Isabelle/Circus

Abderrahmane Feliachi, Marie-Claude Gaudel, Makarius Wenzel
and Burkhart Wolff

September 19, 2015

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 lan-
guage 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.

Contents

1 Introduction ........................................ 5

2 Background ........................................ 6

2.1 Isabelle, HOL and Isabelle/HOL .............. 6
2.1.1 isar ............................................. 6
2.1.2 Higher-order logic (HOL) ..................... 6
2.1.3 Isabelle/HOL .................................... 6

2.2 Advanced Specification Constructs in Isabelle/HOL .......... 6

2.2.1 Constant definitions. .......................... 6
2.2.2 Type definitions. ............................... 7
2.2.3 Extensible records. ......................... 7

2.3 Circus and its UTP Foundation .................. 8

2.3.1 Predicates and Relations. .................... 8
2.3.2 Designs and processes. ...................... 9
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 UTP variables ................................................. 24

8 Predicates and relations ..................................... 24
  8.1 Definitions ............................................ 24
  8.2 Proofs ................................................. 26
      8.2.1 Setup of automated tools ......................... 26
      8.2.2 Misc lemmas .................................... 28

9 Designs ....................................................... 33
  9.1 Definitions ............................................ 34
  9.2 Proofs .................................................. 35

10 Reactive processes ......................................... 37
  10.1 Preliminaries ......................................... 38
  10.2 Definitions ........................................... 40
  10.3 Proofs ................................................. 41

11 CSP processes .............................................. 44
  11.1 Definitions ........................................... 44
  11.2 Proofs ................................................. 45
  11.3 CSP processes and reactive designs ...................... 49
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 $\alpha ::$ order, the type-variable ranges over all types that posse 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 $=_\alpha$ as well as the object-logical quantifiers. 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:

\textbf{definition} upd::\((\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow (\alpha \rightarrow \beta)\) 
\((\_ \_ := \_\_ )\)

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

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

\subsection{Type definitions.}

Types can be introduced in Isabelle/HOL in different ways. The most general way to safely introduce new types is using the \texttt{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:

\texttt{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:

\texttt{by (rule_tac x = 1 in exI, simp)}

where \texttt{rule_tac} is a tactic that applies an introduction rule, and \texttt{exI} corresponds to the introduction of the existential quantification.

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

\texttt{datatype \(\alpha\) option = None | Some \(\alpha\)}

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

\texttt{case } x \texttt{ of None } \Rightarrow \ldots \texttt{ | Some } a \Rightarrow \ldots 

\subsection{Extensible records.}

Isabelle/HOL’s support for \texttt{extensible records} is of particular importance for our work. Record types are denoted, for example, by:

\texttt{record } T = a::T_1 \texttt{ \\
 b::T_2}

which implicitly introduces the record constructor \(\langle a=e_1, b=e_2 \rangle\) 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
\(\delta\), 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:

\[
\text{record } ET = T + c::T_3
\]

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 \(P \sqsupset P'\) and \(P \sqcap P'\)), the concept of concealment (hiding) \(P \setminus A\) 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: \(P \| c \| n \quad n' \| P'\) means that \(P\) and \(P'\) communicate via the channels mentioned in \(c\); moreover, \(P\) may modify the variables mentioned in \(n\) only, and \(P'\) in \(n'\) only, \(n\) and \(n'\) are disjoint name sets.

Moreover, the language comes with a formal notion of refinement based on a denotational semantics. It follows the failure/divergence semantics [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. \(\langle \{x, y\}, x > y \rangle\). 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
channel req
channel ret, out : ID

process FIG ≜ begin
state S == [ idS : P ID ]
Init ≜ idS := ∅

Out
ΔS
v! : ID

v! \notin idS
idS′ = idS \cup \{ v! \}

Remove
ΔS
x? : ID

idS′ = idS \setminus \{ x? \}

• Init ; var v : ID •
(μ X • (req → Out ; out!v → Skip □ ref?x → 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′) . 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 ok. It is used to record the start and termination of a program. A UTP design is defined as follows in Isabelle:

\[(P \vdash Q) \equiv \lambda (A, A′). (ok A \land P (A, A′)) \rightarrow (ok A′ \land Q (A, A′))\]

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: wait, tr and ref. The boolean variable wait records if the process is waiting for an interaction or has terminated. tr records the list (trace) of interactions the process has performed so far. The variable ref contains the set
of interactions (events) the process may refuse to perform. These observational variables define 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 \( \texttt{var var @ Action} \). The scope of local variables is restricted to \( \texttt{Action} \).

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 \( \text{idS} \) 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 \( \alpha \) alpha that contains the observational variables of a reactive process, plus the variable \( \text{idS} \). 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 \( \text{name} \mapsto \text{value} \), 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 up-
dates the value of this variable. Since we may have different HOL types for
different variables, a unique definition for select and update cannot be pro-
vided. 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 \( \alpha \) and \( x\_update \).
Since a variable is characterized essentially by these functions, we define a
general type (synonym) called \( \text{var} \) which represents a variable as a pair of
its select and update function (in the underlying state \( \sigma \)).

For a given alphabet (record) of type \( \sigma \), (\( \beta \), the type \( \sigma \)var represents
the type of the variables whose value type is \( \beta \). One can then extract
the select and update functions from a given variable with the following
functions:

\\[
\text{definition select} :: (\( \beta \), \( \sigma \)) \text{var} \Rightarrow \sigma \Rightarrow \beta \\
\text{where select f } \equiv (\text{fst } f) \\
\]

\\[
\text{definition update} :: (\( \beta \), \( \sigma \)) \text{var} \Rightarrow \beta \Rightarrow \sigma \Rightarrow \sigma \\
\text{where update f v } \equiv (\text{snd } f) (\lambda _ . v) \\
\]

Finally, we introduce a function called \( \text{VAR} \) to implement a syntactic
translation of a variable name to an entity of type \( \text{var} \).

\text{syntax } "\_VAR" :: "id \Rightarrow (\beta , \sigma) \text{var}" ("VAR _")
\text{translations} \text{VAR x } \Rightarrow (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 \_update by a default mechanism. On this basis,
the assignment notation can be written as usual:

\text{syntax } "\_assign" :: "id \Rightarrow (\beta \Rightarrow \sigma) \Rightarrow (\alpha, \sigma) \text{action}" ("_ i:= _")
\text{translations}
"x i:= E" => "CONST ASSIGN (VAR x) E"

and mapped to the semantics of the program variable \((x,x\_update)\) to-
gether 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 \equiv x_update (x lb) gb
```

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

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
\textbf{action}:

\begin{verbatim}
typedef(Action) 
(\alpha::chan_eq,\sigma) action = {p:(\alpha,\sigma)relation_rp. is_CSP_process p}
proof - {...}
qed
\end{verbatim}

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 $((\alpha,\sigma) relation\_rp \Rightarrow (\alpha,\sigma) action$ and $(\alpha,\sigma)action \Rightarrow (\alpha,\sigma)relation\_rp$ as well as the usual two axioms expressing the bijection $Abs\_Action(Rep\_Action(X))=X$ and $is\_CSP\_process p \Longrightarrow Rep\_Action(Abs\_Action(p))=p$ where $is\_CSP\_process$ captures the
healthiness conditions.

Every \textit{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.

\subsection{Basic actions.}

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

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

\textbf{Skip} is defined as a reactive design, with a precondition \textit{true} and a
postcondition stating that the system terminates and all the state variables
are not changed. We represent this fact by stating that the \textit{more} field (seen
in Section 2.2) is not changed, since this field is mapped to all the state
variables. Note that using the \textit{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).

\begin{verbatim}
definition Skip \equiv Abs\_Action (R (true \vdash \lambda (A, A'). tr A' = tr A
\land \neg wait A' \land more A = more A'))
\end{verbatim}

\subsection{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 \textit{Circus} processes is straightforward:

\textbf{definition}

\begin{verbatim}
ASSIGN::"(\beta, \sigma) \triangleright \Rightarrow (\sigma \Rightarrow \beta) \Rightarrow (\alpha::ev\_eq, \sigma)\ action"
\end{verbatim}

where

\begin{verbatim}
ASSIGN x e \equiv Abs\_Action (R (true \vdash Y))
\end{verbatim}

where

\begin{verbatim}
Y = \lambda (A, A'). \tr A' = \tr A \land \neg \wait A' \land
  more A' = (assign x (e (more A))) (more A)
\end{verbatim}

where \texttt{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 \texttt{do\_I}. In the \textit{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.

\textbf{definition} \texttt{do\_I c x P} \equiv X \triangleleft \wait o \fst \triangleright Y

where

\begin{verbatim}
X = (\lambda (A, A'). \tr A = \tr A' \land ((c \triangleright P) \cap \ref A') = {})
\end{verbatim}

and

\begin{verbatim}
Y = (\lambda (A, A'). \hd ((\tr A') - (\tr A)) \in (c \triangleright P) \land
  (c (select x (more A))) = (last (\tr A')))
\end{verbatim}

where \texttt{c} is a channel constructor, \texttt{x} is a variable (of \texttt{var} type) and \texttt{P} is a predicate. The \texttt{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:

\textbf{definition} \texttt{Prefix c x P S} \equiv \texttt{Abs\_Action}(R (true \vdash Y)) \ ; \ S

where

\begin{verbatim}
Y = \texttt{do\_I c x P} \land (\lambda (A, A'). more A' = more A)
\end{verbatim}

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

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

where `read`, `write1` and `write0` 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:

```plaintext
translations
c ! p → P == CONST write1 c p P
c ? p : b → P == CONST Prefix c (VAR p) b P
c ? p → P == CONST read c (VAR p) P
da → 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:

```plaintext
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:

```plaintext
fun tr_filter::"a::chan_eq list ⇒ a set ⇒ a list" where
  tr_filter [] cs = []
| tr_filter (x#xs) cs = (if (∼ chan_in_set x cs)
  then (x#(tr_filter xs cs))
  else (tr_filter xs cs))
```

17
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 \textit{where}. Below, we illustrate how to use the environment to write a \textit{Circus} specification using the FIG process example presented in Figure 1.

\begin{verbatim}
circusprocess FIG =
alphabet = [v::nat, x::nat]
state = [idS::nat set]
channel = [req, ret nat, out nat]
schema Init = idS := {}
schema Out = \exists a. v' = a \land v' \notin idS \land idS' = idS \cup \{v'\}
schema Remove = x \notin idS \land idS' = idS - \{x\}
where var v \cdot Schema Init; (\mu X \cdot (req \rightarrow Schema Out; out!v \rightarrow Skip)
\quad \Box (ret?x \rightarrow Schema Remove); X)
\end{verbatim}

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).

- \textit{alphabet} generates a list of record fields to represent the binding. These fields map names to value lists.

- \textit{state} generates a list of record fields that corresponds to the state variables. The names are mapped to single values. This command, together with \textit{alphabet} command, generates a record that represents all the variables (for the FIG example the command generates the record \texttt{FIG\_alphabet}, that contains the fields \texttt{v} and \texttt{x} of type \texttt{nat list} and the field \texttt{idS} of type \texttt{nat set}).

- \textit{channel} introduces a datatype of typed communication channels (for the FIG example the command generates the datatype \texttt{FIG\_channels} that contains the constructors \texttt{req} without communicated value and \texttt{ret} and \texttt{out} that communicate natural values).

- \textit{schema} allows the definition of schema expressions represented as an alphabetized relation over the process variables (in the example the schema expressions \texttt{FIG.Init}, \texttt{FIG.Out} and \texttt{FIG.Remove} are generated).

- \textit{action} introduces definitions for \textit{Circus} actions in the process. These definitions are based on the denotational semantics of \textit{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:

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

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

This definition assumes that the actions `A1` and `A2` share the same alphabet (binding) and the same channels. In general, refinement involves an important data evolution and growth. The data refinement is defined in [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:

```isar
definition Simulation ("_ ⪯ _") where
  A1 ⪯R A2 = ∀ a b. (Rep_Action A2)(a,b) −→ (Rep_Action A1)(R a,R b)
```

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

4.3 Refinement Proofs

We can use the definition of simulation to transform the proof of refinement to a simple proof of implication by unfolding the operators in terms of their underlying relational semantics. The problem with this approach is that the size of proofs will grow exponentially with 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 theses laws in our environment (see Table 1).
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 \( \text{idS} \) is refined into a set of removed identifiers \( \text{retidS} \) and a number \( \text{max} \), which is the rank of the last issued identifier.

