Isabelle/Circus

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

May 27, 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

1
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 posses a partial ordering.)

2.1.2 Higher-order logic (HOL)

[7, 1] is a classical logic based on a simple type system. It provides the usual logical connectives like $\land$, $\rightarrow$, $\neg$ as well as the object-logical quantifiers $\forall x \bullet P x$ and $\exists x \bullet P x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \rightarrow \beta$. HOL is centered around extensional equality $= :: \alpha \rightarrow \alpha \rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e.g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed $\lambda$-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} \texttt{upd :: } \alpha \Rightarrow \beta \Rightarrow \alpha \Rightarrow \beta \Rightarrow \alpha \Rightarrow \beta \Rightarrow (\alpha \Rightarrow \beta)  \quad ("_L_ := _M_")

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

The pragma ("_L_ := _M_") for the Isabelle syntax engine introduces the notation \texttt{f(x:=y)} for \texttt{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*{2.2.2 Type definitions.}

Types can be introduced in Isabelle/HOL in different ways. The most general way to safely introduce new types is using the \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 \texttt{ option } = \texttt{None } | \texttt{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:

\begin{verbatim}
    case x of None ⇒... | Some a⇒...
\end{verbatim}

\subsection*{2.2.3 Extensible records.}

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

\texttt{record } T = a::T_1

\begin{verbatim}
  b::T_2
\end{verbatim}

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 \( P \parallel P' \) and \( P \cap P' \)), 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: \( P \parallel n \mid c \mid n' \parallel P' \) means that \( P \) and \( P' \) communicate via the channels mentioned in \( c \); moreover, \( P \) may modify the variables mentioned in \( n \) only, and \( P' \) in \( n' \) only, \( n \) and \( n' \) are disjoint name sets.

Moreover, the language comes with a formal notion of refinement based on a denotational semantics. It follows the failure/divergence semantics \([15]\), (but coined in terms of the UTP \([13]\)) providing a notion of execution trace \( \text{tr} \), refusals \( \text{ref} \), 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 \( (\text{alphabet}, \text{predicate}) \) where the free variables appearing in the predicate are all in the alphabet, e.g. \( \{x, y\}, x > y \). As such, it is very similar to the concept of a schema in Z. In the base theory Isabelle/UTP of this work, we represent alphabetized predicates by sets of (extensible) records, e.g. \( \{A. x \ A > y \ A\} \).

An alphabetized relation is an alphabetized predicate where the alphabet is composed of input (undecorated) and output (dashed) variables. In this
channel req
channel ret, out : ID

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

\( \Delta S \)
\[ v! : ID \]
\[ v! \notin idS \]
\[ idS' = idS \cup \{ v! \} \]

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

Remove
\( \Delta S \)
\[ x? : ID \]
\[ idS' = idS \setminus \{ x? \} \]

- Init ; var v : ID •
  \((\mu X • (req → Out ; out!v → Skip □ ref?x → Remove) ; X))
  \)
end

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

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:

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

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

Some healthiness conditions are defined over \texttt{wait}, \texttt{tr} and \texttt{ref} to ensure that a reactive process satisfies some properties \cite{6} (see Table 2 in \cite{9}).

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

3 Isabelle/Circus

\begin{verbatim}
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
\end{verbatim}

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”, \textit{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. \cite{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 \cite{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 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 \texttt{idS} is defined, this is reflected in Isabelle/Circus by the extension of the process alphabet by this variable, i.e. by the extension of the Isabelle/HOL record:

```
record \( \alpha \) alpha = \( \alpha \) alpha_rp + \texttt{idS} :: ID set
```

This introduces the record type \( \alpha \) alpha that contains the observational variables of a reactive process, plus the variable \texttt{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 ”\texttt{name} \mapsto \texttt{value}”, we need a certain infrastructure to access data in them and to update them. The Isabelle representation as records gives us already
two functions (for each record) "select" and "update". The "select" function returns the value of a given variable name, and the "update" functions updates the value of this variable. Since we may have different HOL types for different variables, a unique definition for select and update cannot be provided. There is an instance of these functions for each variable in the record. The name of the variable is used to distinguish the different instances: for the select function the name is used directly and for the update function the name is used as a prefix e.g. for a variable named "x" the names of the select and update functions are respectively \texttt{x} of type \(\alpha\) and \texttt{x_update}. Since a variable is characterized essentially by these functions, we define a general type (synonym) called \texttt{var} which represents a variable as a pair of its select and update function (in the underlying state \(\sigma\)).

