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).

Isabelle/Circus is a formalization of the UTP and the Circus language in Isabelle/HOL. It contains proof rules and tactic support that allows for proofs of refinement for Circus processes (involving both data and behavioral aspects).

This environment supports a syntax for the semantic definitions which is close to textbook presentations of Circus.

These theories are presented with details in [9]. This document is a technical appendix of this report.
3 Isabelle/Circus

3.1 Alphabets and Variables ........................................ 11
3.1.1 Updating and accessing global variables. .................. 11
3.1.2 Updating and accessing local variables. ................. 12
3.2 Synchronization infrastructure: Name sets and channels. . 13
3.2.1 Name sets. ................................................. 13
3.2.2 Channels. .................................................. 13
3.3 Actions and Processes ......................................... 14
3.3.1 Basic actions. ............................................. 15
3.3.2 The universal assignment action. ........................... 15
3.3.3 Communications. ......................................... 16
3.3.4 Hiding. .................................................... 17
3.3.5 Recursion. ............................................... 18
3.3.6 Circus Processes. ....................................... 18

4 Using Isabelle/Circus .............................................. 18
4.1 Writing specifications ........................................... 19
4.2 Relational and Functional Refinement in Circus ............. 20
4.3 Refinement Proofs ........................................... 20

5 Conclusions ....................................................... 22

6 Acknowledgement .................................................. 23

7 Pre-declaration of user commands to workaround statefulness of outer syntax ........................................ 24

8 UTP variables ..................................................... 24

9 Predicates and relations .......................................... 24
9.1 Definitions .................................................... 25
9.2 Proofs ........................................................ 26
  9.2.1 Setup of automated tools ................................ 27
  9.2.2 Misc lemmas ............................................. 28

10 Designs ........................................................ 34
10.1 Definitions .................................................... 34
10.2 Proofs ........................................................ 35

11 Reactive processes .............................................. 37
11.1 Preliminaries .................................................. 38
11.2 Definitions ................................................... 40
11.3 Proofs ........................................................ 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 CSP processes</td>
<td>44</td>
</tr>
<tr>
<td>12.1 Definitions</td>
<td>44</td>
</tr>
<tr>
<td>12.2 Proofs</td>
<td>45</td>
</tr>
<tr>
<td>12.3 CSP processes and reactive designs</td>
<td>49</td>
</tr>
<tr>
<td>13 Circus actions</td>
<td>49</td>
</tr>
<tr>
<td>13.1 Definitions</td>
<td>50</td>
</tr>
<tr>
<td>13.2 Proofs</td>
<td>50</td>
</tr>
<tr>
<td>14 Circus variables</td>
<td>53</td>
</tr>
<tr>
<td>15 Denotational semantics of Circus actions</td>
<td>54</td>
</tr>
<tr>
<td>15.1 Skip</td>
<td>54</td>
</tr>
<tr>
<td>15.2 Stop</td>
<td>55</td>
</tr>
<tr>
<td>15.3 Chaos</td>
<td>55</td>
</tr>
<tr>
<td>15.4 State update actions</td>
<td>55</td>
</tr>
<tr>
<td>15.5 Sequential composition</td>
<td>57</td>
</tr>
<tr>
<td>15.6 Internal choice</td>
<td>57</td>
</tr>
<tr>
<td>15.7 External choice</td>
<td>58</td>
</tr>
<tr>
<td>15.8 Reactive design assignment</td>
<td>58</td>
</tr>
<tr>
<td>15.9 Local state external choice</td>
<td>59</td>
</tr>
<tr>
<td>15.10 Schema expression</td>
<td>59</td>
</tr>
<tr>
<td>15.11 Parallel composition</td>
<td>59</td>
</tr>
<tr>
<td>15.12 Local parallel block</td>
<td>61</td>
</tr>
<tr>
<td>15.13 Assignment</td>
<td>61</td>
</tr>
<tr>
<td>15.14 Variable scope</td>
<td>62</td>
</tr>
<tr>
<td>15.15 Guarded action</td>
<td>63</td>
</tr>
<tr>
<td>15.16 Prefixed action</td>
<td>64</td>
</tr>
<tr>
<td>15.17 Hiding</td>
<td>67</td>
</tr>
<tr>
<td>15.18 Recursion</td>
<td>67</td>
</tr>
<tr>
<td>16 Circus syntax</td>
<td>68</td>
</tr>
<tr>
<td>17 Refinement and Simulation</td>
<td>69</td>
</tr>
<tr>
<td>17.1 Definitions</td>
<td>69</td>
</tr>
<tr>
<td>17.2 Proofs</td>
<td>69</td>
</tr>
<tr>
<td>18 Concrete example</td>
<td>72</td>
</tr>
<tr>
<td>18.1 Process definitions</td>
<td>72</td>
</tr>
<tr>
<td>18.2 Simulation proofs</td>
<td>73</td>
</tr>
</tbody>
</table>
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, 10, 17, 14]. In this paper, we consider Circus [18], 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 definitions and proofs, and makes the reuse of standardized proof procedures possible. Compared to implementations based on a “deep embedding” such as [19] this significantly 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. However, 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 paper.

This paper 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 Circus (alphabet, variables ...). We introduce the representation of some Circus actions and process, with an overview of the Isabelle/Circus syntax. In Section 4, we show on 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 spec-
ifications, and give a refinement proof example. A more developed version of this paper can be found in [9].

2 Background

2.1 Isabelle, HOL and Isabelle/HOL

2.1.1 isar

[12] 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 \( \lambda \)-calculus including a Haskell-style type-system with type-classes (e.g. in \( \alpha :: \text{order} \), the type-variable ranges over all types that posses a partial ordering.)

2.1.2 Higher-order logic (HOL)

[7, 1] is a classical logic based on a simple type system. It provides the usual logical connectives like \( \land \), \( \rightarrow \), \( \neg \) as well as the object-logical quantifiers \( \forall x \cdot P x \) and \( \exists x \cdot P x \); 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 \( \lambda \)-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.

2.1.3 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

2.2.1 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:: (\(\alpha \Rightarrow \beta\)) \(\Rightarrow\) \(\alpha \Rightarrow \beta\) \(\Rightarrow\) (\(\alpha \Rightarrow \beta\))  
\((\_\_ \_ \_ := \_\_\_\_)\)

where  
"upd f x v \equiv \lambda z. \text{if } x=z \text{ then } v \text{ else } f z"

The pragma ("\_\_ \_ \_ := \_\_\_\_\") for the Isabelle syntax engine introduces the notation \(f(x:=y)\) for \(\text{upd } f \times y\). 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.

### 2.2.2 Type definitions.

Types can be introduced in Isabelle/HOL in different ways. The most general way to safely introduce new types is using the **typedef** construct. This allows introducing 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.

Similarly, the **datatype** command allows the definition of inductive datatypes. It introduces a datatype using a list of **constructors**. For instance, a logical compiler is invoked for the following introduction of the type option:

**datatype** \(\alpha\) option = None | Some \(\alpha\)

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:

```plaintext
    case x of None \Rightarrow... | Some a\Rightarrow...
```

### 2.2.3 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 
\quad b::T2

which implicitly introduces the record constructor \((a:=e_1, b:=e_2)\) 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 $T_1 \times T_2 \times \delta$. 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 $T_1$, $T_2$ and $T_3$.

### 2.3 Circus and its UTP Foundation

*Circus* is a formal specification language [18] 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 $P \parallel P'$ and $P \cap P'$), 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: $P \parallel n | c | n' \parallel P'$ 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 [15], (but coined in terms of the UTP [13]) providing a notion of execution trace $tr$, refusals $ref$, and divergences. It is expressed in terms of the UTP [11] which makes it amenable to other refinement-notions in UTP. Figure 1 presents a simple *Circus* specification, FIG, the fresh identifiers generator.

#### 2.3.1 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
process \( FIG \) \( \equiv \) begin

state \( S \) \( \equiv \) \( \{ idS : \mathbb{P} \text{ ID} \} \)

\( \text{Init} \) \( \equiv \) \( idS := \emptyset \)

\( \text{Out} \)\[
\begin{array}{l}
\Delta S \\
v! : \text{ID} \\
v! \notin idS \\
idS' = idS \cup \{v!\}
\end{array}
\]

\( \text{Remove} \)\[
\begin{array}{l}
\Delta S \\
x? : \text{ID} \\
idS' = idS \setminus \{x?\}
\end{array}
\]

\( \bullet \) \( \text{Init} ; \) \( \text{var} \ v : \text{ID} \) \( \bullet \)

\( (\mu \ X \bullet (\text{req} \rightarrow \text{Out} ; \text{out}!v \rightarrow \text{Skip} \ \square \ \text{ref} ?x \rightarrow \text{Remove}) ; \ X) \)

end

Figure 1: The Fresh Identifiers Generator in (Textbook) Circus

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: \(\{(A, A') \cdot x A' = x A + y A\}\), 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.

2.3.2 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 \(\text{ok}\). It is used to record the start and termination of a program. A UTP design is defined as follows in Isabelle:

\[(P \vdash Q) \equiv \lambda (A, A'). (\text{ok} A \land P (A, A')) \rightarrow (\text{ok} A' \land Q (A, A'))\]

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: \(\text{wait}\), \(\text{tr}\) and \(\text{ref}\). The boolean variable \(\text{wait}\) records if the process is waiting for an interaction or has terminated. \(\text{tr}\) records the list (trace) of interactions the process has performed so far. The variable \(\text{ref}\) 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 reactive process satisfies some properties [6] (see Table 2 in [9]).

A CSP process is a UTP reactive process that satisfies two additional healthiness conditions(all well-formedness conditions can be found in [9]). A process that satisfies these conditions is said to be CSP healthy.

3 Isabelle/Circus

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

Figure 2: Isabelle/Circus syntax

The Isabelle/Circus environment allows a syntax of processes 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. [13], based on UTP. Circus actions are defined as CSP healthy reactive processes.

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.

We now describe the foundation as well as the semantic definition of some 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 $\Delta$ operations on global states that generate alphabetized relations, or

- **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 of a specification, the Circus semantics considers 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. The *alphabet of a process* is defined by extending the basic reactive process alphabet (cf. Section 2.3.2) by its variable names and types. For 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:

```isabelle
record $\alpha$ alpha = $\alpha$ 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.

#### 3.1.1 Updating and accessing global variables.

Since the alphabets are represented by HOL records, *i.e.* a kind binding ”name $\mapsto$ 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).

For a given alphabet (record) of type σ, (β, the type σ)var represents the type of the variables whose value type is β. 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 later on, in Section 3.3.2.

3.1.2 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. 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 is just adding a new value as the head of the corresponding values list. Leaving a variable scope is just removing the head of the values list. The select and update functions correspond to selecting and updating the head of the list. This ensures dynamic scoping, as it is stated by the *Circus* semantics.

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.

### 3.2 Synchronization infrastructure: Name sets and channels.

#### 3.2.1 Name sets.

An important notion, used in the definition of parallel *Circus* actions, is name sets as seen in Section 2.3. 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. Thus its definition relies on the characterization of the variables in our representation. As for variables, name sets are defined by their functional characterization. They are used in the definition of the binding merge function $MSt$ below:

$$\forall v (v \in ns_1 \Rightarrow v' = (1, v)) \land (v \in ns_2 \Rightarrow v' = (2, v)) \land (v \notin ns_1 \cup ns_2 \Rightarrow v' = v).$$

The disjoint name sets $ns_1$ and $ns_2$ 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:

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

where $lb$ and $gb$ stands for local and global bindings, $x$ and $x_{\text{update}}$ 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.

#### 3.2.2 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. However, the channels may not have the same type: in the example above, the types of chan1, chan2 and chan3 are respectively channels, nat ⇒ channels and bool ⇒ channels. 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 could 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) mentioned in Section 2.1. 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

It is applicable to 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 below the vital type action:

typedef (Action)
(\alpha::\text{chan\_eq}, \sigma) \text{ action} = \{p::(\alpha, \sigma)\text{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 \((\alpha, \sigma)\text{relation\_rp \Rightarrow} (\alpha, \sigma)\text{ action}\) and \((\alpha, \sigma)\text{action \Rightarrow} (\alpha, \sigma)\text{relation\_rp}\) as well as the usual two axioms expressing the bijection Abs_Action(Rep_Action(X))=X and is_CSP_process p \implies Rep_Action(Abs_Action(p))=p where is_CSP_process captures the healthiness conditions.

Every Circus action is an abstraction of an alphabetized predicate. In [9], we introduce the definitions of all the actions and operators using their denotational semantics. The environment contains, for each action, the proof that this predicate is CSP healthy.

In this section, we present some of the important definitions, namely: basic actions, assignments, communications, hiding, and recursion.

3.3.1 Basic actions.

Stop is defined as a reactive design, with a precondition true and a post-condition stating that the system deadlocks and the traces are not evolving.

definition
Stop \equiv \text{Abs\_Action } (R (\text{true} \vdash \lambda (A, A'). \text{tr } A' = \text{tr } A \land \text{wait } A'))

Skip is defined as a reactive design, with a precondition true and a post-condition stating that the system terminates and all the state variables are not changed. We represent this fact by stating that the more field (seen in Section 2.2) is not changed, since this field is mapped to all the state variables. Note that using 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 \equiv \text{Abs\_Action } (R (\text{true} \vdash \lambda (A, A'). \text{tr } A' = \text{tr } A \\
\land \neg \text{wait } A' \land \text{more } A = \text{more } A'))

3.3.2 The universal assignment action.

In Section 3.1.1, we described how global and local variables are 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**

\[
\text{ASSIGN}:: "(\beta, \sigma) \ \text{var} \Rightarrow (\sigma \Rightarrow \beta) \Rightarrow (\alpha::\text{ev_eq}, \sigma) \ \text{action}"
\]

where

\[
\text{ASSIGN} \ x \ e \equiv \text{Abs}\_\text{Action} \ (R \ (\text{true} \vdash Y))
\]

where

\[
Y = \lambda (A, A'). \text{tr} A = \text{tr} A' \land \neg \text{wait} A' \land
\quad \text{more} A' = (\text{assign} \ x \ (e \ (\text{more} A))) \ (\text{more} A)
\]

where \text{assign} is the projection into the update operation of a semantic variable described in section 3.1.1.

### 3.3.3 Communications.

The definition of prefixed actions is based on the definition of a special relation \( \text{do}_I \). In the Circus denotational semantics [13], various forms of prefixing were defined. In our theory, we define one general form, and the other forms are defined as special cases.

**definition** \( \text{do}_I \ c \ x \ P \equiv X \triangleleft \text{wait} \circ \text{fst} \triangleright Y \)

where

\[
X = (\lambda (A, A'). \text{tr} A = \text{tr} A' \land ((c \ ' P) \cap \text{ref} A') = \{\})
\]

and

\[
Y = (\lambda (A, A'). \text{hd} ((\text{tr} A') - (\text{tr} A)) \in (c \ ' P) \land
\quad (c \ (\text{select} \ x \ (\text{more} A))) = (\text{last} \ (\text{tr} A')))
\]

where \( c \) is a channel constructor, \( x \) is a variable (of \text{var} type) and \( P \) is a predicate. The \( \text{do}_I \) 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** \( \text{Prefix} \ c \ x \ P \ S \equiv \text{Abs}\_\text{Action} (R \ (\text{true} \vdash Y)) \ ; \ S \)

where

\[
Y = \text{do}_I \ c \ x \ P \land (\lambda (A, A'). \text{more} A' = \text{more} A)
\]

where \( c \) is a channel constructor, \( x \) is a variable (of type \text{var}), \( P \) is a predicate and \( S \) is an action. This definition states that the prefixed action semantics is given by the interaction semantics (\( \text{do}_I \)) sequentially 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) (λx.λy. y)) true P

where read, write1 and write0 respectively correspond to inputs, outputs and synchronization. Constrained inputs correspond 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

### 3.3.4 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 (∀ x. 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.

### 3.3.5 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 [13]. The use of least fix-points in [13] 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.

Therefore, we must prove that the Circus actions type defines a complete lattice. This leads to prove that the actions type belongs to the HOL “Complete Lattice class”. Since type classes in HOL are hierarchic, the proof is in three steps: first, a proof that the Circus actions type forms a lattice by instantiating the HOL “Lattice class”; second, a proof that actions type instantiates a subclass of lattices called “Bounded Lattice class”; third, proof of the instantiation from the “Complete Lattice class”. More on these proofs can be found in [9].

### 3.3.6 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 makes it possible 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 Figure 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.