```circus
process DFIG =
  alphabet = [w::nat, y::nat]
  state = [retidS::nat set, max::nat]
  schema Init = retidS' = {} \& max' = 0
  schema Out = w' = max \& max' = max+1 \& retidS' = retidS - {max}
  schema Remove = y < max \& y \notin \text{retidS} \& \text{retidS'} = \text{retidS} \cup \{y\}
    \& max' = max
where var w \cdot Schema Init; (\mu X \cdot (req \rightarrow Schema Out; out!w \rightarrow Skip))
  □ (ret?y \rightarrow Schema Remove); X)
```

Table 1: Proved refinement laws
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:

\[
\text{definition } \text{Sim} A = \text{FIG_alphabet.make} \ (w \ A) \ (y \ A) \\
\qquad \quad \quad \quad \quad \{ \{a. \ a < (\max A) \land a \notin (\text{retidS} A) \}\}
\]

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

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:

\[
\text{theorem } "\text{FIG.FIG} \preceq \text{Sim} \ DFIG.DFIG"
\]
\[
\quad \text{apply } (\text{auto simp: DFIG.DFIG_def FIG.FIG_def mono_Seq} \\
\qquad \quad \quad \text{intro!: VarI SeqI MuI DetI SyncI InpI OutI SkipI})
\]
\[
\quad \text{apply } (\text{simp_all add: SimInit SimOut SimRemove SimDef})
\]
\[
\quad \text{done}
\]

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

5 Conclusions

We have shown for the language Circus, which combines data-oriented modeling in the style of Z and behavioral modeling in the style of CSP, a semantics in form of a shallow embedding in Isabelle/HOL. In particular, by representing the somewhat non-standard concept of the alphabet in UTP in form of extensible records in HOL, we achieved a fairly compact, typed presentation of the language. In contrast to previous work based on some deep embedding [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 — \textit{i.e.} satisfy altogether \textit{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 UTP variables

thory Var
imports Main
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

8 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.

8.1 Definitions

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

type-synonym 'a alphabet = 'a
type-synonym 'a predicate = 'a alphabet ⇒ bool

definition true::'a predicate
where true ≡ λA. True

definition false::'a predicate
where false ≡ λA. False

definition not::'a predicate ⇒ 'a predicate
where not ≡ λP. ¬P

definition conj::'a predicate ⇒ 'a predicate ⇒ 'a predicate
where P ∧ Q ≡ λA. P A ∧ Q A

definition disj::'a predicate ⇒ 'a predicate ⇒ 'a predicate
where P ∨ Q ≡ λA. P A ∨ Q A

definition impl::'a predicate ⇒ 'a predicate ⇒ 'a predicate
where P −→ Q ≡ λA. P A −→ Q A

definition iff::'a predicate ⇒ 'a predicate ⇒ 'a predicate
where P ⇔ Q ≡ λA. P A ⇔ Q A

definition ex::['β ⇒ 'a predicate] ⇒ 'a predicate (binder ∃ 10)
where ∃ x. P x ≡ λA. ∃ x. (P x) A

definition all::['β ⇒ 'a predicate] ⇒ 'a predicate (binder ∀ 10)
where ∀ x. P x ≡ λA. ∀ x. (P x) A

type-synonym 'a condition = ('a × 'a) ⇒ bool

type-synonym 'a relation = ('a × 'a) ⇒ bool

definition cond::'a relation ⇒ 'a condition ⇒ 'a relation ⇒ 'a relation
where (P and b > Q) ≡ (b ∧ P) ∨ (¬b ∧ Q)

definition comp::(('a × 'β) ⇒ bool) ⇒ ( ('β × 'γ) ⇒ bool) ⇒ ('a × 'γ) ⇒ bool
where P ; ; Q ≡ λr. r : ({p. P p} O {q. Q q})

definition Assign::('a, 'b) var ⇒ 'a ⇒ 'b relation
where Assign x a ≡ λ(A, A'). A' = (assign x a) A

syntax
-assignment :: id ⇒ 'a ⇒ 'b relation (· ::= ·)

translations
y ::= vv => CONST Assign (VAR y) vv

abbreviation (input) closure::'a predicate ⇒ bool ([-])
where \[ P \equiv \forall A. P A \]

abbreviation (input) ndet::\'a relation \Rightarrow \'a relation ((\cap -))
where \[ P \cap Q \equiv P \lor Q \]

abbreviation (input) join::\'a relation \Rightarrow \'a relation ((\cup -))
where \[ P \cap Q \equiv P \land Q \]

abbreviation (input) ndetS::\'a relation set \Rightarrow \'a relation ((\cap -))
where \[ \bigcap S \equiv \lambda A. A \in \bigcup \{ \{ p. P p \} | P. P \in S \} \]

abbreviation (input) conjS::\'a relation set \Rightarrow \'a relation ((\cup -))
where \[ \bigcup S \equiv \lambda A. A \in \bigcap \{ \{ p. P p \} | P. P \in S \} \]

abbreviation (input) skip-r::\'a relation
where \[ \Pi r \equiv \lambda (A, A'). A = A' \]

abbreviation (input) Bot::\'a relation
where \[ Bot \equiv \text{true} \]

abbreviation (input) Top::\'a relation
where \[ Top \equiv \text{false} \]

lemmas utp-defs = true-def false-def conj-def disj-def not-def impl-def iff-def ex-def all-def cond-def comp-def Assign-def

8.2 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.

8.2.1 Setup of automated tools

lemma true-intro: \( \text{true} \ x \ \langle \text{proof} \rangle \)
lemma false-elim: \( \text{false} \ x \ \Rightarrow C \ \langle \text{proof} \rangle \)
lemma true-elim: \( \text{true} \ x \ \Rightarrow C \ \Rightarrow C \ \langle \text{proof} \rangle \)

lemma not-intro: \( P \ x \ \Rightarrow \text{false} \ x \ \Rightarrow (\neg P) \ x \ \langle \text{proof} \rangle \)
lemma not-elim: \( (\neg P) \ x \ \Rightarrow P \ x \ \Rightarrow C \ \langle \text{proof} \rangle \)
lemma not-dest: \( (\neg P) \ x \ \Rightarrow \neg P \ x \ \langle \text{proof} \rangle \)

lemma conj-intro: \( P \ x \ \Rightarrow Q \ x \ \Rightarrow (P \land Q) \ x \ \langle \text{proof} \rangle \)
lemma conj-elim: \( (P \land Q) \ x \ \Rightarrow (P \Rightarrow Q \ x \Rightarrow C) \Rightarrow C \ \langle \text{proof} \rangle \)

lemma disj-introC: \( (\neg Q \ x \ \Rightarrow P \ x) \ \Rightarrow (P \lor Q) \ x \ \langle \text{proof} \rangle \)
lemma disj-elim: \( (P \lor Q) \ x \ \Rightarrow (P \Rightarrow C) \Rightarrow (Q \ x \Rightarrow C) \Rightarrow C \ \langle \text{proof} \rangle \)
lemma impl-intro: \((P \implies Q) \implies (P \implies Q)\) (proof)

lemma impl-elimC: \((P \implies Q) \implies (\neg P \implies R) \implies (Q \implies R) \implies R\) (proof)

lemma iff-intro: \((P \implies Q) \implies (Q \implies P) \implies (P \iff Q)\) (proof)

lemma iff-elimC: \((P \iff Q) \implies (P \implies Q \implies R) \implies (Q \implies \neg Q \implies R) \implies R\) (proof)

lemma all-intro: \((\forall a. P a) \implies (\forall a. P a)\) (proof)

lemma all-elim: \((\forall a. P a) \implies (P a \implies R) \implies R\) (proof)

lemma ex-intro: \((\exists a. P a) \implies (P a \implies R) \implies R\) (proof)

lemma ex-elim: \((\exists a. P a) \implies (\exists a. P a \implies Q) \implies Q\) (proof)

lemma comp-intro: \(P (a, b) \implies Q (b, c) \implies (P ; ; Q) (a, c)\) (proof)

lemma comp-elim: \((P ; ; Q) ac \implies (\exists a b c. ac = (a, c) \implies P (a, b) \implies Q (b, c) \implies C) \implies C\) (proof)

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!]
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

27
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

8.2.2 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)

lemma cond-L7: (P ▷ b ▷ (P ▷ c ▷ Q)) = (P ▷ b ▷ c ▷ Q)
(proof)

lemma cond-and-distr: ((P ∧ Q) ▷ b ▷ (R ∧ S)) = ((P ▷ b ▷ R) ∧ (Q ▷ b ▷ S))
(proof)

lemma cond-or-distr: ((P ∨ Q) ▷ b ▷ (R ∨ S)) = ((P ▷ b ▷ R) ∨ (Q ▷ b ▷ S))
(proof)

lemma cond-imp-distr:
((P → Q) ▷ b ▷ (R → S)) = ((P ▷ b ▷ R) → (Q ▷ b ▷ S))
(proof)

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

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

**Lemma** conj-comp:
\((\bigwedge a b c. P (a, b) = P (a, c)) \implies (P \land (Q ;; R)) = ((P \land Q) ;; R)\)

**Lemma** comp-cond-left-distr:
\[\begin{align*}
\text{assumes} & \bigwedge x y z. b (x, y) = b (x, z) \\
\text{shows} & (P \triangleleft b \triangleright (Q \land R)) = ((P \triangleleft b \triangleright Q) \land (P \triangleleft b \triangleright R))
\end{align*}\]

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

**Lemma** ndet-assoc: \(P \cap (Q \cap R) = (P \land Q) \cap R\)

**Lemma** ndet-idemp: \(P \cap P = P\)

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

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

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

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

**Lemma** comp-ndet-r-distr: \((P ;; (Q \cap R)) = ((P ;; Q) \cap (P ;; R))\)

**Lemma** l2-5-1-A: \(\forall X \in S. [X \rightarrow (\bigcap S)]\)

**Lemma** l2-5-1-B: \(\forall X \in S. [X \rightarrow P] \implies [\bigcap S] \rightarrow P\)

**Lemma** l2-5-1: \([\bigcap S] \rightarrow P] \leftrightarrow (\forall X \in S. [X \rightarrow P])\)

**Lemma** empty-disj: \(\bigcap \{\} = \text{Top}\)
lemma l2-5-1-2: \[ P \rightarrow (\bigcup S) \iff (\forall X \in S. [P \rightarrow X]) \]

lemma empty-conj: \[ \bigcup \{\} = \text{Bot} \]

lemma l2-5-2: \[(\bigcup S) \cap Q = (\bigcup \{P \cap Q \mid P \in S\}) \]

lemma l2-5-3: \[(\bigcap S) \cup Q = (\bigcap \{P \cup Q \mid P \in S\}) \]

lemma l2-5-4: \[(\bigcap S) ; Q = (\bigcap \{P ; Q \mid P \in S\}) \]

lemma l2-5-5: \[(\bigcap S) \triangleleft (\bigcap S) = (\bigcap \{P \triangleleft Q \mid P \in S\}) \]

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

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

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

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

