: (ev_eq, type) lattice
begin
definition inf_action:
    inf P Q ≡ Abs_Action ((Rep_Action P) ⊓ (Rep_Action Q))
definition sup_action:
    sup P Q ≡ Abs_Action ((Rep_Action P) ⊔ (Rep_Action Q))
definition leq_action:
    P ≤ Q ≡ P ⊑ Q
definition less_action:
    P < Q ≡ P ⊑ Q ∧ ¬ Q ⊑ P
instance proof
  {...}
end
```

A lattice is a partial order with infimum and supremum of any two actions, the $\sqcap$ (meet) and $\sqcup$ (join) operations select such infimum and supremum actions. The instantiation proof of the lattice class requires the introduction of the definitions of the *meet*, the *join* and the ordering operators $\leq$ and $<$. In addition to the definition, the instantiation provides some proof obligations to ensure that all the notions are well defined (e.g. the ordering relation is reflexive and transitive). After proving these properties, the action type is considered as a lattice.

```
instantiation action :: (ev_eq, type) bounded_lattice
begin
definition bot_action:
    bot ≡ Abs_Action true
definition top_action:
```

```
    top ≡ Abs_Action (R(true ⊢false))
instance proof
  {...}
end
```

For the instantiation of the bounded lattice class, we add definitions of bounds (top and bottom of the lattice) and prove that these bounds are well defined w.r.t the ordering relation.

```
instantiation action :: (ev_eq, type) complete_lattice
begin
definition Sup_action:
    Sup S ≡ if S={} then bot else Abs_Action ⨆(Rep_Action ‘S)
definition Inf_action:
    Inf S ≡ if S={} then top else Abs_Action ⨅(Rep_Action ‘S)
instance proof
  {...}
end
```

Finally, a complete lattice is a partial order with general (infinitary) infimum of any set of actions, a general supremum exists as well. The general $\bigsqcap$ (meet) and $\bigsqcup$ (join) operations select such infimum and supremum actions. These operations indeed determine bounds on this complete lattice structure. The Knaster-Tarski Theorem (in its simplest formulation) states that any monotone function on a complete lattice has a least fixed-point. This is a consequence of the basic boundary properties of the complete lattice operations. Instantiating the complete lattice class allows one to inherit these properties with the definition of the least fixed-point for monotonic functions over *Circus* actions.

Circus **Processes.** A *Circus* process is defined in our environment as a local theory by introducing qualified names for all its components. This is very similar to the notion of *namespaces* popular in programming languages. Defining a *Circus* process locally allows us to encapsulate definitions of alphabet, channels, schema expressions and actions in the same namespace. It is important for the foundation of Isabelle/*Circus* to avoid the ambiguity between local process entities definitions (e.g. `FIG.Out` and `DFIG.Out` in the example of section 4).

4 Using Isabelle/*Circus*

We describe the front-end interface of Isabelle/*Circus*. In order to support a maximum of common *Circus* syntactic look-and-feel, we have programmed at the SML level of Isabelle a compiler that parses and (partially) pretty prints *Circus* process given in the syntax presented in Fig.2.

4.1 Writing specifications

A specification is a sequence of paragraphs. Each paragraph may be a declaration of alphabet, state, channels, name sets, channel sets, schema expressions or

actions. The main action is introduced by the keyword `where`. Below, we illustrate how to use the environment to write a *Circus* specification using the `FIG` process example presented in Figure 1.

```
circusprocess FIG =
  alphabet = [v::nat, x::nat]
  state = [idS::nat set]
  channel = [req, ret nat, out nat]
  schema Init = idS := {}
  schema Out = ∃a. v' = a ∧ v' ∉ idS ∧ idS' = idS ∪{v'}
  schema Remove = x ∉ idS ∧ idS' = idS - {x}
  where var v · Schema Init; (μ X ·(req →Schema Out; out!v →Skip)
                                □ (ret?x →Schema Remove); X)
```

Each line of the specification is translated into the corresponding semantic operators discussed in the previous chapter. In the following, we describe the result of executing each command:

- the compiler introduces a scope of local components whose names are qualified by the process name (`FIG` in the example).
- `alphabet` generates a list of record fields to represent the binding. These fields map names to value lists.
- `state` generates a list of record fields that corresponds to the state variables. The names are mapped to single values. This command, together with `alphabet` command, generates a record that represents all the variables (for the `FIG` example the command generates the record `FIG_alphabet`, that contains the fields `v` and `x` of type `nat list` and the field `idS` of type `nat set`).
- `channel` introduces a datatype of typed communication channels (for the `FIG` example the command generates the datatype `FIG_channels` that contains the constructors `req` without communicated value and `ret` and `out` that communicate natural values).
- `schema` allows the definition of schema expressions represented as an alphabetized relation over the process variables (in the example the schema expressions `FIG.Init`, `FIG.Out` and `FIG.Remove` are generated).
- `action` introduces definitions for *Circus* actions in the process. These definitions are based on the denotational semantics of *Circus* actions. The type parameters of the action type are instantiated with the locally defined channels and alphabet types.
- `where` introduces the main action as in `action` command (in the example the main action is `FIG.FIG` of type `(FIG_channels, FIG_alphabet)action`).

4.2 Relational and Functional Refinement in Circus

The main goal of Isabelle/*Circus* is to provide a proof environment for *Circus* processes. The "shallow-embedding" of *Circus* and UTP in Isabelle/HOL offers the possibility to reuse proof procedures, infrastructure and theorem libraries

already existing in Isabelle/HOL. Moreover, once a process specification is encoded and parsed in Isabelle/*Circus*, proofs of, eg, refinement properties can be developped using the ISAR language for structured proofs.

To show in more details how to use Isabelle/*Circus*, we provide a small example of action refinement proof. The refinement relation is defined as the universal reverse implication in the UTP. In *Circus*, it is defined as follows:

```
definition A1 ⊑c A2 ≡(Rep_Action A1) ⊑utp (Rep_Action A2)
```

where A1 and A2 are *Circus* actions, ⊑c and ⊑utp stands respectively for refinement relation on *Circus* actions and on UTP predicate.

This definition assumes that the actions A1 and A2 share the same alphabet (binding) and the same channels. In general, refinement involves an important data evolution and growth. The data refinement is defined in [15, 4] by backwards and forwards simulations. In this report, we restrict ourselves to a special case, the so-called *functional* backwards simulation. This refers to the fact that the abstraction relation R that relates concrete and abstract actions is just a function:

```
definition Simulation ("_ ≼_ _") where
 A1 ≼R A2 = ∀a b.(Rep_Action A2)(a,b) ⟶(Rep_Action A1)(R a,R b)
```

where A1 and A2 are *Circus* actions and R is a function mapping the corresponding A1 alphabet to the A2 alphabet.

4.3 Refinement Proofs

We can use the definition of simulation to transform the proof of refinement to a simple proof of implication by unfolding the operators in terms of their underlying relational semantics. The problem with this approach is that the size of proofs will grow exponentially with respect to the size of the processes. To avoid this problem, some general refinement laws were defined in [4] to deal with the refinement of *Circus* actions at operators level and not at UTP level. We introduced and proved a subset of theses laws in our environment (see table 3).

In table 3, the relations "$x \sim_S y$" and "$g_1 \simeq_S g_2$" record the fact that the variable x (repectively the guard g_1) is refined by the variable y (repectively by the guard g_2) w.r.t the simulation function S.

These laws can be used in complex refinement proofs to simplify them at the *Circus* level. More rules can be defined and proved to deal with more complicated statements like combination of operators for example. Using these laws, and exploiting the advantages of a shallow embedding, the automated proof of refinement becomes surprisingly simple.

Coming back to our example, let us consider the DFIG specification below, where the management of the identifiers via the set idS is refined into a set of removed identifiers retidS and a number max, which is the rank of the last issued identifier.

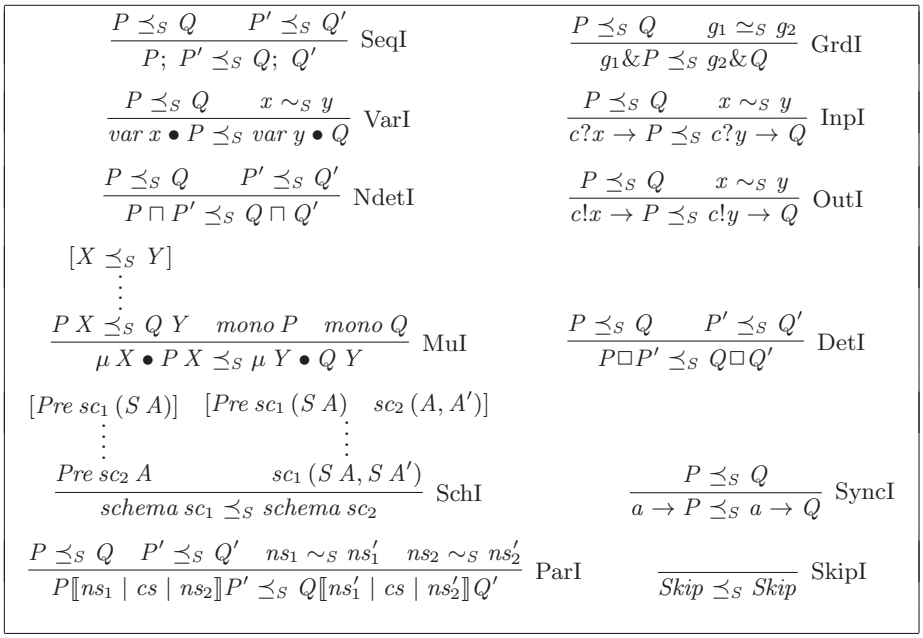

Table 3. Proved refinement laws