```plaintext
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 operator given in Section 3.3. We describe below the result of executing each command of FIG:

- 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, e.g., refinement properties can be developed 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:

\[
\text{definition } A_1 \sqsubseteq c A_2 \equiv (\text{Rep}_\text{Action} A_1) \sqsubseteq \text{utp} (\text{Rep}_\text{Action} A_2)
\]

where A1 and A2 are Circus actions, \(\sqsubseteq c\) and \(\sqsubseteq \text{utp}\) 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 [16, 5] by backwards and forwards simulations. In this paper, 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:

\[
\text{definition } \text{Simulation} ("\_ \preceq \_\_") \text{ where }
A_1 \preceq R A_2 = \forall a b. (\text{Rep}_\text{Action} A_2)(a,b) \rightarrow (\text{Rep}_\text{Action} A_1)(R a,R b)
\]

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 the size of the processes. To avoid this problem, some general refinement laws were defined in [5] to deal with the refinement of Circus actions at operators level and not at UTP level. We introduced and proved a subset of these laws in our environment (see Table 1).
Table 1: Proved refinement laws

In Table 1, the relations \( x \sim_S y \) and \( g_1 \simeq_S g_2 \) record the fact that the variable \( x \) (respectively the guard \( g_1 \)) is refined by the variable \( y \) (respectively 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 \( \text{id}S \) is refined into a set of removed identifiers \( \text{retid}S \) and a number \( \text{max} \), which is the rank of the last issued identifier.

\[
\begin{align*}
circusprocess \text{DFIG} = & \\
\text{alphabet} = [w::\text{nat}, y::\text{nat}] & \\
\text{state} = [\text{retid}S::\text{nat set}, \text{max}::\text{nat}] & \\
\text{schema Init} = & \text{retid}S' = \varnothing & \\
\text{schema Out} = w' = \text{max} & \land \text{max}' = \text{max} + 1 & \land \text{retid}S' = \text{retid}S - \varnothing & \\
\text{schema Remove} = y < \text{max} & \land y \notin \text{retid}S & \land \text{retid}S' = \text{retid}S \cup \varnothing & \land \text{max}' = \text{max} & \\
\end{align*}
\]

where \( \var{w} \cdot \text{Schema Init}; (\mu X \cdot (\text{req} \rightarrow \text{Schema Out}; \text{out}!w \rightarrow \text{Skip})) \land (\text{ret}?y \rightarrow \text{Schema Remove}); X) \)
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:

\[
\text{definition } Sim\ A = FIG\text{-alphabet.make}\ (w\ A)\ (y\ A)\ (\{a. \ a < (\text{max}\ A) \wedge a \not\in (\text{retidS}\ A)\})
\]

where \( A \) is the alphabet of DFIG, and \( FIG\text{-alphabet.make} \) yields an alphabet of type \( FIG\text{\_alphabet} \) initializing the values of \( v, x \) and \( \text{idS} \) by their corresponding values from DFIG\_alphabet: \( w, y \) and \( \{a. \ a < \text{max} \wedge a \not\in \text{retids}\} \).

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:

\[
\text{theorem } "FIG.FIG \preceq Sim\ DFIG.DFIG"\n\]

\[
\text{apply } (\text{auto } \text{simp: DFIG.DFIG\_def } FIG.FIG\_def \text{ mono}\_\text{Seq} \text{ intro!: VarI SeqI MuI DetI SyncI InpI OutI SkipI})\n\]

\[
\text{apply } (\text{simp}\_\text{all } \text{add: Sim\_Remove Sim\_Out Sim\_Init Sim\_def})\n\]

\[
\text{done}
\]

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: Sim\_Init, Sim\_Out and Sim\_Remove 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 [19], 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 [4], using the HOL-TestGen-environment [2].

6 Acknowledgement

We warmly 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.
7 Pre-declaration of user commands to workaround statefulness of outer syntax

theory Commands imports Main
keywords alphabet state channel nameset chanset schema action and circus-process :: thy-decl
begin

⟨ML⟩
end

8 UTP variables

theory Var
imports Commands
begin

UTP variables are characterized by two functions, select and update. The variable type is then defined as a tuple (select * update).

type-synonym (′a, ′r) var = (′r ⇒ ′a) * ((′a ⇒ ′a) ⇒ ′r ⇒ ′r)

The lookup function returns the corresponding select function of a variable.

definition lookup :: (′a, ′r) var ⇒ ′r ⇒ ′a
    where lookup f ≡ (fst f)

The assign function uses the update function of a variable to update its value.

definition assign :: (′a, ′r) var ⇒ ′a ⇒ ′r ⇒ ′r
    where assign f v ≡ (snd f) (λ . v)

The VAR function allows to retrieve a variable given its name.

syntax -VAR :: id ⇒ (′a, ′r) var (VAR -)
translations VAR x => (x, -update-name x)

end

9 Predicates and relations

theory Relations
imports Var
begin
default-sort type

Unifying Theories of Programming (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.

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.

9.1 Definitions

In this section, the definitions of predicates, relations and standard operators are given.

\[
\text{type-synonym } \alpha \text{ alphabet } = \alpha \\
\text{type-synonym } \alpha \text{ predicate } = \alpha \text{ alphabet } \Rightarrow \text{bool}
\]

\[
definition \text{true}::\alpha \text{ predicate} \\
\text{where } \text{true} \equiv \lambda A. \text{True}
\]

\[
definition \text{false}::\alpha \text{ predicate} \\
\text{where } \text{false} \equiv \lambda A. \text{False}
\]

\[
definition \text{not}::\alpha \text{ predicate} \Rightarrow \alpha \text{ predicate} \\
\text{where } \neg P \equiv \lambda A. \neg (P A)
\]

\[
definition \text{conj}::\alpha \text{ predicate} \Rightarrow \alpha \text{ predicate} \\
\text{where } P \land Q \equiv \lambda A. P A \land Q A
\]

\[
definition \text{disj}::\alpha \text{ predicate} \Rightarrow \alpha \text{ predicate} \\
\text{where } P \lor Q \equiv \lambda A. P A \lor Q A
\]

\[
definition \text{impl}::\alpha \text{ predicate} \Rightarrow \alpha \text{ predicate} \\
\text{where } P \rightarrow Q \equiv \lambda A. P A \rightarrow Q A
\]

\[
definition \text{iff}::\alpha \text{ predicate} \Rightarrow \alpha \text{ predicate} \\
\text{where } P \leftrightarrow Q \equiv \lambda A. P A \leftrightarrow Q A
\]

\[
definition \text{ex}::[\beta \Rightarrow \alpha \text{ predicate}] \Rightarrow \alpha \text{ predicate} \\
\text{where } \exists x. P x \equiv \lambda A. \exists x. (P x) A
\]

\[
definition \text{all}::[\beta \Rightarrow \alpha \text{ predicate}] \Rightarrow \alpha \text{ predicate} \\
\text{where } \forall x. P x \equiv \lambda A. \forall x. (P x) A
\]

\[
\text{type-synonym } \alpha \text{ condition } = (\alpha \times \alpha) \Rightarrow \text{bool} \\
\text{type-synonym } \alpha \text{ relation } = (\alpha \times \alpha) \Rightarrow \text{bool}
\]

\[
definition \text{cond}::\alpha \text{ relation} \Rightarrow \alpha \text{ relation} \\
\text{where } (P \circ b \triangleright Q) \equiv (b \land P) \lor ((\neg b) \land Q)
\]

25
definition \( \text{comp} : ((\alpha \times \beta) \Rightarrow \text{bool}) \Rightarrow ((\beta \times \gamma) \Rightarrow \text{bool}) \Rightarrow (\alpha \times \gamma) \Rightarrow \text{bool} \)

where \( P ; ; Q \equiv \lambda r. r : (\{ p. P \ p \} \ O \ \{ q. Q \ q \}) \)

definition \( \text{Assign} : (a, b) \ \text{var} \Rightarrow a \Rightarrow b \ \text{relation} \)

where \( \text{Assign} x a \equiv \lambda (A, A'). A' = (\text{assign} x a) A \)

syntax

-\text{-assignment} :: id \Rightarrow a \Rightarrow b \ \text{relation} (\cdot := \cdot)

translations

\( y ::= vv \Rightarrow \text{CONST Assign (VAR y) vv} \)

abbreviation (input) \text{closure} :: \alpha \ \text{predicate} \Rightarrow \text{bool ([-])}

where \( \{ P \} \equiv \forall A. \ P A \)

abbreviation (input) \text{ndet} :: \alpha \ \text{relation} \Rightarrow a \Rightarrow b \ \text{relation} ((\cdot - \cap \cdot))

where \( P \cap Q \equiv P \land Q \)

abbreviation (input) \text{join} :: \alpha \ \text{relation} \Rightarrow a \Rightarrow b \ \text{relation} ((\cdot - \cup \cdot))

where \( P \cup Q \equiv P \lor Q \)

abbreviation (input) \text{ndetS} :: \alpha \ \text{relation set} \Rightarrow a \Rightarrow b \ \text{relation} ((\bigcap \cdot))

where \( \bigcap S \equiv \lambda A. \ A \in \bigcup \{ p. P \ p \} \ | \ P. P \in S \}

abbreviation (input) \text{conjS} :: \alpha \ \text{relation set} \Rightarrow a \Rightarrow b \ \text{relation} ((\bigcup \cdot))

where \( \bigcup S \equiv \lambda A. \ A \in \bigcap \{ p. P \ p \} \ | \ P. P \in S \}

abbreviation (input) \text{skip-r} :: \alpha \ \text{relation}

where \( \Pi r \equiv \lambda (A, A'). A = A' \)

abbreviation (input) \text{Bot} :: \alpha \ \text{relation}

where \( \text{Bot} \equiv \text{true} \)

abbreviation (input) \text{Top} :: \alpha \ \text{relation}

where \( \text{Top} \equiv \text{false} \)

lemmas \( \text{utp-defs} = \text{true-def false-def conj-def disj-def not-def impl-def iff-def ex-def all-def cond-def comp-def Assign-def} \)

9.2 Proofs

All useful proved lemmas over predicates and relations are presented here. First, we introduce the most important lemmas that will be used by automatic tools to simplify proofs. In the second part, other lemmas are proved using these basic ones.
9.2.1 Setup of automated tools

lemma true-intro: true x \langle proof \rangle
lemma false-elim: false x \implies C \langle proof \rangle
lemma true-elim: true x \implies C \implies C \langle proof \rangle

lemma not-intro: (P x \implies false x) \implies (\neg P) x \langle proof \rangle
lemma not-elim: (\neg P) x \implies P x \implies C \langle proof \rangle
lemma not-dest: (\neg P) x \implies \neg P x \langle proof \rangle

lemma conj-intro: P x \implies Q x \implies (P \land Q) x \langle proof \rangle
lemma conj-elim: (P \land Q) x \implies (P x \implies Q x \implies C) \implies C \langle proof \rangle

lemma disj-introC: (\neg Q x \implies P x) \implies (P \lor Q) x \langle proof \rangle
lemma disj-elim: (P \lor Q) x \implies (P x \implies C) \implies (Q x \implies C) \implies C \langle proof \rangle

lemma impl-intro: (P x \implies Q x) \implies (P \implies Q) x \langle proof \rangle
lemma impl-elimC: (P \implies Q) x \implies (\neg P x \implies R) \implies (Q x \implies R) \implies R \langle proof \rangle
lemma iff-intro: (P x \implies Q x) \implies (Q x \implies P x) \implies (P \iff Q) x \langle proof \rangle
lemma iff-elimC: (P \iff Q) x \implies (P x \implies Q x \implies R) \implies (\neg P x \implies \neg Q x \implies R) \implies R \langle proof \rangle

lemma all-intro: (\forall a. P a x) \implies (\exists a. P a) x \langle proof \rangle
lemma all-elim: (\exists a. P a) x \implies (P a x \implies R) \implies R \langle proof \rangle

lemma ex-intro: P a x \implies (\exists a. P a) x \langle proof \rangle
lemma ex-elim: (\exists a. P a) x \implies (\forall a. P a x \implies Q) \implies Q \langle proof \rangle
lemma comp-intro: P (a, b) \implies Q (b, c) \implies (P ; Q) (a, c) \langle proof \rangle
lemma comp-elims: (P ; Q) ac \implies (\forall a b c. ac = (a, c) \implies P (a, b) \implies Q (b, c) \implies C) \implies C \langle proof \rangle

declare not-def [simp]
declare iff-intro [intro!]
and not-intro [intro!]
and impl-intro [intro!]
and disj-introC [intro!]
and conj-intro [intro!]
and true-intro [intro!]
and comp-intro [intro!]
declare not-dest [dest!]
and iff-elimC [elim!]

27
and false-elim [elim!]
and impl-elimC [elim!]
and disj-elim [elim!]
and conj-elim [elim!]
and comp-elim [elim!]
and true-elim [elim!]

declare all-intro [intro!] and ex-intro [intro]
declare ex-elim [elim!] and all-elim [elim]

lemmas relation-rules = iff-intro not-intro impl-intro disj-introC conj-intro true-intro
comp-intro not-dest iff-elimC false-elim impl-elimC all-elim
disj-elim conj-elim comp-elim all-intro ex-intro ex-elim

lemma split-cond:
A ((P ⊢ b ▷ Q) x) = ((b x → A (P x)) ∧ (¬ b x → A (Q x)))
⟨proof⟩

lemma split-cond-asm:
A ((P ⊢ b ▷ Q) x) = (¬ ((b x ∧ ¬ A (P x)) ∨ (¬ b x ∧ ¬ A (Q x))))
⟨proof⟩

lemmas cond-splits = split-cond split-cond-asm

9.2.2 Misc lemmas

lemma cond-idem:(P ⊢ b ▷ P) = P
⟨proof⟩

lemma cond-symm:(P ⊢ b ▷ Q) = (Q ⊢ ¬ b ▷ P)
⟨proof⟩

lemma cond-assoc: ((P ⊢ b ▷ Q) ⊢ c ▷ R) = (P ⊢ b ∧ c ▷ (Q ⊢ c ▷ R))
⟨proof⟩

lemma cond-distr: (P ⊢ b ▷ (Q ⊢ c ▷ R)) = ((P ⊢ b ▷ Q) ⊢ c ▷ (P ⊢ b ▷ R))
⟨proof⟩

lemma cond-unit-T:(P ⊢ true ▷ Q) = P
⟨proof⟩

lemma cond-unit-F:(P ⊢ false ▷ Q) = Q
⟨proof⟩

lemma cond-L6: (P ⊢ b ▷ (Q ⊢ b ▷ R)) = (P ⊢ b ▷ R)
⟨proof⟩

28
lemma cond-L7: \((P \triangleleft b \triangleright (P \triangleleft c \triangleright Q)) = (P \triangleleft b \lor c \triangleright Q)\)

(proof)

lemma cond-and-distr: \(((P \land Q) \triangleleft b \triangleright (R \land S)) = ((P \triangleleft b \triangleright R) \land (Q \triangleleft b \triangleright S))\)

(proof)

lemma cond-or-distr: \(((P \lor Q) \triangleleft b \triangleright (R \lor S)) = ((P \triangleleft b \triangleright R) \lor (Q \triangleleft b \triangleright S))\)

(proof)

lemma cond-imp-distr:
\(((P \rightarrow Q) \triangleleft b \triangleright (R \rightarrow S)) = ((P \triangleleft b \triangleright R) \rightarrow (Q \triangleleft b \triangleright S))\)

(proof)

lemma cond-eq-distr:
\(((P \leftrightarrow Q) \triangleleft b \triangleright (R \leftrightarrow S)) = ((P \triangleleft b \triangleright R) \leftrightarrow (Q \triangleleft b \triangleright S))\)

(proof)

lemma comp-assoc:
\(((P ; ; (Q ; ; R)) = ((P ; ; Q) ; ; R))\)