lemma not-cond: \[ (\neg(P \triangleleft b \triangleleft Q)) = ((\neg P) \triangleleft b \triangleleft (\neg Q)) \]

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

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

lemma cond-imp-dist: \[ ((P \triangleleft b \triangleright Q) \rightarrow R) = ((P \rightarrow R) \triangleleft b \triangleright (Q \rightarrow R)) \]

lemma cond-conj-distr: \[ ((P \triangleleft b \triangleright Q) \land R) = ((P \land R) \triangleleft b \triangleright (Q \land R)) \]

lemma cond-disj-distr: \[ ((P \triangleleft b \triangleright Q) \lor R) = ((P \lor R) \triangleleft b \triangleright (Q \lor R)) \]
lemma cond-know-b: \((b \land (P \land b \lor Q)) = (b \land P)\)
(proof)

lemma cond-know-nb: \(((\neg (b)) \land (P \land b \lor Q)) = ((\neg (b)) \land Q)\)
(proof)

lemma cond-ass-if: \((P \land b \lor Q) = ((b) \land (P \land b \lor Q))\)
(proof)

lemma cond-ass-else: \((P \land b \lor Q) = (P \land ((\neg b) \land Q))\)
(proof)

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

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

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

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

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

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

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

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

lemma conj-assoc: \(((P:\alpha \text{ predicate}) \land Q) \land S) = (P \land (Q \land S))\)
(proof)

lemma disj-assoc: \(((P:\alpha \text{ predicate}) \lor Q) \lor S) = (P \lor (Q \lor S))\)
(proof)

lemma conj-disj-abs: \((P:\alpha \text{ predicate}) \land (P \lor Q)) = P\)
(proof)

lemma disj-conj-abs: \((P:\alpha \text{ predicate}) \lor (P \land Q)) = P\)
(proof)
lemma conj-disj-distr: \((P::\alpha \text{ predicate}) \land (Q \lor R) = ((P \land Q) \lor (P \land R))\)
(proof)

lemma disj-conj-distr: \((P::\alpha \text{ predicate}) \lor (Q \land R) = ((P \lor Q) \land (P \lor R))\)
(proof)

lemma true-conj-id: \((P \land \text{true}) = P\)
(proof)

lemma true-disj-zero: \((P \lor \text{true}) = \text{true}\)
(proof)

lemma true-conj-zero: \((P \land \text{false}) = \text{false}\)
(proof)

lemma true-disj-id: \((P \lor \text{false}) = P\)
(proof)

lemma imp-vacuous: \((\text{false} \rightarrow u) = \text{true}\)
(proof)

lemma p-and-not-p: \((P \land \neg P) = \text{false}\)
(proof)

lemma conj-disj-not-abs: \(((P::\alpha \text{ predicate}) \land ((\neg P) \lor Q)) = (P \land Q)\)
(proof)

lemma p-or-not-p: \((P \lor \neg P) = \text{true}\)
(proof)

lemma double-negation: \((\neg (\neg (P::\alpha \text{ predicate}))) = P\)
(proof)

lemma not-conj-deMorgans: \((\neg ((P::\alpha \text{ predicate}) \land Q)) = ((\neg P) \lor (\neg Q))\)
(proof)

lemma not-disj-deMorgans: \((\neg ((P::\alpha \text{ predicate}) \lor Q)) = ((\neg P) \land (\neg Q))\)
(proof)

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

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

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

lemma p-equiv-p: \((P \leftrightarrow P) = \text{true}\)
lemma equiv-eq: (((P::α predicate) ∧ Q) ∨ (¬P ∧ ¬Q)) = true) ↔ (P = Q)
(proof)

lemma equiv-eq1: (((P::α predicate) ↔ Q) = true) ↔ (P = Q)
(proof)

lemma cond-subst: b = c ⟹ (P ≺ b △ Q) = (P ≺ c △ Q)
(proof)

lemma ex-disj-distr: ((∃ x. P x) ∨ (∃ x. Q x)) = (∃ x. (P x ∨ Q x))
(proof)

lemma all-disj-distr: ((∀ x. P x) ∨ (∀ x. Q)) = (∀ x. (P x ∨ Q))
(proof)

lemma all-conj-distr: ((∀ x. P x) ∧ (∀ x. Q)) = (∀ x. (P x ∧ Q))
(proof)

lemma all-triv: (∀ x. P) = P
(proof)

lemma closure-true: [true]
(proof)

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

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

lemma closure-conj-distr: ([P] ∧ [Q]) = [P ∧ Q]
(proof)

lemma closure-imp-distr: [P → Q] → [P] → [Q]
(proof)

lemma true-iff[simp]: (P ↔ true) = P
(proof)

lemma true-imp[simp]: (true → P) = P
(proof)

end

9 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.

### 9.1 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 $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$ and $\mathcal{H}_4$.

**record** alpha-d = ok::bool

**type-synonym** $\alpha$ alphabet-d = $\alpha$ alphabet-d-scheme alphabet

**type-synonym** $\alpha$ relation-d = $\alpha$ alphabet-d relation

**definition** design ::= $\alpha$ relation-d $\Rightarrow$ $\alpha$ relation-d $\Rightarrow$ $\alpha$ relation-d (\(\frac{}{\cdot}\))

**where** $\mathcal{P}$ $\vdash$ $\mathcal{Q}$ $\equiv$ $\lambda$ (A, A') . (ok A $\wedge$ $\mathcal{P}$ (A, A')) $\rightarrow$ (ok A' $\wedge$ $\mathcal{Q}$ (A, A'))

**definition** skip-d ::= $\alpha$ relation-d ($\Pi d$)

**where** $\Pi d$ $\equiv$ (true $\vdash$ $\Pi r$)

**definition** $\textit{J}$

**where** $\textit{J}$ $\equiv$ $\lambda$ (A, A') . (ok A $\rightarrow$ ok A') $\wedge$ more A = more A'

**type-synonym** $\alpha$ Healthiness-condition = $\alpha$ relation $\Rightarrow$ $\alpha$ relation

**definition** Healthy $\equiv$ $\alpha$ relation $\Rightarrow$ $\alpha$ Healthiness-condition $\Rightarrow$ bool (- is - healthy)

**where** $\mathcal{P}$ is $\textit{H}$ healthy $\equiv$ ($\mathcal{P}$ = $\textit{H}$ $\mathcal{P}$)

**lemma** Healthy-def $\vdash$: P is $\textit{H}$ healthy = ($\mathcal{H}$ $\mathcal{P}$ = $\mathcal{P}$) (proof)

**definition** $\mathcal{H}_1$::($\alpha$ alphabet-d) Healthiness-condition

**where** $\mathcal{H}_1$ (P) $\equiv$ (ok o fst $\rightarrow$ P)

**definition** $\mathcal{H}_2$::($\alpha$ alphabet-d) Healthiness-condition

**where** $\mathcal{H}_2$ (P) $\equiv$ P $\mathcal{;}$ $\textit{J}$

**definition** $\mathcal{H}_3$::($\alpha$ alphabet-d) Healthiness-condition

**where** $\mathcal{H}_3$ (P) $\equiv$ P $\mathcal{;}$ $\Pi d$

**definition** $\mathcal{H}_4$::($\alpha$ alphabet-d) Healthiness-condition

**where** $\mathcal{H}_4$ (P) $\equiv$ (($\mathcal{P}$ $\mathcal{;}$ true) $\leftrightarrow$ true)


**definition** \( \sigma_f :: \alpha \text{ relation-d} \Rightarrow \text{ '\alpha relation-d} \\
\text{where } \sigma_f D \equiv \lambda (A, A'). D \ A', \ A'(\text{ok:=False}) \)

**definition** \( \sigma_t :: \alpha \text{ relation-d} \Rightarrow \text{ '\alpha relation-d} \\
\text{where } \sigma_t D \equiv \lambda (A, A'). D \ A', \ A'(\text{ok:=True}) \)

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

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

**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

### 9.2 Proofs

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

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

**lemma** pi-comp-left-unit: \( (\Pi d; (P \vdash Q)) = (P \vdash Q) \\
\text{⟨proof⟩} \)

**theorem** t3-1-4-2: \( ((P1 \vdash Q1) \triangleleft b \triangleright (P2 \vdash Q2)) = ((P1 \triangleleft b \triangleright P2) \vdash (Q1 \triangleleft b \triangleright Q2)) \\
\text{⟨proof⟩} \)

**lemma** conv-conj-distr: \( \sigma_t (P \land Q) = (\sigma_t P \land \sigma_t Q) \\
\text{⟨proof⟩} \)

**lemma** conv-disj-distr: \( \sigma_t (P \lor Q) = (\sigma_t P \lor \sigma_t Q) \\
\text{⟨proof⟩} \)

**lemma** conv-imp-distr: \( \sigma_t (P \to Q) = ((\sigma_t P) \to \sigma_t Q) \\
\text{⟨proof⟩} \)

**lemma** conv-not-distr: \( \sigma_t (\neg P) = (\neg (\sigma_t P)) \\
\text{⟨proof⟩} \)

**lemma** div-conj-distr: \( \sigma_f (P \land Q) = (\sigma f P \land \sigma f Q) \\
\text{⟨proof⟩} \)

**lemma** div-disj-distr: \( \sigma f (P \lor Q) = (\sigma f P \lor \sigma f Q) \\
\text{⟨proof⟩} \)

**lemma** div-imp-distr: \( \sigma f (P \to Q) = ((\sigma f P) \to \sigma f Q) \\
\text{⟨proof⟩} \)
lemma div-not-distr: \( \sigma f (\neg P) = (\neg (\sigma f P)) \)  
\( \langle \text{proof} \rangle \)

lemma ok-conv: \( \sigma t \text{ OKAY} = \text{OKAY} \)  
\( \langle \text{proof} \rangle \)

lemma ok-div: \( \sigma f \text{ OKAY} = \text{OKAY} \)  
\( \langle \text{proof} \rangle \)

lemma ok'-conv: \( \sigma t \text{ OKAY}' = \text{true} \)  
\( \langle \text{proof} \rangle \)

lemma ok'-div: \( \sigma f \text{ OKAY}' = \text{false} \)  
\( \langle \text{proof} \rangle \)

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

lemma H2-J-2-a: \( P (a,b) \rightarrow (P ;; J) (a,b) \)  
\( \langle \text{proof} \rangle \)

lemma ok-or-not-ok: \( [P(a, b[\text{ok} := \text{True}]); P(a, b[\text{ok} := \text{False}])] \rightarrow P(a, b) \)  
\( \langle \text{proof} \rangle \)

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

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

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

lemma design-eq1: \( (P \vdash Q) = (P \vdash P \land Q) \)  
\( \langle \text{proof} \rangle \)

lemma H1-idem: \( H1 \circ H1 = H1 \)  
\( \langle \text{proof} \rangle \)

lemma H1-idem2: \( (H1 (H1 P)) = (H1 P) \)
\begin{proof}

\textbf{lemma} \( H2\text{idem} \): \( H2 \circ H2 = H2 \)
\end{proof}

\begin{proof}

\textbf{lemma} \( H2\text{idem2} \): \( (H2 \ (H2 \ P)) = (H2 \ P) \)
\end{proof}

\begin{proof}

\textbf{lemma} \( H1\text{-}H2\text{-}commute \): \( H1 \circ H2 = H2 \circ H1 \)
\end{proof}

\begin{proof}

\textbf{lemma} \( H1\text{-}H2\text{-}commute2 \): \( H1 \ (H2 \ P) = H2 \ (H1 \ P) \)
\end{proof}