\textbf{types} \((\beta, \sigma)\ \texttt{var} = "(\sigma \Rightarrow \beta) \times ((\beta \Rightarrow \sigma) \Rightarrow \sigma)"

For a given alphabet (record) of type \(\sigma\), \((\beta, \text{the type } \sigma)\texttt{var}\) 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:

\textbf{definition} \texttt{select} :: \((\beta, \sigma)\ \texttt{var} \Rightarrow \sigma \Rightarrow \beta"
where \texttt{select} \(f \equiv (\text{fst } f)\)

\textbf{definition} \texttt{update} :: \((\beta, \sigma)\ \texttt{var} \Rightarrow \beta \Rightarrow \sigma \Rightarrow \sigma"
where \texttt{update} \(f \; v \equiv (\text{snd } f) \; (\lambda \_ \cdot v)\)

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

\textbf{syntax} "\_VAR" :: "id \Rightarrow (\beta, \sigma)\ \texttt{var}" ("VAR \_"")
\textbf{translations} \texttt{VAR }x \Rightarrow (x, \_update\_ name x)

Note that in this syntactic translation rule, \_update\_ name \(x\) stands for the concatenation of the string \texttt{_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 \texttt{x_update} by a default mechanism. On this basis, the assignment notation can be written as usual:

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

and mapped to the \textit{semantics} of the program variable \((x, x\_update)\) together with the universal ASSIGN operator defined later on, in Section 3.3.2.

3.1.2 Updating and accessing local variables.

In \textit{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 \texttt{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:

\begin{verbatim}
definition NS lb gb ≡ x_update (x lb) gb
\end{verbatim}

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:

```plaintext
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.