(proof)

lemma conj-comp:
\(\forall a b c. P \ (a, b) = P \ (a, c) \implies (P \land (Q ; ; R)) = ((P \land Q) ; ; R)\)

(proof)

lemma comp-cond-left-distr:
assumes \(\forall x y z. b \ (x, y) = b \ (x, z)\)
shows \(((P \triangleleft b \triangleright Q) ; ; R) = ((P ; ; R) \triangleleft b \triangleright (Q ; ; R))\)

(proof)

lemma ndet-symm: \((P::'a relation) \cap Q = Q \cap P\)

(proof)

lemma ndet-assoc: \(P \cap (Q \cap R) = (P \cap Q) \cap R\)

(proof)

lemma ndet-idemp: \(P \cap P = P\)

(proof)

lemma ndet-distr: \(P \cap (Q \cap R) = (P \cap Q) \cap (P \cap R)\)

(proof)

lemma cond-ndet-distr: \((P \triangleleft b \triangleright (Q \cap R)) = ((P \triangleleft b \triangleright Q) \cap (P \triangleleft b \triangleright R))\)

(proof)

lemma ndet-cond-distr: \((P \cap (Q \triangleleft b \triangleright R)) = ((P \cap Q) \triangleleft b \triangleright (P \cap R))\)

(proof)

lemma comp-ndet-l-distr: \(((P \cap Q) ; ; R) = ((P ; ; R) \cap (Q ; ; R))\)

(proof)
lemma comp-ndet-r-distr: \((P ; ; (Q \cap R)) = ((P ; ; Q) \cap (P ; ; R))\)
(proof)

lemma l2-5-1-A: \(\forall X \in S. [X \rightarrow (\prod S)]\)
(proof)

lemma l2-5-1-B: \(\forall X \in S. [X \rightarrow P]) \rightarrow [\prod S) \rightarrow P]\)
(proof)

lemma l2-5-1: \([\prod S) \rightarrow P) \leftrightarrow (\forall X \in S. [X \rightarrow P])\)
(proof)

lemma empty-disj: \(\prod \{\} = Top\)
(proof)

lemma l2-5-1-2: \([P \rightarrow (\bigcup S)] \leftrightarrow (\forall X \in S. [P \rightarrow X])\)
(proof)

lemma empty-conj: \(\bigcup \{\} = Bot\)
(proof)

lemma l2-5-2: \((\bigcup S) \cap Q = (\bigcup \{P \cap Q | P. P \in S\})\)
(proof)

lemma l2-5-3: \((\prod S) \cup Q = (\prod \{P \cup Q | P. P \in S\})\)
(proof)

lemma l2-5-4: \((\prod S) ; ; Q = (\prod \{P ; ; Q | P. P \in S\})\)
(proof)

lemma l2-5-5: \((Q ; ; (\prod S)) = (\prod \{Q ; ; P | P. P \in S\})\)
(proof)

lemma all-idem :\(\forall b. \forall a. P a) = (\forall a. P a)\)
(proof)

lemma comp-unit-R [simp]: \((P ; ; \Pi r) = P\)
(proof)

lemma comp-unit-L [simp]: \((\Pi r ; ; P) = P\)
(proof)

lemmas comp-unit-simps = comp-unit-R comp-unit-L

lemma not-cond: \(\neg(P \triangle b \triangleright Q)) = ((\neg P) \triangle b \triangleright (\neg Q))\)
(proof)

lemma cond-conj-not-distr:
\[(P \triangleq Q) \land \neg(R \triangleq Q) = ((P \land \neg R) \triangleq Q \land \neg S)\]

**lemma** imp-cond-distr: \((R \implies (P \triangleq Q)) = ((R \implies P) \triangleq (R \implies Q))\)

**lemma** cond-imp-dist: \((P \triangleq Q) \implies R = ((P \implies R) \triangleq (Q \implies R))\)

**lemma** cond-conj-distr: \(((P \triangleq Q) \land R) = ((P \land R) \triangleq (Q \land R))\)

**lemma** cond-disj-distr: \(((P \triangleq Q) \lor R) = ((P \lor R) \triangleq (Q \lor R))\)

**lemma** cond-know-b: \((b \land (P \triangleq Q)) = (b \land P)\)

**lemma** cond-know-nb: \(((\neg b) \land (P \triangleq Q)) = ((\neg b) \land Q)\)

**lemma** cond-ass-if: \((P \triangleq Q) = (((b) \land (P \triangleq Q)))\)

**lemma** cond-ass-else: \((P \triangleq Q) = (P \triangleq ((\neg b) \land Q))\)

**lemma** not-true-eq-false: \((\neg true) = false\)

**lemma** not-false-eq-true: \((\neg false) = true\)

**lemma** conj-idem: \((P ::\alpha \text{ predicate}) \land P) = P\)

**lemma** disj-idem: \((P ::\alpha \text{ predicate}) \lor P) = P\)

**lemma** conj-comm: \((P ::\alpha \text{ predicate}) \land Q) = (Q \land P)\)

**lemma** disj-comm: \((P ::\alpha \text{ predicate}) \lor Q) = (Q \lor P)\)

**lemma** coni-subst: \(P = R \implies ((P ::\alpha \text{ predicate}) \land Q) = (R \land Q)\)

**lemma** disj-subst: \(P = R \implies ((P ::\alpha \text{ predicate}) \lor Q) = (R \lor Q)\)

31
proof
lemma conj-assoc:((P::'a predicate) ∧ Q) ∧ S = (P ∧ (Q ∧ S))
(proof)

lemma disj-assoc:((P::'a predicate) ∨ Q) ∨ S = (P ∨ (Q ∨ S))
(proof)

lemma conj-disj-abs:((P::'a predicate) ∧ (P ∨ Q)) = P
(proof)

lemma disj-conj-abs:((P::'a predicate) ∨ (P ∧ Q)) = P
(proof)

lemma conj-disj-distr:((P::'a predicate) ∧ (Q ∨ R)) = ((P ∧ Q) ∨ (P ∧ R))
(proof)

lemma disj-conj-distr:((P::'a predicate) ∨ (Q ∧ R)) = ((P ∨ Q) ∧ (P ∨ R))
(proof)

lemma true-conj-id:(P ∧ true) = P
(proof)

lemma true-disj-zero:(P ∨ true) = true
(proof)

lemma true-conj-zero:(P ∧ false) = false
(proof)

lemma true-disj-id:(P ∨ false) = P
(proof)

lemma imp-vacuous: (false → u) = true
(proof)

lemma p-and-not-p: (P ∧ ¬ P) = false
(proof)

lemma conj-disj-not-abs: ((P::'a predicate) ∧ ((¬P) ∨ Q)) = (P ∧ Q)
(proof)

lemma p-or-not-p: (P ∨ ¬ P) = true
(proof)

lemma double-negation: (¬ ¬ (P::'a predicate)) = P
(proof)

lemma not-conj-deMorgans: (¬ ((P::'a predicate) ∧ Q)) = ((¬ P) ∨ (¬ Q))
(proof)
lemma not-disj-deMorgans: \((\neg ((P::\alpha \ predicate) \lor Q)) = ((\neg P) \land (\neg Q))\)  
(proof)

lemma p-imp-p: \((P \rightarrow P) = true\)  
(proof)

lemma imp-imp: \(((P::\alpha \ predicate) \rightarrow (Q \rightarrow R)) = ((P \land Q) \rightarrow R)\)  
(proof)

lemma imp-trans: \(((P \rightarrow Q) \land (Q \rightarrow R) \rightarrow P \rightarrow R) = true\)  
(proof)

lemma p-equiv-p: \((P \leftrightarrow P) = true\)  
(proof)

lemma equiv-eq: \(((P::\alpha \ predicate) \land Q) \lor (\neg P \land \neg Q) = true) \leftrightarrow (P = Q)\)  
(proof)

lemma equiv-eq1: \(((P::\alpha \ predicate) \leftrightarrow Q) = true) \leftrightarrow (P = Q)\)  
(proof)

lemma cond-subst: \(b = c \Longrightarrow (P \triangleleft b \triangleright Q) = (P \triangleleft c \triangleright Q)\)  
(proof)

lemma ex-disj-distr: \(((\exists x. \ P x) \lor (\exists x. \ Q x)) = (\exists x. \ (P x \lor Q x))\)  
(proof)

lemma all-disj-distr: \(((\forall x. \ P x) \lor (\forall x. \ Q)) = (\forall x. \ (P x \lor Q))\)  
(proof)

lemma all-conj-distr: \(((\forall x. \ P x) \land (\forall x. \ Q)) = (\forall x. \ (P x \land Q x))\)  
(proof)

lemma all-triv: \((\forall x. \ P) = P\)  
(proof)

lemma closure-true: [true]  
(proof)

lemma closure-p-eq-true: [P] \leftrightarrow (P = true)  
(proof)

lemma closure-equiv-eq: [P \leftrightarrow Q] \leftrightarrow (P = Q)  
(proof)

lemma closure-conj-distr: [(P \land [Q])] = [P \land Q]  
(proof)
lemma closure-imp-distr: \([P \rightarrow Q] \rightarrow [P] \rightarrow [Q]\) 
(proof)

lemma true-iff[simp]: \((P \leftrightarrow \text{true}) = P\) 
(proof)

lemma true-imp[simp]: \((\text{true} \rightarrow P) = P\) 
(proof)

end

10 Designs

theory Designs
imports Relations
begin

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.

10.1 Definitions

In the following, the definitions of designs alphabets, designs and healthiness (well-formedness) conditions are given. The healthiness conditions of designs are defined by \(H_1, H_2, H_3\) and \(H_4\).

record alpha-d = ok::bool

type-synonym \(\alpha\) alphabet-d = \(\alpha\) alpha-d-scheme alphabet

type-synonym \(\alpha\) relation-d = \(\alpha\) alphabet-d relation

definition design::\(\alpha\) relation-d \=> \(\alpha\) relation-d \=> \(\alpha\) relation-d ((\(-\vdash-\))
where \((P \vdash Q) \equiv \lambda (A, A') . (\text{ok } A \land P (A,A')) \rightarrow (\text{ok } A' \land Q (A,A'))\)

definition skip-d :: \(\alpha\) relation-d (IId)
where IId \equiv (\text{true} \vdash IIr)

definition J
where \(J \equiv \lambda (A, A') . (\text{ok } A \rightarrow \text{ok } A') \land \text{more } A = \text{more } A'\)

type-synonym \(\alpha\) Healthiness-condition = \(\alpha\) relation \Rightarrow \(\alpha\) relation

definition Healthy::\(\alpha\) relation \Rightarrow \(\alpha\) Healthiness-condition \Rightarrow bool (- is - healthy)
where \(P\) is \(H\) healthy \equiv (\(P = H\) P)
lemma Healthy-def: $P$ is $H$ healthy $= (H \ P = P)$
(proof)

definition $H1::(\alpha$ alphabet-d) Healthiness-condition
where $H1 (P) \equiv (ok \circ \text{fst} \rightarrow P)$

definition $H2::(\alpha$ alphabet-d) Healthiness-condition
where $H2 (P) \equiv P ; ; J$

definition $H3::(\alpha$ alphabet-d) Healthiness-condition
where $H3 (P) \equiv P ; ; \Pi d$

definition $H4::(\alpha$ alphabet-d) Healthiness-condition
where $H4 (P) \equiv ((P ; ; \text{true}) \leftrightarrow \text{true})$

definition $\sigma f::\alpha$ relation-d $\Rightarrow \alpha$ relation-d
where $\sigma f D \equiv \lambda (A, A') . D (A, A'[(\text{ok}:= \text{False}])]$

definition $\sigma t::\alpha$ relation-d $\Rightarrow \alpha$ relation-d
where $\sigma t D \equiv \lambda (A, A') . D (A, A'[(\text{ok}:= \text{True}])]$

definition OKAY::\alpha$ relation-d
where $\text{OKAY} \equiv \lambda (A, A') . \text{ok} A$

definition OKAY'::\alpha$ relation-d
where $\text{OKAY}' \equiv \lambda (A, A') . \text{ok} A'$

lemmas design-defs = design-def skip-d-def J-def Healthy-def H1-def H2-def H3-def H4-def $\sigma f$-def $\sigma t$-def OKAY-def OKAY'-def

10.2 Proofs

Proof of theorems and properties of designs and their healthiness conditions
are given in the following.

lemma t-comp-lz-d: $(\text{true} ; ; (P \vdash Q)) = \text{true}$
(proof)

lemma pi-comp-left-unit: $(\Pi d ; ; (P \vdash Q)) = (P \vdash Q)$
(proof)

theorem t3-I-4-2:
$((P1 \vdash Q1) \triangleleft b \triangleright (P2 \vdash Q2)) = ((P1 \triangleleft b \triangleright P2) \vdash (Q1 \triangleleft b \triangleright Q2))$
(proof)

lemma conv-conj-distr: $\sigma t (P \wedge Q) = (\sigma t P \wedge \sigma t Q)$
(proof)

lemma conv-disj-distr: $\sigma t (P \lor Q) = (\sigma t P \lor \sigma t Q)$
(proof)
lemma conv-imp-distr: $\sigma t (P \rightarrow Q) = ((\sigma t P) \rightarrow \sigma t Q)$
(proof)

lemma conv-not-distr: $\sigma t (\neg P) = (\neg (\sigma t P))$
(proof)

lemma div-conj-distr: $\sigma f (P \land Q) = (\sigma f P \land \sigma f Q)$
(proof)

lemma div-disj-distr: $\sigma f (P \lor Q) = (\sigma f P \lor \sigma f Q)$
(proof)

lemma div-imp-distr: $\sigma f (P \rightarrow Q) = ((\sigma f P) \rightarrow \sigma f Q)$
(proof)

lemma div-not-distr: $\sigma f (\neg P) = (\neg (\sigma f P))$
(proof)

lemma ok-conv: $\sigma t OKAY = OKAY$
(proof)

lemma ok-div: $\sigma f OKAY = OKAY$
(proof)

lemma ok'-conv: $\sigma t OKAY' = true$
(proof)

lemma ok'-div: $\sigma f OKAY' = false$
(proof)

lemma H2-J-1:
assumes A: $P$ is H2 healthy
shows $[(\lambda (A, A'). (P(A, A'\{ok := False\})) \rightarrow P(A, A'\{ok := True\}))]]$
(proof)

lemma H2-J-2-a: $P(a,b) \rightarrow (P ;; J) (a,b)$
(proof)

lemma ok-or-not-ok : $[P(a, b\{ok := True\}); P(a, b\{ok := False\})] \Rightarrow P(a, b)$
(proof)

lemma H2-J-2-b :
assumes A: $[(\lambda (A, A'). (P(A, A'\{ok := False\})) \rightarrow P(A, A'\{ok := True\}))]]$
and B : $P ;; J) (a,b)$
shows $P (a,b)$
(proof)

lemma H2-J-2 :
assumes $A$: $[(\lambda (A, A'). \; P(A, A'[\langle ok := False\rangle]) \rightarrow P(A, A'[\langle ok := True\rangle]))]$  
shows $P$ is $H2$ healthy

lemma $H2-J$:  
$[\lambda (A, A'). \; P(A, A'[\langle ok := False\rangle]) \rightarrow P(A, A'[\langle ok := True\rangle])] = P$ is $H2$ healthy

lemma $design-eq1$: $(P \vdash Q) = (P \vdash P \land Q)$

lemma $H1-idem$: $H1 \circ H1 = H1$

lemma $H1-idem2$: $(H1 \; (H1 \; P)) = (H1 \; P)$

lemma $H2-idem$: $H2 \circ H2 = H2$

lemma $H2-idem2$: $(H2 \; (H2 \; P)) = (H2 \; P)$

lemma $H1-H2-commute$: $H1 \circ H2 = H2 \circ H1$

lemma $H1-H2-commute2$: $H1 \; (H2 \; P) = H2 \; (H1 \; P)$

lemma $alpha-d-eqD$: $r = r' \implies ok \; r = ok \; r' \land alpha-d.more \; r = alpha-d.more \; r'$

lemma $design-H1$: $(P \vdash Q)$ is $H1$ healthy

lemma $design-H2$: 
$(\forall \; a, b. \; P \; (a, b[\langle ok := True\rangle])) \rightarrow P \; (a, b[\langle ok := False\rangle]) \implies (P \vdash Q)$ is $H2$ healthy

end

11 Reactive processes

theory Reactive-Processes
imports Designs ~~/src/HOL/Library/Sublist

begin
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.