\begin{proof}

\textbf{lemma} \( \alpha\text{-}d\text{-}eqD \): \( r = r' \implies \text{ok} \ r = \text{ok} \ r' \land \alpha\text{-}d\text{-}more \ r = \alpha\text{-}d\text{-}more \ r' \)
\end{proof}

\begin{proof}

\textbf{lemma} \( \text{design-H1} \): \( (P \vdash Q)\text{ is } H1\text{ healthy} \)
\end{proof}

\begin{proof}

\textbf{lemma} \( \text{design-H2} \): \( (\forall \ a \ b. \ P (a, b\text{⟨ok⟩ := True⟩}) \implies P (a, b\text{⟨ok⟩ := False⟩}) \implies (P \vdash Q)\text{ is } H2\text{ healthy} \)
\end{proof}

\textbf{end}

\section{Reactive processes}

\textbf{theory} \( \text{Reactive-Processes} \)
\textbf{imports} \( \text{Designs} \sim \sim /\text{src/}\text{HOL/Library/Sublist} \)

\textbf{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: \textit{wait}, \textit{tr} and \textit{ref}. The boolean variable \textit{wait} records if the process is waiting for an interaction or has terminated. \textit{tr} records the list (trace) of interactions the process has performed so far. The variable \textit{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.
10.1 Preliminaries

**type-synonym** \(\alpha\) trace = \(\alpha\) list

**fun** list-diff::\(\alpha\) list \(\Rightarrow\) \(\alpha\) list \(\Rightarrow\) \(\alpha\) list option where

- list-diff \([\textbf{]}]\) = Some \(l\)
- list-diff \([\textbf{]}\) \(l\) = 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:

\(l1 - l2 \equiv \text{the (list-diff } l1 l2)\)

**instance** (proof)

**end**

**lemma** list-diff-empty [simp]: the (list-diff \([\textbf{]}]) = \(l\)

(proof)

**lemma** prefix-diff-empty [simp]: \(l - [\textbf{]} = l\)

(proof)

**lemma** prefix-diff-eq [simp]: \(l - l = [\textbf{]}\)

(proof)

**lemma** prefix-diff [simp]: \((l \@ t) - l = t\)

(proof)

**lemma** prefix-subst [simp]: \(l \@ t = m \Rightarrow m - l = t\)

(proof)

**lemma** prefix-subst1 [simp]: \(m = l \@ t \Rightarrow m - l = t\)

(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 :: \(\alpha\) \(\Rightarrow\) \(\alpha\) \(\Rightarrow\) bool

**assumes** refl: ev-eq \(a\) \(a\)

**assumes** comm: ev-eq \(a\) \(b\) = ev-eq \(b\) \(a\)
definition filter-chan-set a cs = (∃ e∈cs. ev-eq a e))

lemma in-imp-not-fcs:
    x∈S =⇒ ¬ filter-chan-set x S
⟨proof⟩

fun tr-filter:'a::ev-eq list ⇒ 'a set ⇒ 'a list where
    tr-filter [] cs = []
    | tr-filter (x#xs) cs = (if (filter-chan-set x cs) then (x#(tr-filter xs cs))
                             else (tr-filter xs cs))
⟨proof⟩

lemma tr-filter-conc: (tr-filter (a@b) cs) = ((tr-filter a cs) @ (tr-filter b cs))
⟨proof⟩

lemma filter-chan-set-hd-tr-filter:
    tr-filter l cs ≠ [] −→ filter-chan-set (hd (tr-filter l cs)) cs
⟨proof⟩

lemma tr-filter-conc-eq1:
    (a@b = (tr-filter (a@c) cs)) −→ (b = (tr-filter c cs))
⟨proof⟩

lemma tr-filter-conc-eq2:
    (a@b = (tr-filter (a@c) cs)) −→ (a = (tr-filter a cs))
⟨proof⟩

lemma tr-filter-conc-eq:
    (a@b = (tr-filter (a@c) cs)) = (b = (tr-filter c cs) & a = (tr-filter a cs))
⟨proof⟩

lemma tr-filter-conc-eq3:
    (b = (tr-filter (a@c) cs)) = (∃ b1 b2. b=b1@b2 & b2 = (tr-filter c cs) & b1 =
                              (tr-filter a cs))
⟨proof⟩

lemma tr-filter-un:
    tr-filter l (s1 ∪ s2) = tr-filter (tr-filter l s1) s2
⟨proof⟩

instantiation list :: (ev-eq) ev-eq
begin
fun ev-eq-list where
    ev-eq-list [] [] = True
    | ev-eq-list l [] = False
    | ev-eq-list [] l = False
    | ev-eq-list (x#xs) (y#ys) = (if (ev-eq x y) then (ev-eq-list xs ys) else False)
end
instance
(proof)
end

10.2 Definitions

abbreviation subl :: 'a list ⇒ 'a list ⇒ bool (- ≤ -)
where l1 ≤ l2 == Sublist.prefixeq l1 l2

lemma list-diff-empty-eq: l1 - l2 = [] ⇒ l2 ≤ l1 ⇒ l1 = l2
(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 +
    wait :: bool
    tr :: 'ϑ trace
    ref :: 'ϑ refusal

Note that we define here the class of UTP alphabets that contain wait, tr and 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
type-synonym ('ϑ, 'σ) relation-rp = ('ϑ, 'σ) alphabet-rp relation

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

definition spec :: [bool, bool, ('ϑ, 'σ) relation-rp] ⇒ ('ϑ, 'σ) relation-rp
where spec b b P ≡ λ (A, A'). P (A[wait := b], A'[ok := b])

abbreviation Specifft (¬t'f) where (P)'t where (P)'t ≡ spec True True P
abbreviation Speciff (¬f') where (P)'f where (P)'f ≡ spec False False P
abbreviation Specifft (¬f't) where (P)'f where (P)'f ≡ spec True False P
abbreviation Speciff (¬t'f) where (P)'t where (P)'t ≡ spec False True P

definition R1::(('ϑ, 'σ) alphabet-rp) Healthiness-condition
where R1 (P) ≡ λ(A, A'). (P (A, A')) ∧ (tr A ≤ tr A')

definition R2::(('ϑ, 'σ) alphabet-rp) Healthiness-condition
where R2 (P) ≡ λ(A, A'). (P (A[tr := []], A'[tr := tr A' - tr A]) ∧ tr A ≤ tr A')

definition IIrea
where IIrea ≡ λ(A, A'). (¬ok A ∧ tr A ≤ tr A') ∨ (ok A' ∧ tr A = tr A')
\[ \land (\text{wait } A = \text{wait } A') \land \text{ref } A = \text{ref } A' \land \text{more } A = \text{more } A') \]

**definition** $R3::(\langle \emptyset, \sigma \rangle \text{ alphabet-rp}) \text{ Healthiness-condition}$

**where** $R3 (P) \equiv (\Pi \text{rea} \triangleleft \text{wait} \circ \text{fst} \triangleright P)$

**definition** $R::(\langle \emptyset, \sigma \rangle \text{ alphabet-rp}) \text{ Healthiness-condition}$

**where** $R \equiv R3 \circ R2 \circ R1$

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

### 10.3 Proofs

**lemma** $\text{tr-filter-empty}$

\[ \text{simp: } \text{tr-filter } l \{\} = l \]

(*proof*)

**lemma** $\text{trf-imp-filters}$

\[ [\text{xs} = \text{tr-filter } y s \text{ cs};\ \text{xs} \neq \{\}] \implies \text{filter-chan-set} (\text{hd } \text{xs}) \text{ cs} \]

(*proof*)

**lemma** $\text{filters-imp-trf}$

\[ [\text{filter-chan-set } x \text{ cs};\ \text{xs} = \text{tr-filter } y s \text{ cs}] \implies x#\text{xs} = \text{tr-filter } (x#y s) \text{ cs} \]

(*proof*)

**lemma** $\text{alpha-d-more-eqI}$

\[ \text{assumes } tr \ r = tr \ r' \ \text{wait } r = \text{wait } r' \ \text{ref } r = \text{ref } r' \ \text{more } r = \text{more } r' \]

\[ \text{shows } \alpha-d.\text{more } r = \alpha-d.\text{more } r' \]

(*proof*)

**lemma** $\text{alpha-d-more-eqE}$

\[ \text{assumes } \alpha-d.\text{more } r = \alpha-d.\text{more } r' \]

\[ \text{obtains } tr \ r = tr \ r' \ \text{wait } r = \text{wait } r' \ \text{ref } r = \text{ref } r' \ \text{more } r = \text{more } r' \]

(*proof*)

**lemma** $\text{alpha-rp-eqE}$

\[ \text{assumes } r = r' \]

\[ \text{obtains } ok \ r = ok \ r' \ tr \ r = tr \ r' \ \text{wait } r = \text{wait } r' \ \text{ref } r = \text{ref } r' \ \text{more } r = \text{more } r' \]

(*proof*)

**lemma** $\text{R-idem}$

\[ R \circ R = R \]

(*proof*)

**lemma** $\text{R-idem2}$

\[ R (R \ P) = R \ P \]

(*proof*)

**lemma** $\text{R1-idem}$

\[ R1 \circ R1 = R1 \]

(*proof*)

**lemma** $\text{R1-idem2}$

\[ R1 (R1 \ x) = R1 \ x \]
lemma R2-idem: \( R2 \circ R2 = R2 \)
(\textit{proof})

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

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

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

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

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

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

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

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

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

lemma R-is-R1:
  \hspace{1em} \textit{assumes} \ A: \text{P is R healthy}
  \hspace{1em} \textit{shows} \text{P is R1 healthy}
(\textit{proof})

lemma R-is-R2:
  \hspace{1em} \textit{assumes} \ A: \text{P is R healthy}
  \hspace{1em} \textit{shows} \text{P is R2 healthy}
(\textit{proof})

lemma R-is-R3:
  \hspace{1em} \textit{assumes} \ A: \text{P is R healthy}
  \hspace{1em} \textit{shows} \text{P is R3 healthy}
(\textit{proof})

lemma R-disj:
  \hspace{1em} \textit{assumes} \ A: \text{P is R healthy}
assumes $B$: $Q$ is $R$ healthy
shows $(P \lor Q)$ is $R$ healthy
(proof)

lemma $R$-disj2: $R (P \lor Q) = (R P \lor R Q)$
(proof)

lemma $R1$-comp:
assumes $P$ is $R1$ healthy
and $Q$ is $R1$ healthy
shows $(P; ; Q)$ is $R1$ healthy
(proof)

lemma $R1$-comp2:
assumes $A$: $P$ is $R1$ healthy
assumes $B$: $Q$ is $R1$ healthy
shows $R1 (P; ; Q) = ((R1 P); ; Q)$
(proof)

lemma $J$-is-$R1$: $J$ is $R1$ healthy
(proof)

lemma $J$-is-$R2$: $J$ is $R2$ healthy
(proof)

lemma $R1$-$H2$-commute2: $R1 (H2 P) = H2 (R1 P)$
(proof)

lemma $R1$-$H2$-commute: $R1 o H2 = H2 o R1$
(proof)

lemma $R2$-$H2$-commute2: $R2 (H2 P) = H2 (R2 P)$
(proof)

lemma $R2$-$H2$-commute: $R2 o H2 = H2 o R2$
(proof)

lemma $R3$-$H2$-commute2: $R3 (H2 P) = H2 (R3 P)$
(proof)

lemma $R3$-$H2$-commute: $R3 o H2 = H2 o R3$
(proof)

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

43
lemma \( R \text{meet} \):
assumes \( A: x \text{ is } R \text{ healthy} \)
and \( B: y \text{ is } R \text{ healthy} \)
shows \((x \sqcup y) \text{ is } R \text{ healthy}\)
(proof)

lemma \( R \text{-H2-commute} \): \( R \circ H2 = H2 \circ R \)
(proof)

lemma \( R \text{-H2-commute2} \): \( R (H2 P) = H2 (R P) \)
(proof)