```plaintext
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:

```plaintext
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 \texttt{action}:

\begin{verbatim}
typedef (Action)
(\alpha :: chan_eq, \sigma) action = \{p::(\alpha, \sigma)relation_rp. is_CSP_process p}\nproof - {...}
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 \texttt{Abs_Action} and \texttt{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 \texttt{Abs_Action(Rep_Action(X))=X} and \texttt{is_CSP_process}\(p \implies\texttt{Rep_Action(Abs_Action(p))=p}\) where \texttt{is_CSP_process} captures the healthiness conditions.

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

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

3.3.1 Basic actions.

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

\texttt{definition}
\texttt{Stop} ≡ \texttt{Abs_Action} (R (true \vdash \lambda (A, A'). tr A' = tr A \land \text{wait A'}))

\textit{Skip} is defined as a reactive design, with a precondition \texttt{true} and a post-condition stating that the system terminates and all the state variables are not changed. We represent this fact by stating that the more field (seen in Section 2.2) is not changed, since this field is mapped to all the state variables. Note that using the more-field is a tribute to our encoding of alphabets by extensible records and stands for all future extensions of the alphabet (e.g. state variables).

\texttt{definition} \texttt{Skip} ≡ \texttt{Abs_Action} (R (true \vdash \lambda (A, A'). tr A' = tr A \land \neg \text{wait A'} \land more A = more A'))

3.3.2 The universal assignment action.

In Section 3.1.1, we described how global and local variables are represented by access- and updates functions introduced by fields in extensible records.
In these terms, the "lifting" to the assignment action in *Circus* processes is straightforward:

**definition**

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

where

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

where

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

where *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 *do_I*. In the *Circus* denotational semantics [13], various forms of prefixing were defined. In our theory, we define one general form, and the other forms are defined as special cases.

**definition**

\[ \text{do}_I c \ x \ P \equiv X \triangleleft \text{wait } o \text{ fst } \triangleright Y \]

where

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

and

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

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

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

**definition**

\[ \text{Prefix } c \ x \ P \ S \equiv \text{Abs_Action}(R \ (\text{true } \vdash Y)) \ ; \ S \]

where

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

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

Different types of communication are considered:

- Inputs: the communication is done over a variable.

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

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

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

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

3.3.4 Hiding.

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

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

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

fun tr_filter::"a::chan_eq list ⇒ a set ⇒ a list" where
  tr_filter [] cs = []
| tr_filter (x#xs) cs = (if (∼ chan-in_set x cs)
     then (x#(tr_filter xs cs))
     else (tr_filter xs cs))
where the \texttt{chan-in-set} function checks if a given channel belongs to a channel set using \texttt{chan_eq} as equality function.

### 3.3.5 Recursion.

To represent the recursion operator “$\mu$” over actions, we use the universal least fix-point operator “$\text{lfp}$” 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 $\text{lfp}$ 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 \texttt{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 \texttt{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. \texttt{FIG.Out} and \texttt{DFIG.Out} in the example of Section 4).

### 4 Using Isabelle/Circus

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

A specification is a sequence of paragraphs. Each paragraph may be a declaration of alphabet, state, channels, name sets, channel sets, schema expressions or actions. The main action is introduced by the keyword where. Below, we illustrate how to use the environment to write a Circus specification using the FIG process example presented in Figure 1.

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

Each line of the specification is translated into the corresponding semantic operator given in Section 3.3. We describe below the result of executing each command of FIG:

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

- where introduces the main action as in action command (in the example the main action is FIG.FIG of type (FIG_channels, FIG_alphabet)action).

4.2 Relational and Functional Refinement in Circus

The main goal of Isabelle/Circus is to provide a proof environment for Circus processes. The “shallow-embedding” of Circus and UTP in Isabelle/HOL offers the possibility to reuse proof procedures, infrastructure and theorem libraries already existing in Isabelle/HOL. Moreover, once a process specification is encoded and parsed in Isabelle/Circus, proofs of, e.g., refinement properties can be developed using the ISAR language for structured proofs.

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

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

where \( A_1 \) and \( A_2 \) are Circus actions, \( \sqsubseteq_c \) and \( \sqsubseteq_{\text{utp}} \) stands respectively for refinement relation on Circus actions and on UTP predicate.

This definition assumes that the actions \( A_1 \) and \( A_2 \) share the same alphabet (binding) and the same channels. In general, refinement involves an important data evolution and growth. The data refinement is defined in [16, 5] by backwards and forwards simulations. In this paper, we restrict ourselves to a special case, the so-called functional backwards simulation. This refers to the fact that the abstraction relation \( R \) that relates concrete and abstract actions is just a function:

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

where \( A_1 \) and \( A_2 \) are Circus actions and \( R \) is a function mapping the corresponding \( A_1 \) alphabet to the \( A_2 \) alphabet.

4.3 Refinement Proofs

We can use the definition of simulation to transform the proof of refinement to a simple proof of implication by unfolding the operators in terms of their underlying relational semantics. The problem with this approach is that the size of proofs will grow exponentially with the size of the processes. To avoid this problem, some general refinement laws were defined in [5] to deal with the refinement of Circus actions at operators level and not at UTP level. We introduced and proved a subset of these laws in our environment (see Table 1).
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.

\[
\text{circusprocess DFIG =}
\text{alphabet = [w::nat, y::nat]}
\text{state = [retidS::nat set, max::nat]}
\text{schema Init = retidS' = {} } \land \text{max' = 0}
\text{schema Out = w' = max } \land \text{max' = max+1 } \land \text{retidS' = retidS - {max}}
\text{schema Remove = y < max } \land \text{y \notin retidS } \land \text{retidS' = retidS } \cup \{y\}
\land \text{max' = max}
\text{where var w \cdot Schema Init; (\mu X \cdot (req -> Schema Out; out!w -> Skip))}
\text{ □ (ret?y -> Schema Remove); X)}
\]
We provide the proof of refinement of FIG by DFIG just instantiating the simulation function \( R \) by the following abstraction function, that maps the underlying concrete states to abstract states:

\[
\text{definition } \text{Sim } A = \text{FIG}_\text{alphabet}.\text{make} (w A) (y A)
\]

\[
\langle \{ a. a < (\text{max } A) \land a \notin (\text{retidS } A) \} \rangle
\]

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

To prove that DFIG is a refinement of FIG one must prove that the main action \( \text{DFIG}.\text{DFIG} \) refines the main action \( \text{FIG}.\text{FIG} \). 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:

\[
\begin{align*}
\text{theorem } "\text{FIG}.\text{FIG} \preceq \text{Sim } \text{DFIG}.\text{DFIG}" \\
\text{apply } (\text{auto } \text{simp: } \text{DFIG}.\text{DFIG}_\text{def } \text{FIG}_\text{def } \text{mono}_\text{Seq} \\
\text{intro!: } \text{VarI } \text{SeqI } \text{MuI } \text{DetI } \text{SyncI } \text{InpI } \text{OutI } \text{SkipI}) \\
\text{apply } (\text{simp} \text{all add: } \text{SimRemove } \text{SimOut } \text{SimInit } \text{Sim_def}) \\
\text{done}
\end{align*}
\]

First, the definitions of \( \text{FIG}.\text{FIG} \) and \( \text{DFIG}.\text{DFIG} \) are simplified and the defined refinement laws are used by the \text{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: \text{SimInit}, \text{SimOut} and \text{SimRemove} give proofs of simulation for the schema \text{Init}, \text{Out} and \text{Remove}.

5 Conclusions