In this section, we introduce first some preliminary notions, useful for trace manipulations. The definitions of reactive process alphabets and healthiness conditions are also given. Finally, proved lemmas and theorems are listed.

11.1 Preliminaries

type-synonym 'α trace = 'α list

fun list-diff::'α list ⇒ 'α list ⇒ 'α list option where
    list-diff l [] = Some l
    | list-diff [] l = None
    | list-diff (x#xs) (y#ys) = (if (x = y) then (list-diff xs ys) else None)

instantiation list :: (type) minus
begin
definition list-minus: l1 − l2 ≡ the (list-diff l1 l2)
instance ⟨proof⟩
end

lemma list-diff-empty [simp]: the (list-diff l []) = l ⟨proof⟩

lemma prefix-diff-empty [simp]: l − [] = l ⟨proof⟩

lemma prefix-diff-eq [simp]: l − l = [] ⟨proof⟩

lemma prefix-diff [simp]: (l @ t) − l = t ⟨proof⟩

lemma prefix-subst [simp]: l @ t = m ⇒ m − l = t ⟨proof⟩

lemma prefix-subst1 [simp]: m = l @ t ⇒ m − l = t ⟨proof⟩

lemma prefix-diff1 [simp]: ((l @ m) @ t) − (l @ m) = t ⟨proof⟩
lemma prefix-diff2 [simp]: \( (l @ (m @ t)) - (l @ m) = t \)
(proof)

lemma prefix-diff3 [simp]: \( (l @ m) - (l @ t) = (m - t) \)
(proof)

lemma prefix-diff4 [simp]: \( (a # m) - (a # t) = (m - t) \)
(proof)

class ev-eq =
fixes ev-eq :: 'a ⇒ 'a ⇒ bool
assumes refl: ev-eq a a
assumes comm: ev-eq a b = ev-eq b a
definition filter-chan-set a cs = \( \neg (\exists e \in cs. \ ev-eq a e) \) 

lemma in-imp-not-fcs: \( x \in S \implies \neg \ \text{filter-chan-set} \ x \ S \)
(proof)

fun tr-filter :: 'a::ev-eq list ⇒ 'a set ⇒ 'a list where
\[
\begin{align*}
\text{tr-filter [ ] cs} &= [] \\
\text{tr-filter} (x#xs) cs &= (\text{if (filter-chan-set x cs) then } (x#(\text{tr-filter xs cs})) \\
& \quad \text{else } (\text{tr-filter xs cs}))
\end{align*}
\]

lemma tr-filter-conc: \( (\text{tr-filter} (a@b) \ cs) = ((\text{tr-filter} a \ cs) @ (\text{tr-filter} b \ cs)) \)
(proof)

lemma filter-chan-set-hd-tr-filter:
\( \text{tr-filter l cs} \neq [] \implies \text{filter-chan-set} \ (hd (\text{tr-filter l cs})) \ cs \)
(proof)

lemma tr-filter-conc-eq1:
\( (a@b = (\text{tr-filter} (a@c) \ cs)) \implies (b = (\text{tr-filter} c \ cs)) \)
(proof)

lemma tr-filter-conc-eq2:
\( (a@b = (\text{tr-filter} (a@c) \ cs)) \implies (a = (\text{tr-filter} a \ cs)) \)
(proof)

lemma tr-filter-conc-eq:
\( (a@b = (\text{tr-filter} (a@c) \ cs)) = (b = (\text{tr-filter} c \ cs) \ & \ a = (\text{tr-filter} a \ cs)) \)
(proof)

lemma tr-filter-conc-eq3:
\( (b = (\text{tr-filter} (a@c) \ cs)) = (\exists b1 b2. \ b=b1@b2 \ & \ b2 = (\text{tr-filter} c \ cs) \ & \ b1 = (\text{tr-filter} a \ cs)) \)
lemma tr-filter-un:
tr-filter l (s1 ∪ s2) = tr-filter (tr-filter l s1) s2
(\textit{proof})

instantiation list :: (ev-eq) ev-eq
\begin{\textit{fun}}
\textit{ev-eq-list where}
\begin{align*}
\textit{ev-eq-list} \quad & \quad [] \quad [] = \text{True} \\
| \quad \textit{ev-eq-list} \quad & \quad l \quad [] = \text{False} \\
| \quad \textit{ev-eq-list} \quad & \quad [] \quad l = \text{False} \\
| \quad \textit{ev-eq-list} \quad & \quad (x \# xs) \quad (y \# ys) = (\text{if} \ (\text{ev-eq} \ x \ y) \ \text{then} \ (\text{ev-eq-list} \ xs \ ys) \ \text{else} \ \text{False})
\end{align*}
\end{\textit{fun}}
\begin{\textit{instance}}
\begin{proof}
\end{\textit{instance}}

11.2 Definitions
abbreviation subl::'a list \Rightarrow 'a list \Rightarrow bool (\cdot \leq \cdot)
where l1 \leq l2 == Sublist.prefixeq l1 l2

lemma list-diff-empty-eq: l1 - l2 = [] \Rightarrow l2 \leq l1 \Rightarrow l1 = l2
(\textit{proof})

The definitions of reactive process alphabets and healthiness conditions are
given in the following. The healthiness conditions of reactive processes are
defined by R1, R2, R3 and their composition R.

type-synonym 'ϑ refusal = 'ϑ set

record 'ϑ alpha-rp = alpha-d +
\begin{align*}
| \quad wait:: bool \\
| \quad tr :: 'ϑ trace \\
| \quad ref :: 'ϑ refusal
\end{align*}

Note that we define here the class of UTP alphabets that contain \textit{wait},
\textit{tr} and \textit{ref}, or, in other words, we define here the class of reactive process
alphabets.

type-synonym ('ϑ,'σ) alphabet-rp = ('ϑ,'σ) alpha-rp-scheme alphabet

\begin{align*}
type-synonym ('ϑ,'σ) relation-rp \quad = \quad ('ϑ,'σ) alphabet-rp relation
\end{align*}

definition diff-tr s1 s2 = ((tr s1) - (tr s2))

definition spec :: [bool, bool, ('ϑ,'σ) relation-rp] \Rightarrow ('ϑ,'σ) relation-rp
where \begin{align*}
\begin{proof}
\end{proof}
\end{align*}

abbreviation Speciftt (-t t) where (P)^t \equiv \text{spec} True True P

40
abbreviation Specifff (-f) where \((P)^f_f \equiv \text{spec False False } P\)

abbreviation Specifft (-f) where \((P)^f_t \equiv \text{spec True False } P\)

abbreviation Specifft (-t) where \((P)^t_f \equiv \text{spec False True } P\)

definition \(R1::((\'\vartheta,\'\sigma) \text{ alphabet-rp})\) \(\text{Healthiness-condition}\)
where \(R1(P) \equiv \lambda (A,A'). (\text{spec False } (A,A')) \land (\text{tr } A \leq \text{tr } A')\)

definition \(R2::((\'\vartheta,\'\sigma) \text{ alphabet-rp})\) \(\text{Healthiness-condition}\)
where \(R2(P) \equiv \lambda (A,A'). (\text{spec True } (A,A')) \land (\text{tr } A \leq \text{tr } A')\)

definition \(\Pi\)rea
where \(\Pi\)rea \(\equiv \lambda (A,A'). (\neg \text{ok } A \land (\text{tr } A \leq \text{tr } A') \lor (\text{ok } A \land (\text{tr } A \leq \text{tr } A') \land (\text{ref } A = \text{ref } A') \land (\text{more } A = \text{more } A')\))

definition \(R3::((\'\vartheta,\'\sigma) \text{ alphabet-rp})\) \(\text{Healthiness-condition}\)
where \(R3(P) \equiv (\Pi\)rea \(\triangleleft \text{wait o fst } \triangleright\ P)\)

definition \(R::((\'\vartheta,\'\sigma) \text{ alphabet-rp})\) \(\text{Healthiness-condition}\)
where \(R \equiv R3 \circ R2 \circ R1\)

lemmas \(\text{rp-defs} = R1\)-def \(R2\)-def \(\Pi\)rea-def \(R3\)-def \(R\)-def \spec-def

11.3 Proofs

lemma \(\text{tr-filter-empty} \ [\text{simp}]: \text{tr-filter } l \ \{\} = l\)
\(\text{⟨proof}⟩\)

lemma \(\text{trf-imp-filters}: [xs = \text{tr-filter } ys \ cs; \ xs \neq []] \implies \text{filter-chan-set } (\text{hd } xs) \ cs\)
\(\text{⟨proof}⟩\)

lemma \(\text{filtercs-imp-trf}: [\text{filter-chan-set } x \ cs; \ xs = \text{tr-filter } ys \ cs] \implies x\#xs = \text{tr-filter } (x\#ys) \ cs\)
\(\text{⟨proof}⟩\)

lemma \(\text{alpha-d-more-eqI}:
\text{assumes } \text{tr } r = \text{tr } r' \land \text{wait } r = \text{wait } r' \land \text{ref } r = \text{ref } r' \land \text{more } r = \text{more } r'\)
\text{shows } \text{alpha-d-more } r = \text{alpha-d-more } r'\)
\(\text{⟨proof}⟩\)

lemma \(\text{alpha-d-more-eqE}:
\text{assumes } \text{alpha-d-more } r = \text{alpha-d-more } r'\)
\text{obtains } \text{tr } r = \text{tr } r' \land \text{wait } r = \text{wait } r' \land \text{ref } r = \text{ref } r' \land \text{more } r = \text{more } r'\)
\(\text{⟨proof}⟩\)

lemma \(\text{alpha-rp-eqE}:
\text{assumes } r = r'\)

41
obtains \( ok \ r = ok \ r' \) \( tr \ r = tr \ r' \) \( wait \ r = wait \ r' \) \( ref \ r = ref \ r' \) \( more \ r = more \ r' \)
\( (proof) \)

**lemma** \( R\text{-idem} \): \( R \circ R = R \)
\( (proof) \)

**lemma** \( R\text{-idem2} \): \( R (R P) = R P \)
\( (proof) \)

**lemma** \( R1\text{-idem} \): \( R1 \circ R1 = R1 \)
\( (proof) \)

**lemma** \( R1\text{-idem2} \): \( R1 (R1 x) = R1 x \)
\( (proof) \)

**lemma** \( R2\text{-idem} \): \( R2 \circ R2 = R2 \)
\( (proof) \)

**lemma** \( R2\text{-idem2} \): \( R2 (R2 x) = R2 x \)
\( (proof) \)

**lemma** \( R3\text{-idem} \): \( R3 \circ R3 = R3 \)
\( (proof) \)

**lemma** \( R3\text{-idem2} \): \( R3 (R3 x) = R3 x \)
\( (proof) \)

**lemma** \( R1\text{-}R2\text{-commute} \): \( (R1 \circ R2) = (R2 \circ R1) \)
\( (proof) \)

**lemma** \( R1\text{-}R3\text{-commute} \): \( (R1 \circ R3) = (R3 \circ R1) \)
\( (proof) \)

**lemma** \( R2\text{-}R3\text{-commute} \): \( R2 \circ R3 = R3 \circ R2 \)
\( (proof) \)

**lemma** \( R\text{-abs}\text{-}R1 \): \( R \circ R1 = R \)
\( (proof) \)

**lemma** \( R\text{-abs}\text{-}R2 \): \( R \circ R2 = R \)
\( (proof) \)

**lemma** \( R\text{-abs}\text{-}R3 \): \( R \circ R3 = R \)
\( (proof) \)

**lemma** \( R\text{-is}\text{-}R1 \):
\* assumes \( A \): \( P \) is \( R \) healthy
\* shows \( P \) is \( R1 \) healthy
lemma \( R\text{-is-R2} \):
\[ \text{assumes } A: \text{P is R healthy} \]
\[ \text{shows } P \text{ is } R^2 \text{ healthy} \]
\[ \langle \text{proof} \rangle \]

lemma \( R\text{-is-R3} \):
\[ \text{assumes } A: \text{P is R healthy} \]
\[ \text{shows } P \text{ is } R^3 \text{ healthy} \]
\[ \langle \text{proof} \rangle \]

lemma \( R\text{-disj} \):
\[ \text{assumes } A: \text{P is R healthy} \]
\[ \text{assumes } B: \text{Q is R healthy} \]
\[ \text{shows } (P \lor Q) \text{ is R healthy} \]
\[ \langle \text{proof} \rangle \]

lemma \( R\text{-disj2} \):
\[ R (P \lor Q) = (R P \lor R Q) \]
\[ \langle \text{proof} \rangle \]

lemma \( R\text{-comp} \):
\[ \text{assumes } P \text{ is } R^1 \text{ healthy} \]
\[ \text{and } Q \text{ is } R^1 \text{ healthy} \]
\[ \text{shows } (P; ; Q) \text{ is } R^1 \text{ healthy} \]
\[ \langle \text{proof} \rangle \]

lemma \( R\text{-comp2} \):
\[ \text{assumes } A: \text{P is } R^1 \text{ healthy} \]
\[ \text{assumes } B: \text{Q is } R^1 \text{ healthy} \]
\[ \text{shows } R^1 (P; ; Q) = ((R^1 P); ; Q) \]
\[ \langle \text{proof} \rangle \]

lemma \( J\text{-is-R1} \):
\[ J \text{ is } R^1 \text{ healthy} \]
\[ \langle \text{proof} \rangle \]

lemma \( J\text{-is-R2} \):
\[ J \text{ is } R^2 \text{ healthy} \]
\[ \langle \text{proof} \rangle \]

lemma \( R1\text{-H2-commute2} \):
\[ R1 (H2 P) = H2 (R1 P) \]
\[ \langle \text{proof} \rangle \]

lemma \( R1\text{-H2-commute} \):
\[ R1 \circ H2 = H2 \circ R1 \]
\[ \langle \text{proof} \rangle \]

lemma \( R2\text{-H2-commute2} \):
\[ R2 (H2 P) = H2 (R2 P) \]
\[ \langle \text{proof} \rangle \]

lemma \( R2\text{-H2-commute} \):
\[ R2 \circ H2 = H2 \circ R2 \]
lemma R3-H2-commute2: $R3 (H2 P) = H2 (R3 P)$

lemma R3-H2-commute: $R3 \circ H2 = H2 \circ R3$

lemma R-join:
  assumes $x$ is R healthy
  and $y$ is R healthy
  shows $(x \cap y)$ is R healthy

lemma R-meet:
  assumes $A: x$ is R healthy
  and $B:y$ is R healthy
  shows $(x \cup y)$ is R healthy

lemma R-H2-commute: $R \circ H2 = H2 \circ R$

lemma R-H2-commute2: $R (H2 P) = H2 (R P)$

end

12 CSP processes

theory CSP-Processes
imports Reactive-Processes
begin

A CSP process is a UTP reactive process that satisfies two additional healthiness conditions called CSP1 and CSP2. A reactive process that satisfies CSP1 and CSP2 is said to be CSP healthy.

12.1 Definitions

We introduce here the definitions of the CSP healthiness conditions.

definition CSP1:=$(\emptyset,\sigma)$ alphabet-rp) Healthiness-condition
where CSP1 $(P) \equiv P \lor (\lambda(A, A'). \neg ok A \land tr A \leq tr A')$

definition J-csp
where J-csp $\equiv \lambda(A, A'). (ok A \rightarrow ok A') \land tr A = tr A' \land wait A = wait A' \land ref A = ref A' \land more A = more A'$
**definition** CSP2::((′ϑ,′σ) alphabet-rp) Healthiness-condition

**where** CSP2 (P) ≡ P ;; J-csp

**definition** is-CSP-process::(′ϑ,′σ) relation-rp ⇒ bool where

is-CSP-process P ≡ P is CSP1 healthy ∧ P is CSP2 healthy ∧ P is R healthy

**lemmas** csp-defs = CSP1-def J-csp-def CSP2-def is-CSP-process-def

**lemma** is-CSP-processE1 [elim?]:
**assumes** is-CSP-process P
**obtains** P is CSP1 healthy P is CSP2 healthy P is R healthy
**proof**

**lemma** is-CSP-processE2 [elim?]:
**assumes** is-CSP-process P
**obtains** CSP1 P = P CSP2 P = P R P = P
**proof**

### 12.2 Proofs

Theorems and lemmas relative to CSP processes are introduced here.