definition \( \text{is-CSP-process} \): \( \exists (\vartheta, \sigma) \text{ relation-rp} \Rightarrow \text{bool} \)
where \( \text{is-CSP-process} P \equiv P \text{ is } \text{CSP1 healthy and } P \text{ is } \text{CSP2 healthy and } P \text{ is } R \text{ healthy} \)

lemmas \( \text{csp-defs} = \text{CSP1-def J-csp-def CSP2-def is-CSP-process-def} \)

lemma \( \text{is-CSP-processE1} \) [elim]:
assumes \( \text{is-CSP-process} P \)
obtains \( P \text{ is } \text{CSP1 healthy} \text{ P is } \text{CSP2 healthy} \text{ P is } R \text{ healthy} \)
(proof)
lemma is-CSP-processE2 [elim?]:
  assumes is-CSP-process P
  obtains CSP1 P = P CSP2 P = P R P = P
 ⟨proof⟩

11.2 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-CSP2:
  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:
\( (CSP2 P)(a, b\{ok:=True\}) \implies (P(a, b\{ok:=False\})) \)
(proof)

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

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

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

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

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

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

lemma disj-CSP2:
P is CSP2 healthy ==> Q is CSP2 healthy ==> (P \lor Q) is CSP2 healthy
(proof)

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

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

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

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

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 $(P ; ; Q)$
(proof)

lemma rd-ind-wait: $(R(¬(P \downarrow_1) \vdash (P \downarrow_1)))$
\[\vdash (\lambda (A, A'). P (A, A'\{ok := False\}))\]
\[\vdash (\lambda (A, A'). P (A, A'\{ok := True\}))\]
(proof)

lemma rd-H1: $(R((¬(\lambda (A, A'). P (A, A'\{ok := False\})))))$
\[\vdash (\lambda (A, A'). P (A, A'\{ok := True\}))\]
\[= (R ((¬ H1 \ (\lambda (A, A'). P (A, A'\{ok := False\}))))\]
\[\vdash (H1 \ (\lambda (A, A'). P (A, A'\{ok := True\}))))\]
(proof)

lemma rd-H1-H2: $(R((¬ H1 \ (\lambda (A, A'). P (A, A'\{ok := False\}))))$
\[\vdash (H1 \ (\lambda (A, A'). P (A, A'\{ok := True\}))))\]
\[= (R((¬(H1 o H2) \ (\lambda (A, A'). P (A, A'\{ok := False\}))))\]
\[\vdash (H1 o H2) \ (\lambda (A, A'). P (A, A'\{ok := True\}))))\]
(proof)

lemma rd-H1-H2-R-H1-H2:
\[\vdash (H1 o H2) \ (\lambda (A, A'). P (A, A'\{ok := False\})))\]
\[\vdash (H1 o H2) \ (\lambda (A, A'). P (A, A'\{ok := True\}))))\]
\[= (R o H1 o H2) P\]
(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)
11.3 CSP processes and reactive designs

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

**lemma** rd-is-CSP1: \( (R (r \vdash p)) \) is CSP1 healthy

**lemma** rd-is-CSP2:
- **assumes** \( A : \forall \ a \ b. r (a, b|\{ok := \text{True}\}) \rightarrow r (a, b|\{ok := \text{False}\}) \)
- **shows** \( (R (r \vdash p)) \) is CSP2 healthy

**lemma** rd-is-CSP:
- **assumes** \( A : \forall \ a \ b. r (a, b|\{ok := \text{True}\}) \rightarrow r (a, b|\{ok := \text{False}\}) \)
- **shows** is-CSP-process \( (R (r \vdash p)) \)

**lemma** CSP-is-rd:
- **assumes** \( A : \) is-CSP-process \( P \)
- **shows** \( P = (R (\neg (P f f) \vdash (P t f))) \)

end

12 Circus actions

**theory** Circus-Actions

**imports** ~~/src/HOL/HOLCF/HOLCF CSP-Processes

begin

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

**default-sort** type

12.1 Definitions

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

**typedef** \( \langle \vartheta::\text{ev-eq}, \sigma \rangle \) action = \{ \langle \vartheta, \sigma \rangle \ text{relation-rp. is-CSP-process} \ P \}

**morphisms** relation-of action-of

**proof**

**print-theorems**

49
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 $relation_of$ and $action_of$ as well as the usual axioms expressing the bijection $action_of (relation_of ?x) = ?x$ and $?y \in \{p. \ is-CSP-process\ p\} \implies relation_of (action_of ?y) = ?y.$

**Lemma** $relation_of-CSP$: is-CSP-process $(relation_of x)$

**Lemma** $relation_of-CSP1$: $(relation_of x)$ is CSP1 healthy

**Lemma** $relation_of-CSP2$: $(relation_of x)$ is CSP2 healthy

**Lemma** $relation_of-R$: $(relation_of x)$ is R healthy

### 12.2 Proofs

In the following, Circus actions are proved to be an instance of the $Complete_Lattice$ class.

**Lemma** $relation_of-spec-f-f$:
$$\forall a b. (relation_of y \rightarrow relation_of x) (a, b) \implies (relation_of y)^f \ (a||tr := [], b) \implies (relation_of x)^f \ (a||tr := [], b)$$

(\textit{proof})

**Lemma** $relation_of-spec-t-f$:
$$\forall a b. (relation_of y \rightarrow relation_of x) (a, b) \implies (relation_of y)^f \ (a||tr := [], b) \implies (relation_of x)^f \ (a||tr := [], b)$$

(\textit{proof})

**Instantiation** action::(ev-eq, type) below

**Definition** ref-def : $P \subseteq Q \equiv [(relation_of Q) \rightarrow (relation_of P)]$

**Instance** (\textit{proof})

**Instance** action :: (ev-eq, type) po
(\textit{proof})

**Instantiation** action :: (ev-eq, type) lattice

**Definition** inf-action : $(inf P Q \equiv action_of ((relation_of P) \cap (relation_of Q)))$

**Definition** sup-action : $(sup P Q \equiv action_of ((relation_of P) \cup (relation_of Q)))$
\textbf{definition} \texttt{less-eq-action: (less-eq (P::(a, b) action) Q \equiv P \sqsubseteq Q)}

\textbf{definition} \texttt{less-action: (less (P::(a, b) action) Q \equiv P \sqsubseteq Q \land \neg Q \sqsubseteq P)}

\textit{instance}

⟨\texttt{proof}⟩

end

\textbf{lemma} \texttt{bot-is-action: R (false \vdash true) \in \{ p. \text{is-CSP-process} p \}}

⟨\texttt{proof}⟩

\textbf{lemma} \texttt{bot-eq-true: R (false \vdash true) = R true}

⟨\texttt{proof}⟩

\textbf{instantiation} \texttt{action :: (ev-eq, type) bounded-lattice}

\textbf{begin}

\textbf{definition} \texttt{bot-action: (bot::(a, b) action) \equiv action-of (R(false \vdash true))}

\textbf{definition} \texttt{top-action: (top::(a, b) action) \equiv action-of (R(true \vdash false))}

\textit{instance}

⟨\texttt{proof}⟩

end

\textbf{lemma} \texttt{relation-of-top: relation-of top = R(true \vdash false)}

⟨\texttt{proof}⟩

\textbf{lemma} \texttt{relation-of-bot: relation-of bot = R true}

⟨\texttt{proof}⟩

\textbf{lemma} \texttt{non-emptyE: assumes A \neq \{\} obtains x where x : A}

⟨\texttt{proof}⟩

\textbf{lemma} \texttt{CSP1-Inf:}
\textbf{assumes} \texttt{*:A \neq \{\} shows (\bigsqcap relation-of \ A) is CSP1 healthy}

⟨\texttt{proof}⟩

\textbf{lemma} \texttt{CSP2-Inf:}
\textbf{assumes} \texttt{*:A \neq \{\} shows (\bigsqcap relation-of \ A) is CSP2 healthy}

⟨\texttt{proof}⟩

\textbf{lemma} \texttt{R-Inf:}
\textbf{assumes} \texttt{*:A \neq \{\} shows (\bigsqcap relation-of \ A) is R healthy}

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

lemma Inf-is-action: A ≠ {} → (∩ relation-of ' A) ∈ {p. is-CSP-process p} (proof)

lemma CSP1-Sup: A ≠ {} → (∪ relation-of ' A) is CSP1 healthy (proof)

lemma CSP2-Sup: A ≠ {} → (∪ relation-of ' A) is CSP2 healthy (proof)

lemma R-Sup: A ≠ {} → (∪ relation-of ' A) is R healthy (proof)

lemma CSP-Sup: A ≠ {} → is-CSP-process (∪ relation-of ' A) (proof)

lemma Sup-is-action: A ≠ {} → (∪ relation-of ' A) ∈ {p. is-CSP-process p} (proof)

lemma relation-of-Sup:
 A ≠ {} → 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

13 Circus variables

theory Var-list
imports Main
begin

Circus variables are represented by a stack (list) of values. They are char-

52
acterized by two functions, \( \text{select} \) and \( \text{update} \). The Circus variable type is defined as a tuple \( \langle \text{select} \ast \text{update} \rangle \) with a list of values instead of a single value.

**type-synonym** \( \langle \alpha, \gamma \rangle \text{ var-list} = \langle \alpha \Rightarrow \text{ list} \ast \gamma \Rightarrow \text{ list} \Rightarrow \gamma \Rightarrow \alpha \rangle \)

The \( \text{select} \) function returns the top value of the stack.

**definition** \( \text{select} :: \langle \alpha, \gamma \rangle \text{ var-list} \Rightarrow \gamma \Rightarrow \alpha \)

\[ \text{select } f \equiv \lambda A. \text{hd } ((\text{fst } f) A) \]

The \( \text{increase} \) function pushes a new value to the top of the stack.

**definition** \( \text{increase} :: \langle \alpha, \gamma \rangle \text{ var-list} \Rightarrow \alpha \Rightarrow \gamma \Rightarrow \gamma \Rightarrow \gamma \)

\[ \text{increase } f \text{ val } \equiv ((\text{snd } f) \lambda l. \text{val} \# l) \]

The \( \text{increase}0 \) function pushes an arbitrary value to the top of the stack.

**definition** \( \text{increase}0 :: \langle \alpha, \gamma \rangle \text{ var-list} \Rightarrow \gamma \Rightarrow \gamma \Rightarrow \gamma \)

\[ \text{increase}0 f \equiv ((\text{snd } f) \lambda l. \text{ SOME val. True } \# (\text{tl } l)) \]

The \( \text{decrease} \) function pops the top value of the stack.

**definition** \( \text{decrease} :: \langle \alpha, \gamma \rangle \text{ var-list} \Rightarrow \gamma \Rightarrow \gamma \Rightarrow \gamma \)

\[ \text{decrease } f \equiv ((\text{snd } f) \lambda l. \text{tl } l) \]

The \( \text{update} \) function updates the top value of the stack.