We have shown for the language Circus, which combines data-oriented modeling in the style of Z and behavioral modeling in the style of CSP, a semantics in form of a shallow embedding in Isabelle/HOL. In particular, by representing the somewhat non-standard concept of the alphabet in UTP in form of extensible records in HOL, we achieved a fairly compact, typed presentation of the language. In contrast to previous work based on some deep embedding [19], this shallow embedding allows arbitrary (higher-order) HOL-types for channels, events, and state-variables, such as, e.g., sets of relations etc. Besides, systematic renaming of local variables is avoided by compiling them essentially to global variables using a stack of variable instances. The necessary proofs for showing that the definitions are consistent — i.e. satisfy altogether \text{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

theory Var
imports Main
begin
declare implies-True-equals[simp] False-implies-equals[simp]

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 'α alphabet = 'α

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

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

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

definition not::'α predicate ⇒ 'α predicate (¬ [40] 40)
where ¬ P ≡ λA. ¬ (P A)

definition conj::'α predicate ⇒ 'α predicate ⇒ 'α predicate (infixr ∧ 35)
where P ∧ Q ≡ λA. P A ∧ Q A

definition disj::'α predicate ⇒ 'α predicate ⇒ 'α predicate (infixr ∨ 30)
where P ∨ Q ≡ λA. P A ∨ Q A

definition impl::'α predicate ⇒ 'α predicate ⇒ 'α predicate (infixr −→ 25)
where P −→ Q ≡ λA. P A −→ Q A

definition iff::'α predicate ⇒ 'α predicate ⇒ 'α predicate (infixr ←→ 25)
where P ←→ Q ≡ λA. P A ←→ Q A

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

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

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

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

definition cond::'α relation ⇒ 'α condition ⇒ 'α relation ⇒ 'α relation
where (P ▷ b ▷ Q) ≡ (b ∧ P) ∨ ((¬ b) ∧ Q)

definition comp::(('α × 'β) ⇒ bool) ⇒ (('β × 'γ) ⇒ bool) ⇒ ('α × '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::\(\alpha\) predicate \(\Rightarrow\) bool (\([-]\))
where \([\ P \ ] \equiv \forall \ A. \ P \ A\)

abbreviation (input) ndet::\(\alpha\) relation \(\Rightarrow\) \(\alpha\) relation ((\(- \cap -\)))
where \(P \cap Q \equiv P \lor Q\)

abbreviation (input) join::\(\alpha\) relation \(\Rightarrow\) \(\alpha\) relation ((\(- \cup -\)))
where \(P \cup Q \equiv P \land Q\)

abbreviation (input) ndetS::\(\alpha\) relation set \(\Rightarrow\) \(\alpha\) relation
where \(d S \equiv \lambda A. A \in \bigcup\{\{p. P p\} \mid P. P \in S\}\)

abbreviation (input) conjS::\(\alpha\) relation set \(\Rightarrow\) \(\alpha\) relation
where \(\land S \equiv \lambda A. A \in \bigcap\{\{p. P p\} \mid P. P \in S\}\)

abbreviation (input) skip-r::\(\alpha\) relation
where \(\Pi r \equiv true\)

abbreviation (input) Bot::\(\alpha\) relation
where \(Bot \equiv true\)

abbreviation (input) Top::\(\alpha\) relation
where \(Top \equiv false\)

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: true \(x\) by (simp add: utp-defs)
lemma false-elim: false \(x\) \(\Rightarrow\) \(C\) by (simp add: utp-defs)
lemma true-elim: true \(x\) \(\Rightarrow\) \(C\) \(\Rightarrow\) \(C\) by (simp add: utp-defs)

lemma not-intro: \(P x \Rightarrow false x\) \(\Rightarrow\) \((\neg P)\) \(x\) by (auto simp add: utp-defs)
lemma not-elim: \((\neg P)\) \(x\) \(\Rightarrow\) \(P x\) \(\Rightarrow\) \(C\) by (auto simp add: utp-defs)
lemma not-dest: \((\neg P)\) \(x\) \(\Rightarrow\) \(\neg P x\) by (auto simp add: utp-defs)

lemma conj-intro: \(P x \Rightarrow Q x\) \(\Rightarrow\) \((P \land Q)\) \(x\) by (auto simp add: utp-defs)
lemma conj-elim: \((P \land Q)\) \(x\) \(\Rightarrow\) \((P x \Rightarrow Q x \Rightarrow C)\) \(\Rightarrow\) \(C\) by (auto simp add: utp-defs)
lemma disj-introC: $(\neg Q \Rightarrow P) \Rightarrow (P \vee Q) \Rightarrow P$ by (auto simp add: utp-defs)
lemma disj-elim: $(P \vee Q) \Rightarrow (P \Rightarrow C) \Rightarrow (Q \Rightarrow C) \Rightarrow C$ by (auto simp add: utp-defs)

lemma impl-intro: $(P \Rightarrow Q) \Rightarrow (P \Rightarrow Q) \Rightarrow (P \rightarrow Q)$ by (auto simp add: utp-defs)
lemma impl-elimC: $(P \rightarrow Q) \Rightarrow (\neg P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R$ by (auto simp add: utp-defs)

lemma iff-intro: $(P \Rightarrow Q) \Rightarrow (Q \Rightarrow P) \Rightarrow (P \leftrightarrow Q)$ by (auto simp add: utp-defs)
lemma iff-elimC: $(P \leftrightarrow Q) \Rightarrow (P \Rightarrow Q \Rightarrow R) \Rightarrow (\neg P \Rightarrow \neg Q \Rightarrow R) \Rightarrow R$ by (auto simp add: utp-defs)

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

lemma split-cond:
\[ A \left( ((P \triangleleft b \triangleright Q) \ x) \right) = ((b \ x \rightarrow A \ (P \ x)) \land (\neg b \ x \rightarrow A \ (Q \ x))) \]
by (cases b x) (auto simp add: utp-defs)

lemma split-cond-asm:
\[ A \left( ((P \triangleleft b \triangleright Q) \ x) \right) = \neg (\neg (b \ x \land \neg A \ (P \ x)) \lor (\neg b \ x \land \neg A \ (Q \ x))) \]
by (cases b x) (auto simp add: utp-defs)

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

8.2.2 Misc lemmas

lemma cond-idem: \( (P \triangleleft b \triangleright P) = P \)
by (rule ext) (auto split: cond-splits)

lemma cond-symm: \( (P \triangleleft b \triangleright Q) = (Q \triangleleft \neg b \triangleright P) \)
by (rule ext) (auto split: cond-splits)

lemma cond-assoc: \( ((P \triangleleft b \triangleright Q) \triangleleft c \triangleright R) = (P \triangleleft b \land c \triangleright (Q \triangleleft c \triangleright R)) \)
by (rule ext) (auto split: cond-splits)

lemma cond-distr: \( (P \triangleleft b \triangleright (Q \triangleleft c \triangleright R)) = ((P \triangleleft b \triangleright Q) \triangleleft c \triangleright (P \triangleleft b \triangleright R)) \)
by (rule ext) (auto split: cond-splits)

lemma cond-unit-T:(P \triangleleft true \triangleright Q) = P
by (rule ext) (auto split: cond-splits)

lemma cond-unit-F:(P \triangleleft false \triangleright Q) = Q
by (rule ext) (auto split: cond-splits)

lemma cond-L6: \( (P \triangleleft b \triangleright (Q \triangleleft b \triangleright R)) = (P \triangleleft b \triangleright R) \)
by (rule ext) (auto split: cond-splits)

lemma cond-L7: \( (P \triangleleft b \triangleright (P \triangleleft c \triangleright Q)) = (P \triangleleft b \lor c \triangleright Q) \)
by (rule ext) (auto split: cond-splits)

lemma cond-and-distr: \( ((P \land Q) \triangleleft b \triangleright (R \land S)) = ((P \triangleleft b \triangleright R) \land (Q \triangleleft b \triangleright S)) \)
by (rule ext) (auto split: cond-splits)

lemma cond-or-distr: \( ((P \lor Q) \triangleleft b \triangleright (R \lor S)) = ((P \triangleleft b \triangleright R) \lor (Q \triangleleft b \triangleright S)) \)
by (rule ext) (auto split: cond-splits)
lemma cond-imp-distr:
\((P \rightarrow Q) \triangleleft b \triangleright (R \rightarrow S)\) = \(((P \triangleleft b \triangleright R) \rightarrow (Q \triangleleft b \triangleright S))
by (rule ext) (auto split: cond-splits)

lemma cond-eq-distr:
\((P \leftrightarrow Q) \triangleleft b \triangleright (R \leftrightarrow S)\) = \(((P \triangleleft b \triangleright R) \leftrightarrow (Q \triangleleft b \triangleright S))
by (rule ext) (auto split: cond-splits)

lemma comp-assoc: \((P ; (Q ; ; R)) = ((P ; ; Q) ; ; R))
by (rule ext) blast

lemma conj-comp: \((\Lambda \ a \ b \ c. P (a, b) = P (a, c)) \implies (P \land (Q ; ; R)) = ((P \land Q) ; ; R))
by (rule ext) blast

lemma comp-cond-left-distr:
assumes \((x y z. b(x, y) = b(x, z))\)
shows \((P \triangleleft b \triangleright Q) ; ; R) = ((P ; ; R) \triangleleft b \triangleright (Q ; ; R))
using assms by (auto simp: fun-eq-iff utp-defs)

lemma ndet-symm: \((P::'a relation) \sqcap Q = Q \sqcap P)
by (rule ext) blast

lemma ndet-assoc: \(P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R)
by (rule ext) blast

lemma ndet-idemp: \(P \sqcap P = P)
by (rule ext) blast

lemma ndet-distr: \(P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)
by (rule ext) blast

lemma cond-ndet-distr: \((P \triangleleft b \triangleright (Q \sqcap R)) = ((P \triangleleft b \triangleright Q) \sqcap (P \triangleleft b \triangleright R))
by (rule ext) (auto split: cond-splits)

lemma ndet-cond-distr: \((P \sqcap (Q \triangleleft b \triangleright R)) = ((P \sqcap Q) \triangleleft b \triangleright (P \sqcap R))
by (rule ext) (auto split: cond-splits)

lemma comp-cond-ldistr: \((P \sqcap Q) ; ; R) = ((P ; ; R) \sqcap (Q ; ; R))
by (auto simp: fun-eq-iff utp-defs)

lemma comp-ndet-r-distr: \((P ; ; (Q \sqcap R)) = ((P ; ; Q) \sqcap (P ; ; R))
by (auto simp: fun-eq-iff utp-defs)

lemma l2-5-1-A: \(\forall X \in S. [X \rightarrow (\Pi \ S)]\)
by blast

lemma l2-5-1-B: \(\forall X \in S. [X \rightarrow P]\) \rightarrow [\Pi \ S] \rightarrow P]
lemma l2-5-1: \[\left[\bigcap S\right] \rightarrow P] \leftrightarrow (\forall X \in S. [X \rightarrow P])
by blast

lemma empty-disj: \[\bigcap \{\}\right) = Top
by (rule ext) blast

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

lemma empty-conj: \[\bigcup \{\}\right) = Bot
by (rule ext) blast

lemma l2-5-2: ((\bigcap S) \sqcup Q) = (\bigcap \{P \sqcup Q \mid P \in S\})
by (rule ext) blast

lemma l2-5-3: ((\bigcap S) ; ; Q) = (\bigcap \{Q ; ; Q \mid P \in S\})
by (rule ext) blast

lemma l2-5-4: ((\bigcap S)) ; ; Q) = (\bigcap \{Q ; ; Q \mid P \in S\})
by (rule ext) blast

lemma all-idem : (\forall b. (\forall a. P a) = (\forall a. P a)
by (simp add: all-def)

lemma comp-unit-R [simp]: (P ; ; P \rightarrow r) = P
by (auto simp: fun-eq-iff utp-defs)

lemma comp-unit-L [simp]: (P \rightarrow r ; ; P) = P
by (auto simp: fun-eq-iff utp-defs)

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