**lemma** CSP1-CSP2-commute: CSP1 o CSP2 = CSP2 o CSP1
**proof**

**lemma** CSP2-is-H2: H2 = CSP2
**proof**

**lemma** H2-CSP1-commute: H2 o CSP1 = CSP1 o H2
**proof**

**lemma** H2-CSP1-commute2: H2 (CSP1 P) = CSP1 (H2 P)
**proof**

**lemma** CSP1-R-commute:
CSP1 (R P) = R (CSP1 P)
**proof**

**lemma** CSP2-R-commute:
CSP2 (R P) = R (CSP2 P)
**proof**

**lemma** CSP1-idem: CSP1 = CSP1 o CSP1
**proof**

**lemma** CSP2-idem: CSP2 = CSP2 o CSP2
**proof**

**lemma** CSP-is-CSP1:
assumes $A$: is-CSP-process $P$
shows $P$ is CSP1 healthy
(proof)

lemma CSP-is-CSP1:
assumes $A$: is-CSP-process $P$
shows $P$ is CSP2 healthy
(proof)

lemma CSP-is-R:
assumes $A$: is-CSP-process $P$
shows $P$ is R healthy
(proof)

lemma t-or-f-a: $P(a, b) \implies ((P(a, b\{ok := True\})) \lor (P(a, b\{ok := False\})))$
(proof)

lemma CSP2-ok-a:
$(CSP_2 P)(a, b\{ok := True\}) \implies (P(a, b\{ok := True\}) \lor P(a, b\{ok := False\}))$
(proof)

lemma CSP2-ok-b:
$(P(a, b\{ok := True\}) \lor P(a, b\{ok := False\})) \implies (CSP_2 P)(a, b\{ok := True\})$
(proof)

lemma CSP2-ok:
$(CSP_2 P)(a, b\{ok := True\}) = (P(a, b\{ok := True\}) \lor P(a, b\{ok := False\}))$
(proof)

lemma CSP2-notok-a: $(CSP_2 P)(a, b\{ok := False\}) \implies P(a, b\{ok := False\})$
(proof)

lemma CSP2-notok-b: $P(a, b\{ok := False\}) \implies (CSP_2 P)(a, b\{ok := False\})$
(proof)

lemma CSP2-notok: $(CSP_2 P)(a, b\{ok := False\}) = P(a, b\{ok := False\})$
(proof)

lemma CSP2-t-f:
assumes $A$: $(CSP_2 (R (r \vdash p)))(a, b)$
and $B$: $((CSP_2 (R (r \vdash p)))(a, b\{ok := False\})) \lor ((CSP_2 (R (r \vdash p)))(a, b\{ok := True\})) \implies Q$
shows $Q$
(proof)

lemma disj-CSP1:
assumes $P$ is CSP1 healthy
and $Q$ is CSP1 healthy
shows $(P \lor Q)$ is CSP1 healthy

46
lemma disj-CSP2:  
P is CSP2 healthy ===\rightarrow Q is CSP2 healthy ===\rightarrow (P \lor Q) is CSP2 healthy
\langle proof \rangle

lemma disj-CSP:  
assumes A: is-CSP-process P  
assumes B: is-CSP-process Q  
shows is-CSP-process (P \lor Q)
\langle proof \rangle

lemma seq-CSP1:  
assumes A: P is CSP1 healthy  
assumes B: Q is CSP1 healthy  
shows (P ; ; Q) is CSP1 healthy
\langle proof \rangle

lemma seq-CSP2:  
assumes A: Q is CSP2 healthy  
shows (P ; ; Q) is CSP2 healthy
\langle proof \rangle

lemma seq-R:  
assumes P is R healthy  
and Q is R healthy  
shows (P ; ; Q) is R healthy
\langle proof \rangle

lemma seq-CSP:  
assumes A: P is CSP1 healthy  
and B: P is R healthy  
and C: is-CSP-process Q  
shows is-CSP-process (P ; ; Q)
\langle proof \rangle

lemma rd-ind-wait: (R(\neg(P f \downarrow) \vdash (P ^{\downarrow})))  
= (R(\neg(\lambda (A, A'). P (A, A'\langle ok := False\rangle)))  
\vdash (\lambda (A, A'). P (A, A'\langle ok := True\rangle)))
\langle proof \rangle

lemma rd-H1: (R((\neg(\lambda (A, A'). P (A, A'\langle ok := False\rangle)))  
\vdash (\lambda (A, A'). P (A, A'\langle ok := True\rangle)))  
= (R ((\neg H1 (\lambda (A, A'). P (A, A'\langle ok := False\rangle)))  
\vdash H1 (\lambda (A, A'). P (A, A'\langle ok := True\rangle))))
\langle proof \rangle

lemma rd-H1-H2: (R((\neg H1 (\lambda (A, A'). P (A, A'\langle ok := False\rangle))))
⊢ H1 (λ (A, A'). P (A, A'⟨ok := True⟩))) =
(R((¬(H1 o H2) (λ (A, A'). P (A, A'⟨ok := False⟩))))
⊢ (H1 o H2) (λ (A, A'). P (A, A'⟨ok := True⟩))))
⟨proof⟩

lemma rd-H1-H2-R-H1-H2:
(R ((¬ (H1 o H2) (λ (A, A'). P (A, A'⟨ok := False⟩))))
⊢ (H1 o H2) (λ (A, A'). P (A, A'⟨ok := True⟩)))) =
(R o H1 o H2) P
⟨proof⟩

lemma CSP1-is-R1-H1:
assumes P is R1 healthy
shows CSP1 P = R1 (H1 P)
⟨proof⟩

lemma CSP1-is-R1-H1-2: CSP1 (R1 P) = R1 (H1 P)
⟨proof⟩

lemma CSP1-R1-commute: CSP1 o R1 = R1 o CSP1
⟨proof⟩

lemma CSP1-R1-commute2: CSP1 (R1 P) = R1 (CSP1 P)
⟨proof⟩

lemma CSP1-is-R1-H1-b:
(P = (R o R1 o H1 o H2) P) = (P = (R o CSP1 o H2) P)
⟨proof⟩

lemma CSP1-join:
assumes A: x is CSP1 healthy
and B: y is CSP1 healthy
shows (x ⊓ y) is CSP1 healthy
⟨proof⟩

lemma CSP2-join:
assumes A: x is CSP2 healthy
and B: y is CSP2 healthy
shows (x ⊓ y) is CSP2 healthy
⟨proof⟩

lemma CSP1-meet:
assumes A: x is CSP1 healthy
and B: y is CSP1 healthy
shows (x ⊔ y) is CSP1 healthy
⟨proof⟩

lemma CSP2-meet:
assumes A: x is CSP2 healthy
and B: y is CSP2 healthy
shows (x ⊔ y) is CSP2 healthy
(proof)

lemma CSP-join:
- assumes A: is-CSP-process x
- and B: is-CSP-process y
shows is-CSP-process (x ⊓ y)
(proof)

lemma CSP-meet:
- assumes A: is-CSP-process x
- and B: is-CSP-process y
shows is-CSP-process (x ⊔ y)
(proof)

12.3 CSP processes and reactive designs

In this section, we prove the relation between CSP processes and reactive
designs.

lemma rd-is-CSP1: (R (r ⊢ p)) is CSP1 healthy
(proof)

lemma rd-is-CSP2:
- assumes A: ∀ a b. r (a, bOK := True) → r (a, bOK := False)
shows (R (r ⊢ p)) is CSP2 healthy
(proof)

lemma rd-is-CSP:
- assumes A: ∀ a b. r (a, bOK := True) → r (a, bOK := False)
shows is-CSP-process (R (r ⊢ p))
(proof)

lemma CSP-is-rd:
- assumes A: is-CSP-process P
shows P = (R (¬(P f ⊢ f)) ⊢ (P t f))
(proof)

eend

13 Circus actions

theory Circus-Actions
imports HOLCF CSP-Processes
begin

In this section, we introduce definitions for Circus actions with some useful
theorems and lemmas.

\textbf{default-sort type}

13.1 Definitions

The Circus actions type is defined as the set of all the CSP healthy reactive processes.

\textbf{typedef} (\vartheta::ev-eq, \sigma) \text{ action } = \{p::(\vartheta,\sigma) \text{ relation-rp. is-CSP-process } p\}

\textbf{morphism} relation-of action-of

\textbf{print-theorems}

The 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 (morphisms) definitions \textit{relation_of} and \textit{action_of} as well as the usual axioms expressing the bijection \textit{action_of (relation_of ?x)} = ?x and ?y \in \{p. is-CSP-process p\} \implies \textit{relation_of (action_of ?y)} = ?y.

\textbf{lemma} relation-of-CSP: is-CSP-process (relation-of x)

\textbf{lemma} relation-of-CSP1: (relation-of x) is CSP1 healthy

\textbf{lemma} relation-of-CSP2: (relation-of x) is CSP2 healthy

\textbf{lemma} relation-of-R: (relation-of x) is R healthy

13.2 Proofs

In the following, Circus actions are proved to be an instance of the \textit{Complete_Lattice} class.

\textbf{lemma} relation-of-spec-f-f:

\forall a, b. (relation-of y \rightarrow relation-of x) (a, b) \implies

\text{(relation-of y)}{f} (a|tr := [], b) \implies

\text{(relation-of x)}{f} (a|tr := [], b)

\textbf{proof}

\textbf{lemma} relation-of-spec-t-f:

\forall a, b. (relation-of y \rightarrow relation-of x) (a, b) \implies

\text{(relation-of y)}{t} (a|tr := [], b) \implies

\text{(relation-of x)}{t} (a|tr := [], b)

\textbf{proof}
instantiation action::(ev-eq, type) below
begin
  definition ref-def : P ⊆ Q ≡ [(relation-of Q) → (relation-of P)]
instance ⟨proof⟩
end

instance action :: (ev-eq, type) po ⟨proof⟩

instantiation action :: (ev-eq, type) lattice
begin
  definition inf-action : (inf P Q ≡ action-of ((relation-of P) ∩ (relation-of Q)))
  definition sup-action : (sup P Q ≡ action-of ((relation-of P) ∪ (relation-of Q)))
  definition less-eq-action : (less-eq (P::('a, 'b) action) Q ≡ P ⊆ Q)
  definition less-action : (less (P::('a, 'b) action) Q ≡ P ⊆ Q ∧ ¬Q ⊆ P)
instance ⟨proof⟩
end

lemma bot-is-action: R (false ⊢ true) ∈ {p. is-CSP-process p} ⟨proof⟩

lemma bot-eq-true: R (false ⊢ true) = R true ⟨proof⟩

instantiation action :: (ev-eq, type) bounded-lattice
begin
  definition bot-action : (bot::('a, 'b) action) ≡ action-of (R(false ⊢ true))
  definition top-action : (top::('a, 'b) action) ≡ action-of (R(true ⊢ false))
instance ⟨proof⟩
end

lemma relation-of-top: relation-of top = R(true ⊢ false) ⟨proof⟩

lemma relation-of-bot: relation-of bot = R true ⟨proof⟩

lemma non-emptyE: assumes A ≠ {} obtains x where x : A ⟨proof⟩
lemma CSP1-Inf:
assumes ∗:A ≠ \{\}
shows (⨅ relation-of ' A) is CSP1 healthy
(proof)

lemma CSP2-Inf:
assumes ∗:A ≠ \{\}
shows (⨅ relation-of ' A) is CSP2 healthy
(proof)

lemma R-Inf:
assumes ∗:A ≠ \{\}
shows (⨅ relation-of ' A) is R healthy
(proof)

lemma CSP-Inf:
assumes A ≠ \{\}
shows is-CSP-process (⨅ relation-of ' A)
(proof)

lemma Inf-is-action: A ≠ \{\} \implies (⨅ relation-of ' A) \in \{p. is-CSP-process p\}
(proof)

lemma CSP1-Sup: A ≠ \{\} \implies (⨆ relation-of ' A) is CSP1 healthy
(proof)

lemma CSP2-Sup: A ≠ \{\} \implies (⨆ relation-of ' A) is CSP2 healthy
(proof)

lemma R-Sup: A ≠ \{\} \implies (⨆ relation-of ' A) is R healthy
(proof)

lemma CSP-Sup: A ≠ \{\} \implies is-CSP-process (⨆ relation-of ' A)
(proof)

lemma Sup-is-action: A ≠ \{\} \implies (⨆ relation-of ' A) \in \{p. is-CSP-process p\}
(proof)

lemma relation-of-Sup:
A ≠ \{\} \implies relation-of (action-of (⨆ relation-of ' A)) = (⨆ relation-of ' A)
(proof)

instantiation action :: (ev-eq, type) complete-lattice
begin

definition Sup-action :
(Sup (S:: ('a, 'b) action set) ≡ if S=\{\} then bot else action-of (⨆ (relation-of ' S)))
definition Inf-action :
(Inf (S:: ('a, 'b) action set) ≡ if S=\{\} then top else action-of (⨅ (relation-of ' S)))
instance
(proof)
end
end

14 Circus variables

theory Var-list
imports Main
begin

Circus variables are represented by a stack (list) of values. They are characterized by two functions, select and update. The Circus variable type is defined as a tuple (select * update) with a list of values instead of a single value.

type-synonym ('a, 'σ) var-list = ('σ ⇒ 'a list) * ('a list ⇒ 'a list ⇒ 'σ ⇒ 'σ)

The select function returns the top value of the stack.

definition select :: ('a, 'r) var-list ⇒ 'r ⇒ 'a
where select f ≡ λ A. hd ((fst f) A)

The increase function pushes a new value to the top of the stack.

definition increase :: ('a, 'r) var-list ⇒ 'a ⇒ 'r ⇒ 'r
where increase f val ≡ (snd f) (λ l. val#l)

The increase0 function pushes an arbitrary value to the top of the stack.

definition increase0 :: ('a, 'r) var-list ⇒ 'r ⇒ 'r
where increase0 f ≡ (snd f) (λ l. ((SOME val. True)#l))

The decrease function pops the top value of the stack.

definition decrease :: ('a, 'r) var-list ⇒ 'r ⇒ 'r
where decrease f ≡ (snd f) (λ l. (tl l))

The update function updates the top value of the stack.

definition update :: ('a, 'r) var-list ⇒ ('a ⇒ 'a) ⇒ 'r ⇒ 'r
where update f upd ≡ (snd f) (λ l. (upd (hd l)#(tl l))

The update0 function initializes the top of the stack with an arbitrary value.

definition update0 :: ('a, 'r) var-list ⇒ 'r ⇒ 'r
where update0 f ≡ (snd f) (λ l. ((SOME upd. True) (hd l)#(tl l))

axiomatization where select-increase: (select v (increase v a s)) = a

The VAR–LIST function allows to retrieve a Circus variable from its name.
syntax  \texttt{VAR-LIST :: id \Rightarrow (\textquoteleft a, \textquoteleft r) \text{ var-list} \ (\text{VAR'-LIST} -)}

translations  \texttt{VAR-LIST} \text{ x =\Rightarrow (x, \text{-update-name x)}

end

15 Denotational semantics of Circus actions

theory \textit{Denotational-Semantics}

imports \textit{Circus-Actions Var-list}

begin

In this section, we introduce the definitions of Circus actions denotational semantics. We provide the proof of well-formedness of every action. We also provide proofs concerning the monotonicity of operators over actions.