**definition** \( \text{update} :: \langle \alpha, \gamma \rangle \text{ var-list} \Rightarrow \alpha \Rightarrow \gamma \Rightarrow \gamma \Rightarrow \gamma \)

\[ \text{update } f \text{ upd } \equiv ((\text{snd } f) \lambda l. \text{ upd } \text{ (hd } l) \# (\text{tl } l)) \]

The \( \text{update}0 \) function initializes the top of the stack with an arbitrary value.

**definition** \( \text{update}0 :: \langle \alpha, \gamma \rangle \text{ var-list} \Rightarrow \gamma \Rightarrow \gamma \Rightarrow \gamma \)

\[ \text{update}0 f \equiv ((\text{snd } f) \lambda l. \text{ SOME upd. True } \# (\text{tl } l)) \]

**axiomatization** \( \text{where } \text{select-increase}: \text{select } v \text{ (increase } v \text{ a } s) = a \)

The \( \text{VAR-LIST} \) function allows to retrieve a Circus variable from its name.

**syntax** \( \text{-VAR-LIST} :: \text{id} \Rightarrow \langle \alpha, \gamma \rangle \text{ var-list} \ (\text{VAR'}-\text{LIST} \ -) \)

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

end

14 Denotational semantics of Circus actions

**theory** \( \text{Denotational-Semantics} \)

**imports** \( \text{Circus-Actions} \text{ Var-list} \)

**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.
14.1 Skip

definition Skip :: ('\theta::ev-eq,'sigma) action where
Skip ≡ action-of
(R (true ⊨ \lambda(A, A'), tr A' = tr A \land \neg \text{wait } A' \land more A = more A'))

lemma Skip-is-action:
(R (true ⊨ \lambda(A, A'), tr A' = tr A \land \neg \text{wait } A' \land more A = more A')) ∈ {p. is-CSP-process p}
⟨proof⟩

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

lemma relation-of-Skip:
relation-of Skip =
(R (true ⊨ \lambda(A, A'), tr A' = tr A \land \neg \text{wait } A' \land more A = more A'))
⟨proof⟩

definition CSP3::('\theta::ev-eq,'sigma) alphabet-rp) Healthiness-condition
where CSP3 (P) ≡ relation-of Skip ; ; P

definition CSP4::('\theta::ev-eq,'sigma) alphabet-rp) Healthiness-condition
where CSP4 (P) ≡ P ; ; relation-of Skip

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

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

lemma Skip-comp-absorb: (relation-of Skip ; ; relation-of Skip) = relation-of Skip
⟨proof⟩

14.2 Stop

definition Stop :: ('\theta::ev-eq,'sigma) action where
Stop ≡ action-of (R (true ⊨ \lambda(A, A'), tr A' = tr A \land \text{wait } A'))

lemma Stop-is-action:
(R (true ⊨ \lambda(A, A'), tr A' = tr A \land \text{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 ⊨ \lambda(A, A'), tr A' = tr A \land \text{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)

14.3 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)

14.4 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')))

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')) ; ; relation-of Ac) ∈ \{p. is-CSP-process p\}
(proof)

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

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-CSP3: relation-of (state-update-before sc Ac) is CSP3 healthy
\textbf{lemma} \textit{state-update-before-is-CSP4}:
\begin{itemize}
    \item \textbf{assumes} $A : \text{relation-of} \; Ac \; \text{is \; CSP4\; healthy}$
    \item \textbf{shows} \text{relation-of} \; (\text{state-update-before \; sc \; Ac}) \; \text{is \; CSP4\; healthy}
\end{itemize}
(\textit{proof})

\textbf{definition} \textit{state-update-after} :: $'\sigma$ relation $\Rightarrow \langle '\sigma', '\sigma' \rangle$ action $\Rightarrow \langle '\sigma', '\sigma' \rangle$ action
\begin{itemize}
    \item \textbf{where} \text{state-update-after sc Ac} $=$ \text{action-of}(\text{relation-of Ac ; ;} $R$ (true $\vdash$ $(\lambda(A, A'). \text{sc} \; \text{(more A, more A') \& \neg \text{wait A'} \& \text{tr A} \; = \; \text{tr A'})}$)
\end{itemize}
\textbf{lemma} \textit{state-update-after-is-action}:
\begin{itemize}
    \item \text{relation-of} \; (\text{state-update-after sc Ac}) \; \in \{ \text{p. \; is-CSP-process} \; p \} 
\end{itemize}
(\textit{proof})

\textbf{lemmas} \textit{state-update-after-is-CSP} = \textit{state-update-after-is-action}[simplified]

\textbf{lemma} \textit{relation-of-state-update-after}:
\begin{itemize}
    \item \text{relation-of} \; (\text{state-update-after sc Ac}) \; = \; (\text{relation-of Ac ; ;} \; R \; (\text{true} \; \vdash \; (\lambda(A, A'). \text{sc} \; \text{(more A, more A') \& \neg \text{wait A'} \& \text{tr A} \; = \; \text{tr A'})}) \; \in \{ \text{p. \; is-CSP-process} \; p \} 
\end{itemize}
(\textit{proof})

\textbf{lemma} \textit{mono-state-update-after}:
\begin{itemize}
    \item \textit{mono} \; (\text{state-update-after sc})
\end{itemize}
(\textit{proof})

\textbf{lemma} \textit{state-update-after-is-CSP3}:
\begin{itemize}
    \item \textbf{assumes} $A : \text{relation-of} \; Ac \; \text{is \; CSP3\; healthy}$
    \item \textbf{shows} \text{relation-of} \; (\text{state-update-after sc Ac}) \; \text{is \; CSP3\; healthy}
\end{itemize}
(\textit{proof})

\textbf{lemma} \textit{state-update-after-is-CSP4}:
\begin{itemize}
    \item \text{relation-of} \; (\text{state-update-after sc Ac}) \; \text{is \; CSP4\; healthy}
\end{itemize}
(\textit{proof})

\section{14.5 Sequential composition}

\textbf{definition} \textit{Seq}:: $('\sigma', '\sigma')$ action $\Rightarrow \langle '\sigma', '\sigma' \rangle$ action $\Rightarrow \langle '\sigma', '\sigma' \rangle$ action \text{(infixl ' ; ; ' 24)}
\begin{itemize}
    \item \textbf{where} $P \; ; ; Q \equiv \text{action-of} \; (\text{relation-of} \; P \; ; ; \; \text{relation-of} \; Q)$
\end{itemize}
\textbf{lemma} \textit{Seq-is-action}:
\begin{itemize}
    \item \text{relation-of} \; (\text{relation-of} \; P \; ; ; \; \text{relation-of} \; Q) \; \in \{ \text{p. \; is-CSP-process} \; p \}
\end{itemize}
(\textit{proof})

\textbf{lemmas} \textit{Seq-is-CSP} = \textit{Seq-is-action}[simplified]
lemma relation-of-Seq: relation-of \((P \cdot; Q)\) = (relation-of P ; relation-of Q)  
(proof)

lemma mono-Seq: mono \((op \cdot; P)\)  
(proof)

lemma CSP3-imp-left-Skip:  
assumes A: relation-of P is CSP3 healthy  
shows \((\text{Skip} \cdot; P) = P\)  
(proof)

lemma CSP4-imp-right-Skip:  
assumes A: relation-of P is CSP4 healthy  
shows \((P \cdot; \text{Skip}) = P\)  
(proof)

lemma Seq-assoc: \((A \cdot; (B \cdot; C))\) = \(((A \cdot; B) \cdot; C)\)  
(proof)

lemma Skip-absorb: \((\text{Skip} \cdot; \text{Skip}) = \text{Skip}\)  
(proof)

14.6 Internal choice

definition Ndet::\((\vartheta::ev-eq, \sigma) action \Rightarrow (\vartheta, \sigma) action \Rightarrow (\vartheta, \sigma) action \text{ (infixl \(\cap\) 18)}\)  
where \(P \cap Q \equiv \text{action-of } ((\text{relation-of } P) \lor (\text{relation-of } Q))\)

lemma Ndet-is-action: \(((\text{relation-of } P) \lor (\text{relation-of } Q)) \in \{p.\ is-CSP-process p\}\)  
(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 \((op \cap P)\)  
(proof)

14.7 External choice

definition Det::\((\vartheta::ev-eq, \sigma) action \Rightarrow (\vartheta, \sigma) action \Rightarrow (\vartheta, \sigma) action \text{ (infixl \([+]\) 18)}\)  
where \(P [+] Q \equiv \text{action-of}(R(\neg((\text{relation-of } P)^f) \land \neg((\text{relation-of } Q)^f)) \vdash (((\text{relation-of } P)^f) \land (\text{relation-of } Q)^f) \land A' \land \text{tr } A = \text{tr } A' \land \text{wait } A'')\)

notation(xsymbol)
Det (infixl □ 18)

lemma Det-is-action:
(R((¬(relation-of P)↑f) ∧ ¬(relation-of Q)↑f) ⊨
 (((relation-of P)↑f ∧ (relation-of Q)↑f))
 < λ(A, A'). tr A = tr A' ∧ wait A' ▷
 ((relation-of P)↑f ∨ (relation-of Q)↑f)))) ∈ {p. is-CSP-process p}
⟨proof⟩

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

lemma relation-of-Det:
relation-of (P □ Q) = (R((¬(relation-of P)↑f) ∧ ¬(relation-of Q)↑f) ⊨
 (((relation-of P)↑f ∧ (relation-of Q)↑f))
 < λ(A, A'). tr A = tr A' ∧ wait A' ▷
 ((relation-of P)↑f ∨ (relation-of Q)↑f))))
⟨proof⟩

lemma mono-Det: mono (op [+] P)
⟨proof⟩

14.8 Reactive design assignment

definition rd-assign s = action-of (R (true ⊨ λ(A, A'). ref A' = ref A ∧ tr A' = tr A ∧ ¬wait A' ∧ more A' = s))

lemma rd-assign-is-action:
(R (true ⊨ λ(A, A'). ref A' = ref A ∧ tr A' = tr A ∧ ¬wait A' ∧ more A' = s))
∈ {p. 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⟩

14.9 Local state external choice

definition Loc::'σ ⇒ ('ψ::ev-eq,'σ) action ⇒ 'σ ⇒ ('θ,'σ) action ⇒ ('θ,'σ) action

('')loc - ⋅ - ') ⊕ ('')loc - ⋅ - ')
where (loc s1 • P) □ (loc s2 • Q) ≡
((rd-assign s1)' ▷ 'P) □ ((rd-assign s2)' ▷ 'Q)
14.10 Schema expression

definition Schema :: 'σ relation ⇒ ('∅::ev-eq,'σ) action where
Schema sc ≡ action-of(R ((λ(A,A¹). (Pre sc) (more A)) ⊨
(λ(A,A¹). sc (more A, more A¹) ∧ ¬wait A¹ ∧ tr A = tr A¹)))

lemma Schema-is-action:
(R ((λ(A,A¹). (Pre sc) (more A)) ⊨
(λ(A,A¹). sc (more A, more A¹) ∧ ¬wait A¹ ∧ tr A = tr A¹))) ∈ {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,A¹). (Pre sc) (more A)) ⊨
(λ(A,A¹). sc (more A, more A¹) ∧ ¬wait A¹ ∧ tr A = tr A¹)))
(proof)

lemma Schema-is-state-update-before: Schema u = state-update-before u Skip
(proof)

14.11 Parallel composition

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

fun MergeSt :: 'σ local-state ⇒ 'σ local-state ⇒ ('∅,'σ) relation-rp where