lemma not-cond: (\neg(P \leftarrow b \supset Q)) = (\neg(P \leftarrow b \supset (\neg Q))
by (rule ext) (auto split: cond-splits)

lemma cond-cond-not-distr:
((P \leftarrow b \supset Q) \land \neg(R \leftarrow b \supset S)) = ((P \land \neg R) \leftarrow b \supset (Q \land \neg S))
by (rule ext) (auto split: cond-splits)

lemma imp-cond-distr: (R \rightarrow (P \leftarrow b \supset Q)) = ((R \rightarrow P) \leftarrow b \supset (R \rightarrow Q))
by (rule ext) (auto split: cond-splits)

lemma cond-imp-dist: ((P \leftarrow b \supset Q) \rightarrow R) = ((P \rightarrow R) \leftarrow b \supset (Q \rightarrow R))
by (rule ext) (auto split: cond-splits)

30
\[\text{lemma cond-conj-distr}: ((P \triangleleft b \triangleright Q) \land R) = ((P \land R) \triangleleft b \triangleright (Q \land R))
\]
by (rule ext) (auto split: cond-splits)

\[\text{lemma cond-disj-distr}: ((P \triangleleft b \triangleright Q) \lor R) = ((P \lor R) \triangleleft b \triangleright (Q \lor R))\]
by (rule ext) (auto split: cond-splits)

\[\text{lemma cond-know-b}: (b \land (P \triangleleft b \triangleright Q)) = (b \land P)\]
by (rule ext) (auto split: cond-splits)

\[\text{lemma cond-know-nb}: (\neg (b)) \land (P \triangleleft b \triangleright Q)) = ((\neg (b)) \land Q)\]
by (rule ext) (auto split: cond-splits)

\[\text{lemma cond-ass-if}: (P \triangleleft b \triangleright Q)) = (((b) \land P \triangleleft b \triangleright Q))\]
by (rule ext) (auto split: cond-splits)

\[\text{lemma cond-ass-else}: (P \triangleleft b \triangleright Q) = (P \triangleleft b \triangleright ((\neg b) \land Q))\]
by (rule ext) (auto split: cond-splits)

\[\text{lemma not-true-eq-false}: (\neg \text{true}) = \text{false}\]
by (rule ext) blast

\[\text{lemma not-false-eq-true}: (\neg \text{false}) = \text{true}\]
by (rule ext) blast

\[\text{lemma conj-idem}: ((P::\alpha \text{ predicate}) \land P) = P\]
by (rule ext) blast

\[\text{lemma disj-idem}: ((P::\alpha \text{ predicate}) \lor P) = P\]
by (rule ext) blast

\[\text{lemma conj-comm}: ((P::\alpha \text{ predicate}) \land Q) = (Q \land P)\]
by (rule ext) blast

\[\text{lemma disj-comm}: ((P::\alpha \text{ predicate}) \lor Q) = (Q \lor P)\]
by (rule ext) blast

\[\text{lemma conj-subst}: P = R \implies ((P::\alpha \text{ predicate}) \land Q) = (R \land Q)\]
by (rule ext) blast

\[\text{lemma disj-subst}: P = R \implies ((P::\alpha \text{ predicate}) \lor Q) = (R \lor Q)\]
by (rule ext) blast

\[\text{lemma conj-assoc}: ((P::\alpha \text{ predicate}) \land Q) \land S) = (P \land (Q \land S))\]
by (rule ext) blast

\[\text{lemma disj-assoc}: ((P::\alpha \text{ predicate}) \lor Q) \lor S) = (P \lor (Q \lor S))\]
by (rule ext) blast
lemma conj-disj-abs: \((P::'a\ predicate) \land (P \lor Q)\) = \(P\)
by (rule ext) blast

lemma disj-conj-abs: \((P::'a\ predicate) \lor (P \land Q)\) = \(P\)
by (rule ext) blast

lemma conj-disj-distr: \(((P::'a\ predicate) \land (Q \lor R))\) = \(((P \land Q) \lor (P \land R))\)
by (rule ext) blast

lemma disj-conj-distr: \(((P::'a\ predicate) \lor (Q \land R))\) = \(((P \lor Q) \land (P \lor R))\)
by (rule ext) blast

lemma true-conj-id: \((P \land true)\) = \(P\)
by (rule ext) blast

lemma true-disj-zero: \((P \lor true)\) = \(true\)
by (rule ext) blast

lemma true-conj-zero: \((P \land false)\) = \(false\)
by (rule ext) blast

lemma true-disj-id: \((P \lor false)\) = \(P\)
by (rule ext) blast

lemma imp-vacuous: \((false \rightarrow u)\) = \(true\)
by (rule ext) blast

lemma p-and-not-p: \((P \land \neg P)\) = \(false\)
by (rule ext) blast

lemma p-or-not-p: \((P \lor \neg P)\) = \(true\)
by (rule ext) blast

lemma double-negation: \((\neg \neg (P::'a\ predicate))\) = \(P\)
by (rule ext) blast

lemma not-conj-deMorgans: \((\neg ((P::'a\ predicate) \land Q))\) = \(((\neg P) \lor (\neg Q))\)
by (rule ext) blast

lemma not-disj-deMorgans: \((\neg ((P::'a\ predicate) \lor Q))\) = \(((\neg P) \land (\neg Q))\)
by (rule ext) blast

lemma p-imp-p: \((P \rightarrow P)\) = \(true\)
by (rule ext) blast

lemma imp-imp: \(((P::'a\ predicate) \rightarrow (Q \rightarrow R))\) = \(((P \land Q) \rightarrow R)\)
by (rule ext) blast

lemma imp-trans: \((P \rightarrow Q) \land (Q \rightarrow R) \rightarrow P \rightarrow R) = \text{true}\nby (rule ext) blast

lemma p-equiv-p: \((P \leftrightarrow P) = \text{true}\nby (rule ext) blast

lemma equiv-eq: \(((\neg \neg (\alpha \text{ predicate}) \land Q) \lor (\neg P \land \neg Q)) = \text{true}) \leftrightarrow (P = Q)\nby (auto simp add: fun-eq-iff utp-defs)

lemma equiv-eq1: \(((P \leftrightarrow Q) = \text{true}) \leftrightarrow (P = Q)\nby (auto simp add: fun-eq-iff utp-defs)

lemma cond-subst: \(b = c \implies (P \triangleleft b \triangleright Q) = (P \triangleleft c \triangleright Q)\nby simp