15.1 Skip

definition \textit{Skip} \text{ :: (\textquoteleft \text{\var-

list}\textquoteleft\text{, said\textquoteleft\text{)} \text{ action\ where}}

Skip \equiv \text{ action-of} \text{(R (true \vdash \lambda(A, A'). tr A' = tr A \land \neg wait A' \land more A = more A'))}

lemma \textit{Skip-is-action}:

\begin{align*}
(R (true \vdash \lambda(A, A'). tr A' = tr A \land \neg wait A' \land more A = more A')) \in \{p. \text{is-CSP-process p}\}
\end{align*}

\langle \text{proof} \rangle

lemmas \textit{Skip-is-CSP} = \textit{Skip-is-action[simplified]}

\langle \text{proof} \rangle

lemma \textit{relation-of-Skip}:

\begin{align*}
\text{relation-of Skip} = (R (true \vdash \lambda(A, A'). tr A' = tr A \land \neg wait A' \land more A = more A'))
\end{align*}

\langle \text{proof} \rangle

definition \textit{CSP3} :: ((\textquoteleft \var-

list\textquoteleft\text{, said\textquoteleft\text{)} \text{ alphabet-rp) \text{Healthiness-condition\ where}}

CSP3 (P) \equiv \text{ relation-of Skip ; ; P}

definition \textit{CSP4} :: ((\textquoteleft \var-

list\textquoteleft\text{, said\textquoteleft\text{)} \text{ alphabet-rp) \text{Healthiness-condition\ where}}

CSP4 (P) \equiv P ; ; \text{relation-of Skip}

lemma \textit{Skip-is-CSP3} : (\text{relation-of Skip}) \text{ is CSP3 healthy}

\langle \text{proof} \rangle

lemma \textit{Skip-is-CSP4} : (\text{relation-of Skip}) \text{ is CSP4 healthy}

\langle \text{proof} \rangle

lemma \textit{Skip-comp-absorb} : (\text{relation-of Skip ; ; relation-of Skip}) = \text{relation-of Skip}

\langle \text{proof} \rangle

54
15.2 Stop

definition Stop :: ('θ::ev-eq,'σ) action
where Stop ≡ action-of (R (true ⊢ λ(A, A'). tr A' = tr A ∧ wait A'))

lemma Stop-is-action:
(R (true ⊢ λ(A, A'). tr A' = tr A ∧ wait A')) ∈ {p. is-CSP-process p}
⟨proof⟩

lemmas Stop-is-CSP = Stop-is-action[simplified]

lemma relation-of-Stop:
relation-of Stop = (R (true ⊢ λ(A, A'). tr A' = tr A ∧ wait A'))
⟨proof⟩

lemma Stop-is-CSP3: (relation-of Stop) is CSP3 healthy
⟨proof⟩

lemma Stop-is-CSP4: (relation-of Stop) is CSP4 healthy
⟨proof⟩

15.3 Chaos

definition Chaos :: ('θ::ev-eq,'σ) action
where Chaos ≡ action-of (R(false ⊢ true))

lemma Chaos-is-action: (R(false ⊢ true)) ∈ {p. is-CSP-process p}
⟨proof⟩

lemmas Chaos-is-CSP = Chaos-is-action[simplified]

lemma relation-of-Chaos: relation-of Chaos = (R(false ⊢ true))
⟨proof⟩

15.4 State update actions

definition Pre ::'σ relation ⇒ 'σ predicate
where Pre sc ≡ λA. ∃ A'. sc (A, A')

definition state-update-before :: 'σ relation ⇒ ('θ::ev-eq,'σ) action ⇒ ('θ,'σ) action
where state-update-before sc Ac = action-of(R ((λ(A, A'). (Pre sc) (more A)) ⊢
(λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')))
⟨; ; ; relation-of Ac⟩

lemma state-update-before-is-action:
(R ((λ(A, A'). (Pre sc) (more A)) ⊢
(λ(A, A').sc (more A, more A') & ¬wait A' & tr A = tr A))}
lemma relation-of-state-update-before:
relation-of (state-update-before sc Ac) = (R ((λ(A, A'). (Pre sc) (more A)) ⊨ (λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')) ; ; relation-of Ac)
(proof)

lemma mono-state-update-before: mono (state-update-before sc)
(proof)

lemma state-update-before-is-CSP:
relation-of (state-update-before sc Ac) ∈ \{p. is-CSP-process p\}
(proof)

lemmas state-update-before-is-CSP = state-update-before-is-action[simplified]

lemma relation-of-state-update-after:
relation-of (state-update-after sc Ac) = (R ((λ(A, A'). (Pre sc) (more A)) ⊨ (λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')) ; ; relation-of Ac)
(proof)

lemma mono-state-update-after: mono (state-update-after sc)
(proof)

lemma state-update-after-is-CSP:
relation-of (state-update-after sc Ac) ∈ \{p. is-CSP-process p\}
(proof)

lemmas state-update-after-is-CSP = state-update-after-is-action[simplified]

lemma relation-of-state-update-after:
relation-of (state-update-after sc Ac) = (relation-of Ac ; ; R (true ⊨ (λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')) ; ; relation-of Ac)
(proof)

lemma mono-state-update-after: mono (state-update-after sc)
(proof)

lemma state-update-after-is-CSP:
assumes A : relation-of Ac is CSP4 healthy
shows relation-of (state-update-after sc Ac) is CSP4 healthy
(proof)

definition state-update-after :: 'σ relation ⇒ ('σ::ev-eq,'σ) action ⇒ ('σ,'σ) action
where state-update-after sc Ac = action-of(relation-of Ac ; ; R (true ⊨ (λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')))

lemma state-update-after-is-action:
(relation-of Ac ; ; R (true ⊨ (λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')) ; ; relation-of Ac)
(proof)

lemmas state-update-after-is-CSP = state-update-after-is-action[simplified]

lemma relation-of-state-update-after:
relation-of (state-update-after sc Ac) = (relation-of Ac ; ; R (true ⊨ (λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A')) ; ; relation-of Ac)
(proof)

lemma mono-state-update-after: mono (state-update-after sc)
(proof)

lemma state-update-after-is-CSP:
assumes A : relation-of Ac is CSP3 healthy
shows relation-of (state-update-after sc Ac) is CSP3 healthy
(proof)
lemma state-update-after-is-CSP4: relation-of (state-update-after sc Ac) is CSP4 healthy
⟨proof⟩

15.5 Sequential composition

definition
Seq:: (′ϑ::ev-eq,′σ) action ⇒ (′ϑ,′σ) action ⇒ (′ϑ,′σ) action (infixl ′ ; ′ 24)
where P ′ ; ′ Q ≡ action-of (relation-of P ; ; relation-of Q)

lemma Seq-is-action: (relation-of P ; ; relation-of Q) ∈ {p. is-CSP-process p}
⟨proof⟩

lemmas Seq-is-CSP = Seq-is-action[simplified]

lemma relation-of-Seq: relation-of (P ′ ; ′ Q) = (relation-of P ; ; relation-of Q)
⟨proof⟩

lemma mono-Seq: mono (op ′ ; ′ P)
⟨proof⟩

lemma CSP3-imp-left-Skip:
  assumes A: relation-of P is CSP3 healthy
  shows (Skip ′ ; ′ P) = P
⟨proof⟩

lemma CSP4-imp-right-Skip:
  assumes A: relation-of P is CSP4 healthy
  shows (P ′ ; ′ Skip) = P
⟨proof⟩

lemma Seq-assoc: (A ′ ; ′ (B ′ ; ′ C)) = ((A ′ ; ′ B) ′ ; ′ C)
⟨proof⟩

lemma Skip-absorb: (Skip ′ ; ′ Skip) = Skip
⟨proof⟩

15.6 Internal choice

definition
Ndet:: (′ϑ::ev-eq,′σ) action ⇒ (′ϑ,′σ) action ⇒ (′ϑ,′σ) action (infixl ⊓ 18)
where P ⊓ Q ≡ action-of ((relation-of P) ∪ (relation-of Q))

lemma Ndet-is-action: ((relation-of P) ∪ (relation-of Q)) ∈ {p. is-CSP-process p}
⟨proof⟩

lemmas Ndet-is-CSP = Ndet-is-action[simplified]
lemma relation-of-Ndet: relation-of \((P \cap Q) = ((\text{relation-of } P) \lor (\text{relation-of } Q))\)
(proof)

lemma mono-Ndet: mono \((\text{op} \cap P)\)
(proof)

15.7 External choice

definition Det::\((\text{ev-eq},'\sigma)\) action ⇒ \((\text{ev-eq},'\sigma)\) action \((\text{infixl} [+])\)
where \(P [+] Q \equiv \text{action-of}(R((\neg((\text{relation-of } P) f f) \land \neg((\text{relation-of } Q) f f)) \vdash (((\text{relation-of } P) f f \land (\text{relation-of } Q) f f)) \triangleright \lambda (A, A'). \text{tr} A = \text{tr} A' \land \text{wait} A' \triangleright \left(((\text{relation-of } P) f f \lor (\text{relation-of } Q) f f))\right)))\)
notation(xsymbol)

Det (\text{infixl} \Box 18)

lemma Det-is-action:
\((R((\neg((\text{relation-of } P) f f) \land \neg((\text{relation-of } Q) f f)) \vdash (((\text{relation-of } P) f f \land (\text{relation-of } Q) f f)) \triangleright \lambda (A, A'). \text{tr} A = \text{tr} A' \land \text{wait} A' \triangleright \left(((\text{relation-of } P) f f \lor (\text{relation-of } Q) f f))\right)))\) \(\in \{p. \text{is-CSP-process } p\}\)
(proof)

lemmas Det-is-CSP = Det-is-action[simplified]

lemma relation-of-Det:
relation-of \((P \Box Q) = (R((\neg((\text{relation-of } P) f f) \land \neg((\text{relation-of } Q) f f)) \vdash (((\text{relation-of } P) f f \land (\text{relation-of } Q) f f)) \triangleright \lambda (A, A'). \text{tr} A = \text{tr} A' \land \text{wait} A' \triangleright \left(((\text{relation-of } P) f f \lor (\text{relation-of } Q) f f))\right)))\)
(proof)

lemma mono-Det: mono \((\text{op} [+]) P)\)
(proof)

15.8 Reactive design assignment

definition rd-assign s = action-of \((R (\text{true} \vdash \lambda (A, A'). \text{ref} A' = \text{ref} A \land \text{tr} A' = \text{tr} A \land \neg \text{wait} A' \land \text{more} A' = s))\)

lemma rd-assign-is-action:
\((R (\text{true} \vdash \lambda (A, A'). \text{ref} A' = \text{ref} A \land \text{tr} A' = \text{tr} A \land \neg \text{wait} A' \land \text{more} A' = s))\) \(\in \{p. \text{is-CSP-process } p\}\)
(proof)

lemmas rd-assign-is-CSP = rd-assign-is-action[simplified]
lemma relation-of-rd-assign:
relation-of (rd-assign s) =
(R (true ⊢ λ(A, A'). ref A' = ref A ∧ tr A' = tr A ∧ ¬wait A' ∧
more A' = s))
⟨proof⟩

15.9 Local state external choice
definition Loc :: 'σ local-state ⇒ ('σ' ϑ :: ev-eq, 'σ) action ⇒ ('σ, 'σ) action
where (loc s1 • P) ⊕ (loc s2 • Q) ≡ ((rd-assign s1); 'P) □ ((rd-assign s2); ' Q)

15.10 Schema expression
definition Schema :: 'σ relation ⇒ ('σ' ϑ :: ev-eq, 'σ) action
where Schema sc ≡ action-of
(R ((λ(A, A'). (Pre sc) (more A)) ⊢
(λ(A, A'). sc (more A, more A') ∧ ¬wait A' ∧ tr A = tr A')))
lemma Schema-is-action:
(R ((λ(A, A'). (Pre sc) (more A)) ⊢
(λ(A, A'). sc (more A, more A') ∧ ¬wait A' ∧ tr A = tr A')))
∈ {p. is-CSP-process p}
⟨proof⟩

lemmas Schema-is-CSP = Schema-is-action[simplified]

lemma relation-of-Schema:
relation-of (Schema sc) = (R ((λ(A, A'). (Pre sc) (more A)) ⊢
(λ(A, A'). sc (more A, more A') ∧ ¬wait A' ∧ tr A = tr A')))
⟨proof⟩

lemma Schema-is-state-update-before: Schema u = state-update-before u Skip
⟨proof⟩

15.11 Parallel composition
type-synonym 'σ local-state = ('σ × ('σ ⇒ 'σ ⇒ 'σ))

fun MergeSt :: 'σ local-state ⇒ 'σ local-state ⇒ ('σ, 'σ) relation-rp
where MergeSt (s1, s1') (s2, s2') = ((λ(S, S'). (s1' s1) (more S) = more S');
(λ(S::'σ) alphabet-rp, S'). (s2' s2) (more S) = more S'))
definition listCons :: 'σ ⇒ 'σ list ⇒ 'σ list ⇒ 'σ list list
where a ## l = ((map (Cons a)) l)

fun ParMergel :: 'σ::ev-eq list ⇒ 'σ list ⇒ 'σ set ⇒ 'σ list list
where ParMergel [] [] cs = []
| ParMerge [] (b#tr2) cs = (if (filter-chan-set b cs) then []
else (b ## (ParMerge [] tr2 cs))) |
| ParMerge (a#tr1) [] cs = (if (filter-chan-set a cs) then []
else (a ## (ParMerge tr1 [] cs))) |
| ParMerge (a#tr1) (b#tr2) cs = (if (filter-chan-set a cs) then (if (ev-eq a b)
then (a ## (ParMerge tr1 tr2 cs))
else (if (filter-chan-set b cs)
then []
else (b ## (ParMerge (a#tr1) tr2 cs)))))
else (if (filter-chan-set b cs)
then (a ## (ParMerge tr1 (b#tr2) cs))
else (a ## (ParMerge tr1 (b#tr2) cs)))
@ (b ## (ParMerge (a#tr1) tr2 cs))) |

**Definition** ParMerge::'θ::{ev-eq list ⇒ 'θ list ⇒ 'θ set ⇒ 'θ list set where ParMerge tr1 tr2 cs = set (ParMerge tr1 tr2 cs) |

**Lemma** set-Cons1: tr1 ∈ set l ⇒ a ≠ tr1 ∈ op ≠ a ≠ set l |
(Proof) |

**Lemma** tr-in-set-eq: (tr1 ∈ op ≠ b ≠ set l) = (tr1 ≠ [] ∧ hd tr1 = b ∧ tl tr1 ∈ set l) |
(Proof) |

**Definition** M-par::'θ::{ev-eq, 'σ} alpha-rp-scheme ⇒ ('σ ⇒ 'σ) ⇒ ('σ, 'σ) alpha-rp-scheme ⇒ ('σ ⇒ 'σ) ⇒ ('σ set ⇒ ('θ, 'σ) relation-rp |
where M-par s1 x1 s2 x2 cs = ((λ(S, S')). ((diff-tr s1' S) (diff-tr s2 S) cs & ev-eq (tr-filter (tr s1) cs) (tr-filter (tr s2) cs)) ∩ ((λ(S, S')). (wait s1 ∨ wait s2) ∧ ref S' ⊆ (((ref s1) ∪ (ref s2)) ∩ cs) ∪ (((ref s1) ∩ (ref s2)) − cs))) |
< wait o snd >> ((λ(S, S')). (¬ wait s1 ∨ ¬ wait s2)) ∧ MergeSt (((more s1), x1) (((more s2), x2))) |