MergeSt (s1,s¹) (s2,s²) = (λ(S,S¹). (s1' s¹) (more S) = (more S¹));
(λ(S,(ϕ,σ) alphabet-rp, S¹). (s2' s²) (more S) = (more S¹))

definition listCons :: '∅ ⇒ '∅ list list ⇒ '∅ list list (- # # -) where
a ## l = ((map (Cons a)) l)

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

lemma set-Cons1: tr1 \in set l \implies a \# tr1 \in op \# a \# set l
(proof)

lemma tr-in-set-eq: (tr1 \in op \# b \# set l) = (tr1 \neq [] \land \text{hd tr1} = b \land \text{tl tr1} \in set l)
(proof)

definition M-par::(\emptyset :: ev-eq, \sigma) alpha-rp-scheme \Rightarrow ('\sigma \Rightarrow '\sigma \Rightarrow '\sigma)
\Rightarrow ('\emptyset, '\sigma) alpha-rp-scheme \Rightarrow ('\sigma \Rightarrow '\sigma \Rightarrow '\sigma)
\Rightarrow ('\emptyset set) \Rightarrow ('\emptyset, '\sigma) relation-rp where
M-par s1 x1 s2 x2 cs =
((\lambda (S, S'). (\text{diff-tr S' S}) \in ParMerge (\text{diff-tr s1 S}) (\text{diff-tr s2 S}) cs &
ev-eq tr1 (\text{tr-filter tr1 cs}) (\text{tr-filter tr1 cs}) \land
((\lambda (S, S'). \text{wait s1} \lor \text{wait s2}) \land
\text{ref S'} \subseteq ((\text{ref s1}) \cup (\text{ref s2}) \cap cs) \cup ((\text{ref s1}) \cap (\text{ref s2}) \cap cs)))))
<> \text{wait s1} \lor \text{wait s2}) \land
((\lambda (S, S'). \text{wait s1} \lor \text{wait s2}) \land
\text{MergeSt ((more s1), x1) ((more s2), x2)))

definition Par::(\emptyset :: ev-eq, \sigma) action \Rightarrow
('\sigma \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow ('\emptyset set) \Rightarrow ('\sigma \Rightarrow '\sigma \Rightarrow '\sigma) \Rightarrow ('\emptyset, '\sigma) action \Rightarrow ('\emptyset, '\sigma) action (- - - - - - -) where
Par A1 \| ns1 | cs | ns2 \| A2 \equiv (\text{action-of } R ((\lambda (S, S')).
\text{\exists tr1 tr2. (\text{relation-of A1)} \land ; ; (\lambda (S, S'). tr1 = (tr S)) (S, S') \land
\text{spec False (wait S)} (\text{relation-of A2}) ; ; (\lambda (S, \cdot). tr2 = (tr S))) (S, S') \land
((\text{tr-filter tr1 cs}) = (\text{tr-filter tr2 cs}))) \land
\text{\exists tr1 tr2. (\text{spec False (wait S)} (\text{relation-of A1}) ; ; (\lambda (S, \cdot). tr1 = (tr S)) (S, S') \land
(\text{relation-of A2}) \land ; ; (\lambda (S, S'). tr2 = (tr S))) (S, S') \land
((\text{tr-filter tr1 cs}) = (\text{tr-filter tr2 cs}))) \land
((\lambda (S, S'). \text{\exists s1 s2. ((\lambda (A, A'). (\text{relation-of A1}) \land ; ; (M-par s1 ns1 s2 ns2 cs) (S, S'))))))

lemma Par-is-action: R ((\lambda (S, S')).
\text{\exists tr1 tr2. (\text{relation-of A1)} \land ; ; (\lambda (S, S'). tr1 = (tr S)) (S, S') \land
\text{spec False (wait S)} (\text{relation-of A2}) ; ; (\lambda (S, \cdot). tr2 = (tr S))) (S, S') \land
((\text{tr-filter tr1 cs}) = (\text{tr-filter tr2 cs}))) \land
\text{\exists tr1 tr2. (\text{spec False (wait S)} (\text{relation-of A1}) ; ; (\lambda (S, \cdot). tr1 = (tr S)) (S, S') \land
(\text{relation-of A2}) \land ; ; (\lambda (S, S'). tr2 = (tr S))) (S, S') \land
((\text{tr-filter tr1 cs}) = (\text{tr-filter tr2 cs}))) \land
((\lambda (S, S'). \text{\exists s1 s2. ((\lambda (A, A'). (\text{relation-of A1}) \land ; ; (M-par s1 ns1 s2 ns2 cs) (S, S'))))) \in \{ p. is-CSP-process p \}
(proof)
lemmas Par-is-CSP = Par-is-action[simplified]

lemma relation-of-Par:
relation-of (A1 \parallel ns1 | cs | ns2 \parallel A2) = (R ((\lambda (S, S')).
\neg (\exists tr1 tr2. ((relation-of A1)'_f ; ; (\lambda (S, S'). tr1 = (tr S))) (S, S')
\land (spec False (wait S) (relation-of A2) ; ; (\lambda (S, S'). tr2 = (tr S))) (S, S')
\land ((tr-filter tr1 cs) = (tr-filter tr2 cs)))) \land
\neg (\exists tr1 tr2. (spec False (wait S) (relation-of A1); ; (\lambda(S, -). tr1 = tr S)) (S, S')
\land ((relation-of A2)'_f ; ; (\lambda (S, S'). tr2 = (tr S))) (S, S')
\land ((tr-filter tr1 cs) = (tr-filter tr2 cs))))\) 

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

(\text{proof})

14.12 Local parallel block

definition ParLoc::'a \Rightarrow (\forall \Rightarrow 'a \Rightarrow (\forall::ev-eq, 'a) action \Rightarrow \forall set \Rightarrow 'a \Rightarrow ('a \Rightarrow 'a)
\Rightarrow 'a) \Rightarrow ('a::'a) action \Rightarrow ('a::'a) action
'((\lambda(par | - - - )) [ - ] '((\lambda(par | - - - )))

where
(par s1 | ns1 \cdot P) \parallel cs \parallel (par s2 | ns2 \cdot Q) \equiv ((\text{rd-assign s1})'; 'P) \parallel ns1 | cs | ns2 \parallel ((\text{rd-assign s2})'; 'Q)

14.13 Assignment

definition ASSIGN::('v, 'a) var-list \Rightarrow (\forall \Rightarrow 'v) \Rightarrow (\forall::ev-eq, 'a) action where
ASSIGN x e \equiv action-of (R (true \Rightarrow (\lambda (S, S'). tr S' = tr S \land \neg\text{wait} S' \land
\text{more} S' = (update x (\lambda-. (e (\text{more} S)))) (\text{more} S))))

syntax -assign::id \Rightarrow ('a \Rightarrow 'v) \Rightarrow (\forall::ev-eq, 'a) action \ (\cdot:=\cdot)

translations y := 'v \Rightarrow CONSTR ASSIGN (VAR y) \ 'v

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

(\text{proof})

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

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

(\text{proof})

61
**Lemma** Assign-is-state-update-before: \( \text{ASSIGN } x \ e = \text{state-update-before} \ (\lambda \ (s, s') \ .\ s' = (\text{update } x \ (\lambda - \ (e \ s))) \ s) \ \text{Skip} \)

(\textit{proof})

**14.14 Variable scope**

**Definition** Var::('v, 'σ) var-list ⇒ ('θ, 'σ) action ⇒ ('θ::ev-eq,'σ) action where

\( \text{Var } v \ A \equiv \text{action-of}(\)

\( (R(\text{true} \ ⇒ \ (\lambda \ (A, A'). \ \exists \ a. \ tr \ A' = tr \ A \land \lnot \text{wait} \ A' \land \text{more} \ A' = (\text{increase } v \ a \ (\text{more} \ A)))))); ;

\( (\text{relation-of } A; ;)

\( (R(\text{true} \ ⇒ \ (\lambda \ (A, A'). \ tr \ A' = tr \ A \land \lnot \text{wait} \ A' \land \text{more} \ A' = (\text{decrease } v \ (\text{more} \ A))))))))))

\( \in \{p. \text{is-CSP-process } p\}

(\textit{proof})

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

**Lemma** relation-of-Var:

\( \text{relation-of } (\text{Var } v \ A) =

\( (\text{relation-of } A; ;)

\( (R(\text{true} \ ⇒ \ (\lambda \ (A, A'). \ tr \ A' = tr \ A \land \lnot \text{wait} \ A' \land \text{more} \ A' = (\text{decrease } v \ (\text{more} \ A))))))))))

(\textit{proof})

**Lemma** mono-Var : mono (Var x)

(\textit{proof})

**Definition** Let::('v, 'σ) var-list ⇒ ('θ, 'σ) action ⇒ ('θ::ev-eq,'σ) action where

\( \text{Let } v \ A \equiv \text{action-of}(\)

\( (R(\text{true} \ ⇒ \ (\lambda \ (A, A'). \ tr \ A' = tr \ A \land \lnot \text{wait} \ A' \land \text{more} \ A' = (\text{decrease } v \ (\text{more} \ A))))))))))

\( \in \{p. \text{is-CSP-process } p\}

(\textit{proof})

**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)))))
) ∈ { p. is-CSP-process p}
⟨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⟩

14.15 Guarded action

definition Guard::σ predicate ⇒ (▷ : ev-eq, 'σ) action ⇒ (▷, '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)) ∨
  (¬((g o more o fst)) ∧ (λ (A, A'). tr A' = tr A ∧ wait A')))))

lemma Guard-is-action:
(R (((g o more o fst) → ¬ ((relation-of P)↑ f))) ⊨
 (((g o more o fst) ∧ ((relation-of P)↑ f)) ∨
  (¬((g o more o fst)) ∧ (λ (A, A'). tr A' = tr A ∧ wait A'))))) ∈ { p. is-CSP-process p}
⟨proof⟩

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

lemma relation-of-Guard:
relation-of (g 'κ· P) = (R (((g o more o fst) → ¬ ((relation-of P)↑ f))) ⊨
 (((g o more o fst) ∧ ((relation-of P)↑ f)) ∨
  (¬((g o more o fst)) ∧ (λ (A, A'). tr A' = tr A ∧ wait A')))))
lemma mono-Guard: mono $(\text{Guard } g)$

lemma false-Guard: $\text{false } \& \text{ P } = \text{Stop}$

lemma false-Guard1: $\forall a \ b. \ g(\alpha-rp. \text{more } a) = \text{False } \implies (\text{relation-of } (g \ \& \text{ P })) (a, b) = (\text{relation-of } \text{Stop}) (a, b)$

lemma true-Guard: $\text{true } \& \text{ P } = \text{P}$

lemma true-Guard1: $\forall a \ b. \ g(\alpha-rp. \text{more } a) = \text{True } \implies (\text{relation-of } (g \ \& \text{ P })) (a, b) = (\text{relation-of } \text{P}) (a, b)$

lemma Guard-is-state-update-before: $g \ \& \text{ P } = \text{state-update-before } (\lambda (s, s'). g s) \text{ P}$

14.16 Prefixed action

definition do where
do e ≡ $\lambda (A, A'). \text{tr } A = \text{tr } A' \land (e \ (\text{more } A)) \notin (\text{ref } A') \triangleleft \text{wait } o \text{ snd } \triangleright$
\(\lambda (A, A'). \text{tr } A' = (\text{tr } A \oplus [(e \ (\text{more } A)])\)

definition do-I c S ≡ $(\lambda (A, A'). \text{tr } A' = \text{tr } A' \land (c \ (\text{more } A))) \triangleleft \text{wait } o \text{ snd } \triangleright$
\(\lambda (A, A'). \text{hd } (\text{tr } A' - \text{tr } A) \in S \land (c \ (\text{more } A) = (\text{last } (\text{tr } A'))))$

definition iPrefix c i j S P ≡ $\text{action-of } (R(\text{true } \vdash (\lambda (A, A'). \text{do-I c S})) (A, A') \land \text{more } A' = \text{more } A))'; \text{ P}$

definition oPrefix c S P ≡ $\text{action-of } (R(\text{true } \vdash (\text{do } c) \land (\lambda (A, A'). \text{more } A' = \text{more } A))'); \text{ P}$