```
circusprocess DFIG =
  alphabet = [w::nat, y::nat]
  state = [retidS::nat set, max::nat]
  schema Init = retidS' = {} ∧max' = 0
  schema Out = w' = max ∧ max' = max+1 ∧ retidS' = retidS - {max}
  schema Remove = y < max ∧ y ∉ retidS ∧  retidS' = retidS ∪ {y}
                  ∧ max' = max
  where var w · Schema Init; (μ X ·(req →Schema Out; out!w →Skip)
                                □ (ret?y →Schema Remove); X)
```

We provide the proof of refinement of `FIG` by `DFIG` just instantiating the simulation function `R` by the following abstraction function, that maps the underlying concrete states to abstract states:

```
definition Sim A = FIG_alphabet.make (w A) (y A)
                                ({a. a < (max A) ∧ a ∉ (retidS A)})
```

where A is the alphabet of `DFIG`, and `FIG_alphabet.make` yields an alphabet of type `FIG_Alphabet` initializing the values of `v`, `x` and `idS` by their corresponding values from `DFIG_alphabet`: `w`, `y` and `{a. a < max ∧ a ∉ retidS}`).

To prove that `DFIG` is a refinement of `FIG` one must prove that the main action `DFIG.DFIG` refines the main action `FIG.FIG`. The definition is then simplified, and the refinement laws are applied to simplify the proof goal. Thus, the full proof consists of a few lines in ISAR:

```
theorem "FIG.FIG ≼Sim DFIG.DFIG"
  apply (auto simp: DFIG.DFIG_def FIG.FIG_def mono_Seq
             intro!: VarI SeqI MuI DetI SyncI InpI OutI SkipI)
```

```
  apply (simp_all add: SimRemove SimOut SimInit Sim_def)
done
```

First, the definitions of `FIG.FIG` and `DFIG.DFIG` are simplified and the defined refinement laws are used by the auto tactic as introduction rules. The second step replaces the definition of the simulation function and uses some proved lemmas to finish the proof. The three lemmas used in this proof: `SimInit`, `SimOut` and `SimRemove` give proofs of simulation for the schema `Init`, `Out` and `Remove`.

5 Conclusions

We have shown for the language *Circus*, which combines data-oriented modeling in the style of Z and behavioral modeling in the style of CSP, a semantics in form of a shallow embedding in Isabelle/HOL. In particular, by representing the somewhat non-standard concept of the *alphabet* in UTP in form of extensible records in HOL, we achieved a fairly compact, typed presentation of the language. In contrast to previous work based on some deep embedding [18], this shallow embedding allows arbitrary (higher-order) HOL-types for channels, events, and state-variables, such as, e.g., sets of relations etc. Besides, systematic renaming of local variables is avoided by compiling them essentially to global variables using a stack of variable instances. The necessary proofs for showing that the definitions are consistent — i. e. satisfy altogether `is_CSP_healthy` — have been done, together with a number of algebraic simplification laws on *Circus* processes.

Since the encoding effort can be hidden behind the scene by flexible extension mechanisms of the Isabelle, it is possible to have a compact notation for both specifications and proofs. Moreover, existing standard tactics of Isabelle such as `auto`, `simp` and `metis` can be reused since our *Circus* semantics is representationally close to HOL. Thus, we provide an environment that can cope with combined refinements concerning data and behavior. Finally, we demonstrate its power — w.r.t. both expressivity and proof automation — with a small, but prototypic example of a process-refinement.

In the future, we intend to use Isabelle/*Circus* for the generation of test-cases, on the basis of [6], using the HOL-TestGen-environment [2].

6 Acknowledgement

We would like to thank Markarius Wenzel for his valuable help with the Isabelle framework. Furthermore, we are greatly indebted to Ana Cavalcanti for her comments on the semantic foundation of this work.

References

1. Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic, 2nd edition, 2002. now published by Springer.

2. Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012. To appear.
3. Michael Butler. CSP2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–196, 2000.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
5. A. L. C. Cavalcanti and J. C. P. Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220 – 268. Springer-Verlag, 2006.
6. Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in Circus. *Acta Informatica*, 48(2):97–147, 2011.
7. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
8. Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010, 3rd Int. Symp. on Unifying Theories of Programming*, volume 6445 of *LNCS*, pages 188–206. Springer Verlag, 2010.
9. Clemens Fischer. How to combine Z with process algebra. In *11th Int. Conf. of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.
10. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.
11. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
12. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for Circus. *Electron. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
13. Markus Roggenbach. CSP-CASL: a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354:42–71, 2006.
14. A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
15. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451—470. Springer-Verlag, 2002.
16. K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In *1st Int. Conf. on Formal Engineering Methods*, ICFEM '97, pages 283–292. IEEE, 1997.
17. J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 184—203. Springer-Verlag, 2002.
18. Frank Zeyda and Ana Cavalcanti. Encoding Circus programs in ProofPowerZ. In *Unifying Theories of Programming, 2nd Int. Symp., UTP 2008, Revised Selected Papers*, volume 5713 of *LNCS*. Springer-Verlag, 2009.