**Definition** Par::'θ::{ev-eq, 'σ} action ⇒
(′σ ⇒ 'σ ⇒ 'σ) ⇒ 'θ set ⇒ ('σ ⇒ 'σ ⇒ 'σ) ⇒ ('θ, 'σ) action ⇒ (′θ, 'σ) action (- [] - | - | - ) |
where A1 [] ns1 | cs | ns2 [] A2 ≡ (action-of (R ((λ (S, S')). |
(∃ tr1 tr2. ((relation-of A1) f; ; (λ (S, S'). tr1 = (tr S))) (S, S')) |
∧ (spec False (wait S) (relation-of A2) ; ; (λ (S, -). tr2 = (tr S))) (S, S') |
∧ ((tr-filter tr1 cs) = (tr-filter tr2 cs))) ∧ |
(∃ tr1 tr2. (spec False (wait S) (relation-of A1); ; (λ(S, -). tr1 = tr S)) (S, S') |
∧ ((relation-of A2) f; ; (λ(S, S'). tr2 = (tr S))) (S, S') |
∧ ((tr-filter tr1 cs) = (tr-filter tr2 cs))) |
(λ (S, S′). (∃ s1 s2. ((λ (A, A). (relation-of A1))f (A, s1))
∧ ((relation-of A2))f (A, s2))) ; M-par s1 ns1 s2 ns2 cs (S, S′)))))))

lemma Par-is-action: \( R ((λ (S, S')).)
\n→ (∃ tr1 tr2. ((relation-of A1)f ; (λ (S, S'). tr1 = (tr S)) (S, S′))
∧ (spec False (wait S) (relation-of A2) ; ; (λ (S, S′). tr2 = (tr S))) (S, S′))
∧ ((tr-filter tr1 cs) = (tr-filter tr2 cs))) ∧
→ (∃ tr1 tr2. (spec False (wait S) (relation-of A1); ; (λ(S, -). tr1 = tr S)) (S, S′))
∧ ((relation-of A2)f ; ; (λ (S, S′). tr2 = (tr S))) (S, S′)
∧ ((tr-filter tr1 cs) = (tr-filter tr2 cs))) ⊨
(λ (S, S′). (∃ s1 s2. ((λ (A, A). (relation-of A1)f (A, s1))
∧ ((relation-of A2)f (A, s2))) ; M-par s1 ns1 s2 ns2 cs (S, S′)))) ∈ {p.
is-CSP-process p}
(proof)

lemmas Par-is-CSP = Par-is-action[simplified]

lemma relation-of-Par:
relation-of (A1 [ ns1 | cs | ns2 ] A2) = \( R ((λ (S, S')).)
\n→ (∃ tr1 tr2. ((relation-of A1)f ; ; (λ (S, S′). tr1 = (tr S)) (S, S′))
∧ (spec False (wait S) (relation-of A2) ; ; (λ (S, S′). tr2 = (tr S))) (S, S′))
∧ ((tr-filter tr1 cs) = (tr-filter tr2 cs))) ∧
→ (∃ tr1 tr2. (spec False (wait S) (relation-of A1); ; (λ(S, -). tr1 = tr S)) (S, S′))
∧ ((relation-of A2)f ; ; (λ (S, S′). tr2 = (tr S))) (S, S′)
∧ ((tr-filter tr1 cs) = (tr-filter tr2 cs))) ⊨
(λ (S, S′). (∃ s1 s2. ((λ (A, A). (relation-of A1)f (A, s1))
∧ ((relation-of A2)f (A, s2))) ; M-par s1 ns1 s2 ns2 cs (S, S′))))
(proof)

lemma mono-Par: mono (λQ. P [ ns1 | cs | ns2 ] Q)
(proof)

15.12 Local parallel block

definition ParLoc::σ ⇒ (σ ⇒ σ ⇒ σ) ⇒ (σ::ev-eq, σ) action ⇒ σ set ⇒ σ ⇒ (σ ⇒ σ)
⇒ σ ⇒ (σ, σ) action ⇒ (σ, σ) action

(l′((par - | - ⋅ - ) [ - ] l′((par - | - ⋅ - )))

where
(par s1 | ns1 • P) [ cs ] (par s2 | ns2 • Q) ≡ ((rd-assign s1); ; P) [ ns1 | cs | ns2 ] ((rd-assign s2); ; Q)

15.13 Assignment

definition ASSIGN::(ν, σ) var-list ⇒ (σ ⇒ ν) ⇒ (ν::ev-eq, σ) action where
ASSIGN x e ≡ action-of \( R \ (true ⊨ (λ (S, S′). tr S′ = tr S ∧ ¬\text{wait} S′ ∧
\ (more S′ = (update x (λ- (e (more S))) (more S)))))))

syntax -assign::id ⇒ (σ ⇒ ν) ⇒ (ν, σ) action - := -
translations \( y \ := \ vv \rightarrow \text{CONST ASSIGN (VAR y) vv} \)

**Lemma** Assign-is-action:
\[
(R \ (\text{true} \vdash (\lambda \ (S, S'). \ tr S' = tr S \land \neg \text{wait} S' \land \\
\text{(more} S' = (\text{update} x (\lambda - (e \ (\text{more} S)))) \ (\text{more} S)))))) \in \{ p. \ is-CSP-process \ p \}
\]

**Proof**

**Lemmas** Assign-is-CSP = Assign-is-action[simplified]

**Lemma** relation-of-Assign:
\[
\text{relation-of} \ (\text{ASSIGN} \ x \ e) = (R \ (\text{true} \vdash (\lambda \ (S, S'). \ tr S' = tr S \land \neg \text{wait} S' \land \\
\text{(more} S' = (\text{update} x (\lambda - (e \ (\text{more} S)))) \ (\text{more} S))))))
\]

**Proof**

**Lemma** Assign-is-state-update-before: ASSIGN \ x \ e = state-update-before (\lambda \ (s, s') . s' = (update x (\lambda - (e \ s))) s) Skip

**Proof**

### 15.14 Variable scope

**Definition** Var:: (\('v, \ 's) \ var-list \Rightarrow ('\emptyset, \ 's) \ action \Rightarrow ('\emptyset::ev-eq,'s) \ action \ where

\[
\text{Var} \ v \ A \equiv \text{action-of}(\begin{align*}
(R(\text{true} \vdash (\lambda \ (A, A'). \ \exists \ a. \ tr A' = tr A \land \neg \text{wait} A' \land \text{more} A' = (\text{increase} v a \ (\text{more} A))))); ; \\
(\text{relation-of} A); ;
(R(\text{true} \vdash (\lambda \ (A, A'). \ tr A' = tr A \land \neg \text{wait} A' \land \text{more} A' = (\text{decrease} v a \ (\text{more} A)))))))) \in \{ p. \ is-CSP-process \ p \}
\]

**Proof**

**Lemmas** Var-is-CSP = Var-is-action[simplified]

**Lemma** relation-of-Var:
\[
\text{relation-of} \ (\text{Var} \ v \ A) = \\
(\{R(\text{true} \vdash (\lambda \ (A, A'). \ \exists \ a. \ tr A' = tr A \land \neg \text{wait} A' \land \text{more} A' = (\text{increase} v a \ (\text{more} A))))); ; \\
(\text{relation-of} A); ;
(R(\text{true} \vdash (\lambda \ (A, A'). \ tr A' = tr A \land \neg \text{wait} A' \land \text{more} A' = (\text{decrease} v a \ (\text{more} A))))))
\]
lemma mono-Var : mono (Var x)
(proof)

definition Let::{('v, 'σ) var-list ⇒ ('θ, 'σ) action ⇒ ('θ::ev-eq,'σ) action} where
Let v A ≡ action-of((relation-of A; ;
    (R(true ⊢ (λ (A, A'). tr A' = tr A ∧ ¬wait A' ∧ more A' = (decrease v (more A))))))}

syntax -let::idt ⇒ ('ϑ, 'σ) action ⇒ ('ϑ, 'σ) action (let - • - [1000] 999)
translations let y • Act => CONST Let (VAR-LIST y) Act

lemma Let-is-action:
(relation-of A; ;
    (R(true ⊢ (λ (A, A'). tr A' = tr A ∧ ¬wait A' ∧ more A' = (decrease v (more A)))))) ∈ {p. is-CSP-process p}
(proof)

lemmas Let-is-CSP = Let-is-action[simplified]

lemma relation-of-Let:
relation-of (Let v A) =
(relation-of A; ;
    (R(true ⊢ (λ (A, A'). tr A' = tr A ∧ ¬wait A' ∧ more A' = (decrease v (more A))))))
(proof)

lemma mono-Let : mono (Let x)
(proof)

lemma Var-is-state-update-before: Var v A = state-update-before (λ (s, s’). ∃ a. s’ = increase v a s) (Let v A)
(proof)

lemma Let-is-state-update-after: Let v A = state-update-after (λ (s, s’). s’ = decrease v s) A
(proof)

15.15 Guarded action

definition Guard::'σ predicate ⇒ ('θ::ev-eq, 'σ) action ⇒ ('θ, 'σ) action (- 'κ ' -)
where g 'κ P ≡ action-of(R (((g o more o fst) → ¬ (relation-of P)⇩f)) ⊢
((g o more o fst) ∧ ((relation-of P)⇩f)) ∨
\[
((\neg (g \text{ more } o \text{ fst})) \land (\lambda \langle A, A' \rangle. \text{tr } A' = \text{tr } A \land \text{wait } A')))\]

**Lemma** Guard-is-action:

\[
(R \ ((prec) (g \text{ more } o \text{ fst}) \rightarrow \neg ((\text{relation-of } P^f_j)) \vdash \neg ((\text{relation-of } P^l_j)) \lor \neg (g \text{ more } o \text{ fst}) \land ((\text{relation-of } P^l_j)) \lor \neg (g \text{ more } o \text{ fst})) \land (\lambda \langle A, A' \rangle. \text{tr } A' = \text{tr } A \land \text{wait } A'))) \in \{p. \text{is-CSP-process } p\}
\]

**Proof**

**Lemmas** Guard-is-CSP = Guard-is-action[simplified]

**Lemma** relation-of-Guard:

relation-of (g ‘&‘ P) =\( R \ ((((g \text{ more } o \text{ fst}) \rightarrow \neg ((\text{relation-of } P^f_j)) \vdash \neg ((\text{relation-of } P^l_j)) \lor \neg (g \text{ more } o \text{ fst}) \land ((\text{relation-of } P^l_j)) \lor \neg (g \text{ more } o \text{ fst})) \land (\lambda \langle A, A' \rangle. \text{tr } A' = \text{tr } A \land \text{wait } A'))) \in \{p. \text{is-CSP-process } p\}
\]

**Proof**

**Lemma** mono-Guard : mono (Guard g)

**Proof**

**Lemma** false-Guard : false ‘&‘ P = Stop

**Proof**

**Lemma** false-Guard1: \( \forall a b. g (\text{alpha-rp more } a) = \text{False} \Rightarrow (\text{relation-of (g ‘&‘ P)) (a, b) = (relation-of Stop) (a, b})

**Proof**

**Lemma** true-Guard: true ‘&‘ P = P

**Proof**

**Lemma** true-Guard1: \( \forall a b. g (\text{alpha-rp more } a) = \text{True} \Rightarrow (\text{relation-of (g ‘&‘ P)) (a, b) = (relation-of P) (a, b})

**Proof**

**Lemma** Guard-is-state-update-before: g ‘&‘ P = state-update-before (\langle s, s' \rangle . g s) P

**Proof**

**15.16 Prefixed action**

**Definition** do where

\[do \ e \equiv (\lambda (A, A'). \text{tr } A = \text{tr } A' \land (e (\text{more } A)) \notin (\text{ref } A')) \triangleq \text{wait } o \text{snd} \triangleright (\lambda (A, A'). \text{tr } A' = \text{tr } A \triangleq [e (\text{more } A)])\]

**Definition** do-I':(’σ ⇒ ’θ) ⇒ ’θ set ⇒ (’θ, ’σ) relation-rp

**Where** do-I c S \equiv ((\lambda (A, A'). tr A = tr A' \& S \cap (\text{ref } A') = \{\})

\text{\& \triangleq \text{wait } o \text{snd} \triangleright (\lambda (A, A'). \text{hd} (tr A' - tr A) \in S \& (e (\text{more } A) = (\text{last } (tr A')))))

64
definition
iPrefix::(σ ⇒ θ::ev-eq) ⇒ (σ relation) ⇒ ((θ, σ) action ⇒ (θ, σ) action) ⇒ (σ ⇒ θ set) ⇒ (θ, σ) action ⇒ (θ, σ) action where
iPrefix c i j S P ≡ action-of(R(true ⇒ (λ (A, A'), (do-I c (S (more A))) (A, A') & more A' = more A)))); P

definition
oPrefix::(σ ⇒ θ) ⇒ (θ::ev-eq, σ) action ⇒ (θ, σ) action where
oPrefix c P ≡ action-of(R(true ⇒ (do c) ∧ (λ (A, A'), more A' = more A)))); P

definition Prefix0::θ ⇒ (θ::ev-eq, σ) action ⇒ (θ, σ) action where
Prefix0 c P ≡ action-of(R(true ⇒ (do (λ -. c)) ∧ (λ (A, A'), more A' = more A)))); P

definition
read::(v ⇒ θ) ⇒ (v, σ) var-list ⇒ (θ::ev-eq, σ) action ⇒ (θ, σ) action
where read c x P ≡ iPrefix (λ A. c (select x A)) (λ (s, s'), ∃ a. s' = increase x a s) (Let x) (λ -. range c) P

definition
read1::(v ⇒ θ) ⇒ (v, σ) var-list ⇒ (σ ⇒ v set) ⇒ (θ::ev-eq, σ) action ⇒ (θ, σ) action
where read1 c x S P ≡ iPrefix (λ A. c (select x A)) (λ (s, s'), ∃ a. a ∈ (S s) & s' = increase x a s) (Let x) (λ x. c'(S s)) P

definition
write1::(v ⇒ θ) ⇒ (σ ⇒ v) ⇒ (θ::ev-eq, σ) action ⇒ (θ, σ) action
where write1 c a P ≡ oPrefix (λ A. c (a A)) P

definition
write0::θ ⇒ (θ::ev-eq, σ) action ⇒ (θ, σ) action
where write0 c P ≡ Prefix0 c P

definition
read ::[id, pttrn, (θ, σ) action] => (θ, σ) action ((¬?:: → ·))
readS ::[id, pttrn, θ set,(θ, σ) action] => (θ, σ) action ((¬?::·· :· :· → ·))
readSS ::[id, pttrn, σ => θ set,(θ, σ) action] => (θ, σ) action ((¬?::·· ·· :· :· → ·))
write ::[id, σ, (θ, σ) action] => (θ, σ) action ((¬!· ·· → ·))
writeS ::[d, (θ, σ) action] => (θ, σ) action ((¬ ·→ ·))

translations
-read c p P == CONST read c (VAR-LIST p) P
-readS c p b P == CONST read1 c (VAR-LIST p) (λ -. b) P
-readSS c p b P == CONST read1 c (VAR-LIST p) b P
-write c p P == CONST write1 c p P

65
lemma Prefix-is-action:
\((R(\text{true} \vdash (\text{do } c) \land (\lambda (A, A'). \text{more } A' = \text{more } A))) \in \{\text{p. is-CSP-process } p\}\)
\(\langle \text{proof} \rangle\)

lemma Prefix1-is-action:
\((R(\text{true} \vdash \lambda (A, A'). \text{do-I } c (S (\alpha \cdot \text{more } A))) (A, A') \land \alpha \cdot \text{more } A' = \alpha \cdot \text{more } A)) \in \{\text{p. is-CSP-process } p\}\)
\(\langle \text{proof} \rangle\)

lemma Prefix0-is-action:
\((R(\text{true} \vdash (\text{do } c) \land (\lambda (A, A'). \text{more } A' = \text{more } A))) \in \{\text{p. is-CSP-process } p\}\)
\(\langle \text{proof} \rangle\)

lemmas Prefix-is-CSP = Prefix-is-action[simplified]

lemmas Prefix1-is-CSP = Prefix1-is-action[simplified]

lemmas Prefix0-is-CSP = Prefix0-is-action[simplified]

lemma relation-of-iPrefix:
relation-of (iPrefix c i j S P) =
\((\langle(R(\text{true} \vdash (\lambda (A, A'). (\text{do-I } c (S (\text{more } A))) (A, A') \land \text{more } A' = \text{more } A)))\rangle ; \text{relation-of } P)\)
\(\langle \text{proof} \rangle\)

lemma relation-of-oPrefix:
relation-of (oPrefix c P) =
\((\langle(R(\text{true} \vdash (\text{do } c) \land (\lambda (A, A'). \text{more } A' = \text{more } A)))\rangle ; \text{relation-of } P)\)
\(\langle \text{proof} \rangle\)

lemma relation-of- Prefix0:
relation-of (Prefix0 c P) =
\((\langle(R(\text{true} \vdash (\text{do } \lambda - c) \land (\lambda (A, A'). \text{more } A' = \text{more } A)))\rangle ; \text{relation-of } P)\)
\(\langle \text{proof} \rangle\)

lemma mono-iPrefix : mono (iPrefix c i j s)
\(\langle \text{proof} \rangle\)

lemma mono-oPrefix : mono (oPrefix c)
\(\langle \text{proof} \rangle\)

lemma mono-Prefix0 : mono(Prefix0 c)
\(\langle \text{proof} \rangle\)
15.17 Hiding