definition Prefix0 c S P ≡ $\text{action-of } (R(\text{true } \vdash (\text{do } (\lambda - c)) \land (\lambda (A, A'). \text{more } A' = \text{more } A))'); \text{ P}$
definition
read1::((v ⇒ 'θ) ⇒ (v, 'σ) var-list ⇒ ('θ::ev-eq, 'σ) action ⇒ ('θ, 'σ) action
where read1 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

lemma Prefix-is-action:
(R(true ⊢ (do c) ∧ (λ (A, A'). more A' = more A))) ∈ {p. is-CSP-process p}  
(proof).

lemma Prefix1-is-action:
(R(true ⊢ λ(A, A'). do-I c (S (alpha-rp.more A)) (A, A') ∧ alpha-rp.more A' = alpha-rp.more A)) ∈ {p. is-CSP-process p}  
(proof).

lemma Prefix0-is-action:
(R(true ⊢ (do c) ∧ (λ (A, A'). more A' = more A))) ∈ {p. is-CSP-process p}  
(proof).
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) =
((R(true ⊢ (λ (A, A'). (do-I c (S (more A)) (A, A') & more A' = more A))) ; relation-of P) (proof)

lemma relation-of-oPrefix:
relation-of (oPrefix c P) =
((R(true ⊢ (do c) ∧ (λ (A, A'). more A' = more A))) ; relation-of P) (proof)

lemma relation-of-Prefix0:
relation-of (Prefix0 c P) =
((R(true ⊢ (do (λ -. c)) ∧ (λ (A, A'). more A' = more A))) ; relation-of P) (proof)

lemma mono-iPrefix : mono (iPrefix c i j s)
(proof)

lemma mono-oPrefix : mono (oPrefix c)
(proof)

lemma mono-Prefix0 : mono(Prefix0 c)
(proof)

14.17 Hiding

definition Hide::('θ::ev-eq, 'σ) action ⇒ 'θ set ⇒ ('θ, 'σ) action (infixl \ 18)
where
P \ cs ≡ action-of(R(λ(S, S'). ∃ s. (diff-tr S' S) = (tr-filter (s - (tr S)) cs) &
(relation-of P)(S, S'\tr := s, ref := (ref S' ∪ cs [])))); (relation-of Skip))

definition
hid P cs == (R(λ(S, S'). ∃ s. (diff-tr S' S) = (tr-filter (s - (tr S)) cs) &
(relation-of P)(S, S'\tr := s, ref := (ref S' ∪ cs [])))); (relation-of Skip))

lemma hid-is-R: hid P cs is R healthy
(proof)

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

lemma hid-is-CSP1: hid P cs is CSP1 healthy
(proof)

lemma hid-is-CSP2: hid P cs is CSP2 healthy
(proof)

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

lemma Hide-is-action:
(R(\lambda(S, S'). \exists s. (diff-tr S' S) = (tr-filter (s - (tr S)) cs) &
 (relation-of P)(S, S'(tr := s, ref := (ref S') \cup cs))); (relation-of Skip)) ∈
\{p. is-CSP-process p\}
(proof)

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

lemma relation-of-Hide:
relation-of (P \ cs) = (R(\lambda(S, S'). \exists s. (diff-tr S' S) = (tr-filter (s - (tr S)) cs) &
 (relation-of P)(S, S'(tr := s, ref := (ref S') \cup cs))); (relation-of Skip))
(proof)

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

14.18 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 for-
mulation) 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.

syntax -MU::[idt, idt] ⇒ ('θ, 'σ) action] ⇒ ('θ, 'σ) action (μ - • -)
translations -MU X P == CONST lfp (λ X. P)
(proof)(proof)end
15 Circus syntax

theory Circus-Syntax
imports Denotational-Semantics
keywords alphabet state channel nameset chanset schema action and
  circus-process :: thy-decl
begin

abbreviation list-select::[′r ⇒ ′a list] ⇒ (′r ⇒ ′a) where
  list-select Sel ≡ hd o Sel

abbreviation list-update::[(′a list ⇒ ′a list) ⇒ ′r ⇒ ′r] ⇒ (′a ⇒ ′a) ⇒ ′r ⇒ ′r where
  list-update Upd ≡ λ e. Upd (λ l. (e (hd l))#(tl l))

abbreviation list-update-const::[(′a list ⇒ ′a list) ⇒ ′r ⇒ ′r] ⇒ ′a ⇒ ′r relation where
  list-update-const Upd ≡ λ e. λ (A, A'). A' = Upd (λ l. e#(tl l)) A

abbreviation update-const::[(′a ⇒ ′a) ⇒ ′r ⇒ ′r] ⇒ ′a ⇒ ′r relation where
  update-const Upd ≡ λ e. λ (A, A'). A' = Upd (λ -. e) A

syntax
  -synt-assign :: id ⇒ ′a ⇒ ′b relation (⋅ := ⋅)

⟨ML⟩

nonterminal circus-action and circus-schema

syntax
  -circus-action :: ′a => circus-action (⋅)
  -circus-schema :: ′a => circus-schema (⋅)

⟨ML⟩

end

16 Refinement and Simulation

theory Refinement
imports Denotational-Semantics Circus-Syntax
begin

16.1 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**  
\[ Simul S b = \text{extend} (\text{make} (\text{ok} b) (\text{wait} b) (\text{tr} b) (\text{ref} b)) (S (\text{more} b)) \]

**definition**  
\[ Simulation::(\\'\vartheta::\text{ev-eq},\\'\sigma)\text{ action} \Rightarrow (\\'\vartheta, \\'\sigma1)\text{ action} \Rightarrow \text{bool} (- \leq -) \]

**where**  
\[ A \leq S B \equiv \forall a b. (\text{relation-of } B) (a, b) \rightarrow (\text{relation-of } A) (Simul S a, Simul S b) \]

### 16.2 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** Stop-Sim: \[ \text{Stop} \leq S \text{Stop} \]

**lemma** Skip-Sim: \[ \text{Skip} \leq S \text{Skip} \]

**lemma** Chaos-Sim: \[ \text{Chaos} \leq S \text{Chaos} \]

**lemma** Ndet-Sim:  
**assumes** \[ A: P \leq S Q \text{ and } B: P' \leq S Q' \]
**shows** \[ (P \sqcap P') \leq S (Q \sqcap Q') \]

**lemma** Det-Sim:  
**assumes** \[ A: P \leq S Q \text{ and } B: P' \leq S Q' \]
**shows** \[ (P \sqcap P') \leq S (Q \sqcap Q') \]

**lemma** Schema-Sim:  
**assumes** \[ A: \bigwedge a. (\text{Pre sc1}) (S a) \Rightarrow (\text{Pre sc2}) a \]
and \[ B: \bigwedge a b. ([\text{Pre sc1} (S a) ; \text{sc2} (a, b)] \Rightarrow \text{sc1} (S a, S b) \]
**shows** \[ (\text{Schema sc1}) \leq S (\text{Schema sc2}) \]

**lemma** SUb-Sim:  
**assumes** \[ A: \bigwedge a. (\text{Pre sc1}) (S a) \Rightarrow (\text{Pre sc2}) a \]
and \[ B: \bigwedge a b. ([\text{Pre sc1} (S a) ; \text{sc2} (a, b)] \Rightarrow \text{sc1} (S a, S b) \]
and \[ C: P \leq S Q \]
**shows** \[ (\text{state-update-before sc1 } P) \leq S (\text{state-update-before sc2 } Q) \]
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 $C: \land a b. S (ns'2 a b) = ns2 (S a) (S b)$  
and $D: \land a b. S (ns'1 a b) = ns1 (S a) (S b)$  
shows $(P \parallel ns1 \mid cs \parallel ns2 \parallel P') \preceq S (Q \parallel ns'1 \mid cs \parallel ns'2 \parallel Q')$

(proof)

lemma **Assign-Sim**:  
assumes $A: \land A. vy A = vx (S A)$  
and $B: \land ff A. (S (y-update ff A) = x-update ff (S A))$  
shows $(x := ' vx) \preceq S (y := ' vy)$

(proof)

lemma **Var-Sim**:  
assumes $A: P \preceq S Q$ and $B: \land ff A. (S (snd b) ff A) = (snd a) ff (S A)$  
shows $(\text{Var a } P) \preceq S (\text{Var b } Q)$

(proof)

lemma **Guard-Sim**:  
assumes $A: P \preceq S Q$ and $B: \land A. h A = g (S A)$  
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 $B: \land A. (d A) = c (S A)$  
shows $a'?'c \rightarrow P \preceq S a'?'d \rightarrow Q$

(proof)

lemma **Read1-Sim**:  
assumes $A: P \preceq S Q$ and $B: \land A. (d A) = c (S A)$  
shows $a'?'c'\cdot s \rightarrow P \preceq S a'?'d'\cdot s \rightarrow Q$

(proof)

lemma **Read1S-Sim**:  
assumes $A: P \preceq S Q$ and $B: \land A. (d A) = c (S A)$ and $C: \land A. (s' A) = s (S A)$  
shows $a'?'c'\cdot s' \rightarrow P \preceq S a'?'d'\cdot s' \rightarrow Q$

(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-Siml: 
assumes A: \( \bigwedge X. (X \preceq S Y) \Rightarrow ((P X) \preceq S Q) \) and B: mono \( P \)
shows \( (\text{lfp } P) \preceq S Q \)
(proof)

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

lemma bot-Sim: \( \text{bot} \preceq S \text{bot} \)
(proof)

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

lemma ref-eq: \( ((P :: ('a::ev-eq,'b) action) = Q) = (P \subseteq Q \land Q \subseteq P) \)
(proof)

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

lemma rd-impl: 
assumes A:R \((P \vdash Q) \in \{ p. \text{-is-CSP-process } p \}\)
and B:R \((P' \vdash Q') \in \{ p. \text{-is-CSP-process } p \}\)
and C:\( \bigwedge a b. P (a, b) \Rightarrow P' (a, b) \)
and D:\( \bigwedge a b. Q' (a, b) \Rightarrow Q (a, b) \)
shows \( R (P' \vdash Q') (a, b) \rightarrow R (P \vdash Q) (a::('a::ev-eq,'b) \text{-alpha-rp-scheme}, b) \)
(proof)

end
17 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 \textit{Sim}.

17.1 Process definitions

circus-process FIG =
alphabet = [v::nat, x::nat]
state = [idS::nat set]
channel = [out nat, req, ret nat]
schema Init = idS' = {}

schema Out = \exists a. v' = a \land a \notin idS \land idS' = idS \cup \{v'\}

schema Remove = x \in idS \land idS' = idS - \{x\}

where var v •

\[ \mu X • (((\text{req} \rightarrow (\text{Schema FIG.}\text{Out}))'; \text{out}!'(hd o v) \rightarrow \text{Skip}))
\]

\[ \square (\text{ret}'?x \rightarrow (\text{Schema FIG.}\text{Remove}))'; \text{X}) \]

circus-process DFIG =
alphabet = [v::nat, x::nat]
state = [retidS::nat set, max::nat]
channel = FIG-channels
schema Init = retidS' = \{\} \land max' = 0

schema Out = v' = max \land max' = (max + 1) \land retidS' = retidS - \{v'\}

schema Remove = x < max \land retidS' = retidS \cup \{x\} \land max' = max

where var v •

\[ \mu X • (((\text{req} \rightarrow (\text{Schema DFIG.}\text{Out}))'; \text{out}!'(hd o v) \rightarrow \text{Skip}))
\]

\[ \square (\text{ret}'?x \rightarrow (\text{Schema DFIG.}\text{Remove}))'; \text{X}) \]

definition Sim where
Sim A = FIG-alphabet.make (DFIG-alphabet.v A) (DFIG-alphabet.x A)
\({\{a. a < (DFIG-alphabet.max A) \land a \notin (DFIG-alphabet.retidS A)\}}\)

17.2 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: (Schema FIG.Init) ≤Sim (Schema DFIG.Init)
(proof)
lemma SimOut: (Schema FIG.Out) \preceq\ Sim (Schema DFIG.Out)
⟨proof⟩

lemma SimRemove: (Schema FIG.Remove) \preceq\ Sim (Schema DFIG.Remove)
⟨proof⟩

lemma FIG.FIG \preceq\ Sim DFIG.DFIG
⟨proof⟩

end

References