**definition** 

\[ \text{Hide} : (\emptyset : \text{ev-eq}, '\sigma) \text{ action } \Rightarrow \emptyset \text{ set } \Rightarrow (\emptyset, '\sigma) \text{ action (infixl } \setminus 18) \]

where

\[ P \setminus cs \equiv \text{action-of} (R(\lambda (S, S'). \exists s. (\text{diff-tr } S' S) = (\text{tr-filter} (s - (\text{tr } S)) cs) \& (\text{relation-of } P)(S, S'[\text{tr := } s, \text{ref := } (\text{ref } S') \cup cs]) ; (\text{relation-of Skip})) \]

**definition** 

\[ \text{hid } P cs \equiv (R(\lambda (S, S'). \exists s. (\text{diff-tr } S' S) = (\text{tr-filter} (s - (\text{tr } S)) cs) \& (\text{relation-of } P)(S, S'[\text{tr := } s, \text{ref := } (\text{ref } S') \cup cs]) ; (\text{relation-of Skip})) \]

**lemma** hid-is-R: hid P cs is R healthy

**lemma** hid-Skip: hid P cs = (hid P cs ; ; relation-of Skip)

**lemma** hid-is-CSP1: hid P cs is CSP1 healthy

**lemma** hid-is-CSP2: hid P cs is CSP2 healthy

**lemma** hid-is-CSP: is-CSP-process (hid P cs)

**lemma** Hide-is-action:

\[ (R(\lambda (S, S'). \exists s. (\text{diff-tr } S' S) = (\text{tr-filter} (s - (\text{tr } S)) cs) \& (\text{relation-of } P)(S, S'[\text{tr := } s, \text{ref := } (\text{ref } S') \cup cs]) ; (\text{relation-of Skip})) \in \{ p. \text{is-CSP-process } p \} \]

**lemmas** Hide-is-CSP = Hide-is-action[simplified]

**lemma** relation-of-Hide:

\[ \text{relation-of } (P \setminus cs) = (R(\lambda (S, S'). \exists s. (\text{diff-tr } S' S) = (\text{tr-filter} (s - (\text{tr } S)) cs) \& (\text{relation-of } P)(S, S'[\text{tr := } s, \text{ref := } (\text{ref } S') \cup cs]) ; (\text{relation-of Skip})) \]

**lemma** mono-Hide : mono(\lambda P. P \setminus cs)

15.18 Recursion

To represent the recursion operator "\(\mu\)" over actions, we use the universal least fix-point operator "lfp" defined in the HOL library for lattices. The operator "lfp" is inherited from the "Complete Lattice class" under some
conditions. All theorems defined over this operator can be reused.

In the Circus-Actions theory, we presented the proof that Circus actions form a complete lattice. 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.

\[
\text{syntax } -\text{MU}::[\text{idt, idt} \Rightarrow (\text{'\emptyset, '\sigma} \text{ action})] \Rightarrow (\text{'\emptyset, '\sigma} \text{ action}) \quad (\mu \cdot -)
\]

\[
\text{translations } -\text{MU X P} == \text{CONST lfp} (\lambda X. P)
\]

\langle proof \rangle \langle proof \rangle end

16 Circus syntax

theory Circus-Syntax
imports Denotational-Semantics
begin

abbreviation list-select::[\text{r} \Rightarrow \text{a list}] \Rightarrow (\text{r} \Rightarrow \text{a}) \text{ where}
list-select Sel ≡ hd o Sel

abbreviation list-update::[(\text{a list} \Rightarrow \text{a list}) \Rightarrow \text{r} \Rightarrow \text{r}] \Rightarrow (\text{r} \Rightarrow \text{r} \Rightarrow \text{r}) \text{ where}
list-update Upd ≡ λ e. Upd (λ l. (e (hd l)))\#(tl l))

abbreviation list-update-const::[(\text{a list} \Rightarrow \text{a list}) \Rightarrow \text{r} \Rightarrow \text{r}] \Rightarrow (\text{r} \Rightarrow \text{r} \Rightarrow \text{r}) \text{ where}
list-update-const Upd ≡ λ e. λ (A, A'). A' = Upd (λ l. e\#(tl l)) A

abbreviation update-const::[(\text{a} \Rightarrow \text{a}) \Rightarrow \text{r} \Rightarrow \text{r}] \Rightarrow (\text{r} \Rightarrow \text{r} \Rightarrow \text{r}) \text{ where}
update-const Upd ≡ λ e. λ (A, A'). A' = Upd (λ e. A)

syntax
-synt-assign :: id ⇒ 'a ⇒ 'b relation (\cdot := \cdot)

⟨ML⟩

nonterminal circus-action and circus-schema

syntax
-\text{circus-action} :: 'a =⇒ circus-action (\cdot)
-\text{circus-schema} :: 'a =⇒ circus-schema (\cdot)

⟨ML⟩
17 Refinement and Simulation

theory Refinement
imports Denotational-Semantics Circus-Syntax
begin

17.1 Definitions

In the following, data (state) simulation and functional backwards simulation are defined. The simulation is defined as a function $S$, that corresponds to a state abstraction function.

definition $\text{Simul } S \ b = \text{extend (make (ok } b \text{) (wait } b \text{) (tr } b \text{) (ref } b \text{)) (S (more } b \text{))}$

definition $\text{Simulation ::= (\forall \sigma. \text{ev-eq}, \sigma) action } \Rightarrow \text{ (\forall \sigma. \sigma1) action } \Rightarrow \text{ bool (- \leq -)}$

where $A \leq S B \equiv \forall a b. (\text{relation-of } B \ (a, b) \Rightarrow (\text{relation-of } A \ (\text{Simul } S \ a, \text{Simul } S \ b))$

17.2 Proofs

In order to simplify refinement proofs, some general refinement laws are defined to deal with the refinement of Circus actions at operators level and not at UTP level. Using these laws, and exploiting the advantages of a shallow embedding, the automated proof of refinement becomes surprisingly simple.

lemma $\text{Stop-Sim: Stop } \leq S \text{ Stop}$
(proof)

lemma $\text{Skip-Sim: Skip } \leq S \text{ Skip}$
(proof)

lemma $\text{Chaos-Sim: Chaos } \leq S \text{ Chaos}$
(proof)

lemma $\text{Ndet-Sim:}$

assumes $A: \ P \leq S \ Q \text{ and } B: \ P' \leq S \ Q'$

shows $(P \cap P') \leq S (Q \cap Q')$
(proof)

lemma $\text{Det-Sim:}$

assumes $A: \ P \leq S \ Q \text{ and } B: \ P' \leq S \ Q'$

shows $(P \ △ P') \leq S (Q \ △ Q')$
(proof)
lemma Schema-Sim:
  assumes A: \( \forall a. \text{Pre sc1} (S a) \implies \text{Pre sc2} a \)
  and B: \( \forall a b. \text{Pre sc1} (S a) ; \text{sc2} (a, b) \implies \text{sc1} (S a, S b) \)
  shows \( \text{Schema sc1} \preceq_S \text{Schema sc2} \) (proof)

lemma SUB-Sim:
  assumes A: \( \forall a. \text{Pre sc1} (S a) \implies \text{Pre sc2} a \)
  and B: \( \forall a b. \text{Pre sc1} (S a) ; \text{sc2} (a, b) \implies \text{sc1} (S a, S b) \)
  and C: \( P \preceq_S Q \)
  shows \( \text{state-update-before sc1 P} \preceq_S \text{state-update-before sc2 Q} \) (proof)

lemma Seq-Sim:
  assumes A: \( P \preceq_S Q \)
  and B: \( P' \preceq_S Q' \)
  shows \( (P ; P') \preceq_S (Q ; Q') \) (proof)

lemma Par-Sim:
  assumes A: \( P \preceq_S Q \)
  and B: \( P' \preceq_S Q' \)
  and C: \( \forall a b. S (n s' 2 a b) = n s 2 (S a) (S b) \)
  and D: \( \forall a b. S (n s' 1 a b) = n s 1 (S a) (S b) \)
  shows \( (P [ n s 1 | cs | n s 2 ] P') \preceq_S (Q [ n s' 1 | cs | n s' 2 ] Q') \) (proof)

lemma Assign-Sim:
  assumes A: \( \forall A. \text{vy A} = \text{vx} (S A) \)
  and B: \( \forall A. (S (y-update \text{ff A})) = \text{x-update \text{ff}} (S A) \)
  shows \( (x ::= \text{vx}) \preceq_S (y ::= \text{vy}) \) (proof)

lemma Var-Sim:
  assumes A: \( P \preceq_S Q \)
  and B: \( \forall A. (S ((\text{snd b}) \text{ff A})) = (\text{snd a}) \text{ff} (S A) \)
  shows \( (\text{Var a P}) \preceq_S (\text{Var b Q}) \) (proof)

lemma Guard-Sim:
  assumes A: \( P \preceq_S Q \)
  and B: \( \forall A. \text{h A} = \text{g} (S A) \)
  shows \( (g \ ' \& \ ' P) \preceq_S (h \ ' \& \ ' Q) \) (proof)

lemma Write0-Sim:
  assumes A: \( P \preceq_S Q \)
  shows \( a \rightarrow P \preceq_S a \rightarrow Q \) (proof)

lemma Read-Sim:
assumes $A$: $P \preceq S Q$ and $B$: $\bigwedge A. (d A) = c (S A)$
shows $a’?c \rightarrow P \preceq S a’?d \rightarrow Q$
(proof)

lemma $Read1$-$Sim$:
assumes $A$: $P \preceq S Q$ and $B$: $\bigwedge A. (d A) = c (S A)$
shows $a’?c’\rightarrow^s P \preceq S a’?d’\rightarrow^s Q$
(proof)

lemma $Read1$-$S$-$Sim$:
assumes $A$: $P \preceq S Q$ and $B$: $\bigwedge A. (d A) = c (S A)$ and $C$: $\bigwedge A. (s’ A) = s (S A)$
shows $a’?c’\in^s P \preceq S a’?d’\in^s Q$
(proof)

lemma $Write$-$Sim$:
assumes $A$: $P \preceq S Q$ and $B$: $\bigwedge A. (d A) = c (S A)$
shows $a’!c \rightarrow P \preceq S a’!d \rightarrow Q$
(proof)

lemma $Hide$-$Sim$:
assumes $A$: $P \preceq S Q$
shows $(P \setminus cs) \preceq S (Q \setminus cs)$
(proof)

lemma $lfp$-$Sim$:
assumes $A$: $\bigwedge X. (X \preceq S Q) \implies ((P X) \preceq S Q)$ and $B$: mono $P$
shows $(lfp P) \preceq S Q$
(proof)

lemma $Mu$-$Sim$:
assumes $A$: $\bigwedge X Y. X \preceq S Y \implies (P X) \preceq S (Q Y)$
and $B$: mono $P$ and $C$: mono $Q$
shows $(lfp P) \preceq S (lfp Q)$
(proof)

lemma $bot$-$Sim$:
$bot \preceq S bot$
(proof)

lemma $sim$-$is$-$ref$: $P \sqsubseteq Q = P \preceq (id) Q$
(proof)

lemma $ref$-$eq$: $((P::(a::ev\text{-}eq,b) \text{ action}) = Q) = (P \sqsubseteq Q \& Q \sqsubseteq P)$
(proof)

lemma $rd$-$ref$:
assumes $A{:}R (P \vdash Q) \in \{p.\ is\text{-}CSP\text{-}process\ p\}$
and $B{:}R (P’\vdash Q’) \in \{p.\ is\text{-}CSP\text{-}process\ p\}$
and $C{:}\bigwedge a \ b.\ P (a, b) \Longrightarrow P’ (a, b)$
\[
\text{and } D: \forall a \ b. \ Q'(a, b) \Rightarrow Q(a, b) \\
\text{shows } (\text{action-of } (R (P \vdash Q))) \subseteq (\text{action-of } (R (P' \vdash Q')))
\]

\[\langle \text{proof} \rangle \]

\[\text{lemma } \text{rd-impl}:\]

\[\text{assumes } A: R (P \vdash Q) \in \{p. \text{-CSP-process } p\}\]

\[\text{and } B: R (P' \vdash Q') \in \{p. \text{-CSP-process } p\}\]

\[\text{and } C: \forall a \ b. \ P(a, b) \Rightarrow P'(a, b)\]

\[\text{and } D: \forall a \ b. \ Q'(a, b) \Rightarrow Q(a, b)\]

\[\text{shows } R (P' \vdash Q') (a, b) \rightarrow R (P \vdash Q) (a::('a::ev-eq, 'b) alpha-rp-scheme, b)\]

\[\langle \text{proof} \rangle \]

\[\text{end} \]

18 Concrete example

theory Refinement-Example

imports Refinement

begin

In this section, we present a concrete example of the use of our environment. We define two Circus processes FIG and DFIG, using our syntax. We give the proof of refinement (simulation) of the first process by the second one using the simulation function \(\text{Sim}\).

18.1 Process definitions

circus-process FIG =

\[
\begin{align*}
\text{alphabet} & = [v::nat, x::nat] \\
\text{state} & = [idS::nat set] \\
\text{channel} & = [\text{out nat}, \text{req}, \text{ret nat}] \\
\text{schema Init} & = \text{idS}' = \{\} \\
\text{schema Out} & = \exists a. \ v' = a \land a \notin \text{idS} \land \text{idS}' = \text{idS} \cup \{v'\} \\
\text{schema Remove} & = x \in \text{idS} \land \text{idS}' = \text{idS} - \{x\} \\
\text{where} \quad \text{var } v \bullet (\text{Schema FIG.Init}^*; \\
\quad \quad \quad \quad \mu X \bullet (((\text{req } \rightarrow (\text{Schema FIG.Out})^*; \ ' \text{out}'! (\text{hd o v}) \rightarrow \text{Skip})) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{□ } (\text{ret}'? x \rightarrow (\text{Schema FIG.Remove}))^*; \ ' X))
\end{align*}
\]

circus-process DFIG =

\[
\begin{align*}
\text{alphabet} & = [v::nat, x::nat] \\
\text{state} & = [\text{retidS::nat set}, \text{max::nat}] \\
\text{channel} & = \text{FIG-channels} \\
\text{schema Init} & = \text{retidS}' = \{\} \land \text{max}' = 0 \\
\text{schema Out} & = v' = \text{max} \land \text{max}' = (\text{max} + 1) \land \text{retidS}' = \text{retidS} - \{v'\} \\
\text{schema Remove} & = x < \text{max} \land \text{retidS}' = \text{retidS} \cup \{x\} \land \text{max}' = \text{max} \\
\text{where} \quad \text{var } v \bullet (\text{Schema DFIG.Init}^*; \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{□ } (\text{ret}'? x \rightarrow (\text{Schema DFIG.Remove}))^*; \ ' X)
\end{align*}
\]
\[ \mu X \cdot (((\text{reg} \to (\text{Schema DFIG}.Out))\,;\, \text{out}!\cdot (\text{hd} \circ o \circ v) \to \text{Skip})) \]
\[ \Box (\text{ret}'\cdot x \to (\text{Schema DFIG}.\text{Remove}))\,;\, x \] \]

**definition Sim** where
\[ Sim \ A = \text{FIG-alphabet.make (DFIG-alphabet.v A) (DFIG-alphabet.x A)} \]
\[ \{ a. a < (DFIG-alphabet.max A) \land a \notin (DFIG-alphabet.retids A) \} \]

### 18.2 Simulation proofs

For the simulation proof, we give first proofs for simulation over the schema expressions. The proof is then given over the main actions of the processes.

**lemma SimInit**: \((\text{Schema FIG}.\text{Init}) \preceq Sim (\text{Schema DFIG}.\text{Init})\)

(proof)

**lemma SimOut**: \((\text{Schema FIG}.\text{Out}) \preceq Sim (\text{Schema DFIG}.\text{Out})\)

(proof)

**lemma SimRemove**: \((\text{Schema FIG}.\text{Remove}) \preceq Sim (\text{Schema DFIG}.\text{Remove})\)

(proof)

**lemma FIG.FIG \preceq Sim DFIG.DFIG**

(proof)

### References