lemma ex-disj-distr: \((\exists x. P x) \lor (\exists x. Q x)) = (\exists x. (P x \lor Q x))\nby (rule ext) blast

lemma all-disj-distr: \((\forall x. P x) \lor (\forall x. Q x)) = (\forall x. (P x \lor Q x))\nby (rule ext) blast

lemma all-conj-distr: \((\forall x. P x) \land (\forall x. Q x)) = (\forall x. (P x \land Q x))\nby (rule ext) blast

lemma all-triv: \((\forall x. P) = P\nby (rule ext) blast

lemma closure-true: [\text{true}]\nby blast

lemma closure-p-eq-true: \([P] \leftrightarrow (P = \text{true})\nby (simp add: fun-eq-iff utp-defs)

lemma closure-equiv-eq: \([P \leftrightarrow Q] \leftrightarrow (P = Q)\nby (simp add: fun-eq-iff utp-defs)

lemma closure-conj-distr: \([P] \land [Q]) = [P \land Q]\nby blast

lemma closure-imp-distr: \([P \rightarrow Q] \rightarrow [P] \rightarrow [Q]\nby blast

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

lemma true-imp[simp]: \((\text{true} \rightarrow P) = P\nby blast
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 H1, H2, H3 and H4.

record alpha-d = ok::bool

type-synonym 'α alphabet-d = 'α alpha-d-scheme alphabet

type-synonym 'α relation-d = 'α alphabet-d relation

definition design::'α relation-d ⇒ 'α relation-d ⇒ 'α relation-d ('(⊢ ·)')
where (P ⊢ Q) ≡ λ (A, A') . (ok A ∧ P (A,A')) → (ok A' ∧ Q (A,A'))

definition skip-d :: 'α relation-d (Πd)
where Πd ≡ (true ⊢ Πr)

definition J
where J ≡ λ (A, A') . (ok A → ok A') ∧ more A = more A'

type-synonym 'α Healthiness-condition = 'α relation ⇒ 'α relation

definition Healthy::'α relation ⇒ 'α Healthiness-condition ⇒ bool (- is - healthy)
where P is H healthy ≡ (P = H P)

lemma Healthy-def': P is H healthy = (H P = P)
  unfolding Healthy-def by auto

definition H1::('α alphabet-d) Healthiness-condition
where H1 (P) ≡ (ok o fst →→ P)

definition H2::('α alphabet-d) Healthiness-condition
where H2 (P) ≡ P ; ; J
definition $H3::(\alpha \text{ alphabet-d})$ Healthiness-condition
where $H3 (P) \equiv P ; ; \Pi d$

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

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

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

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

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

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

9.2 Proofs

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

lemma t-comp-lz-d: (true; ; (P ; Q)) = true
  apply (auto simp: fun-eq-iff design-defs)
  apply (rule-tac b = b (ok:=False) in comp-intro, auto)
done

lemma pi-comp-left-unit: (\Pi d; ; (P ; Q)) = (P ; Q)
by (auto simp: fun-eq-iff design-defs)

theorem t3-1-4-2:
((P1 ; Q1) \triangleleft b \triangleright (P2 ; Q2)) = ((P1 \triangleleft b \triangleright P2) ; (Q1 \triangleleft b \triangleright Q2))
by (auto simp: fun-eq-iff design-defs split: cond-splits)

lemma conv-conj-distr: $\sigma t (P \land Q) = (\sigma t P \land \sigma t Q)$
by (auto simp: design-defs fun-eq-iff)

lemma conv-disj-distr: $\sigma t (P \lor Q) = (\sigma t P \lor \sigma t Q)$
by (auto simp: design-defs fun-eq-iff)

lemma conv-imp-distr: $\sigma t (P \rightarrow Q) = ((\sigma t P) \rightarrow \sigma t Q)$
by (auto simp: design-defs fun-eq-iff)

lemma conv-not-distr: $\sigma t (\neg P) = (\neg(\sigma t P))$
by (auto simp: design-defs fun-eq-iff)
lemma \( \text{div-conj-distr} \): \( \sigma f (P \land Q) = (\sigma f P \land \sigma f Q) \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{div-disj-distr} \): \( \sigma f (P \lor Q) = (\sigma f P \lor \sigma f Q) \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{div-imp-distr} \): \( \sigma f (P \longrightarrow Q) = ((\sigma f P) \longrightarrow \sigma f Q) \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{div-not-distr} \): \( \sigma f (\neg P) = (\neg (\sigma f P)) \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{ok-conv} \): \( \sigma t \text{ OKAY} = \text{OKAY} \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{ok-div} \): \( \sigma f \text{ OKAY} = \text{OKAY} \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{ok'}-conv \): \( \sigma t \text{ OKAY'} = \text{true} \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{ok'}-div \): \( \sigma f \text{ OKAY'} = \text{false} \)
by (auto simp: design-defs fun-eq-iff)

lemma \( \text{H2-J-1} \):
assumes \( A: P \text{ is H2 healthy} \)
sows \( ((\lambda (A, A'). (P(A, A'[[ok := False]]) \longrightarrow P(A, A'[[ok := True]])))) \)
using \( A \) by (auto simp: design-defs fun-eq-iff)

lemma \( \text{H2-J-2-a} \): \( P(a,b) \longrightarrow (P ;; J)(a,b) \)
unfolding \( J\text{-def} \) by auto

lemma \( \text{ok-or-not-ok} \): \( [P(a, b[[ok := True]]) ;; P(a, b[[ok := False]])] \Longrightarrow P(a, b) \)
apply (case-tac ok b)
apply (subgoa-tac b[[ok:=True]] = b)
apply (simp-all)
apply (subgoa-tac b[[ok:=False]] = b)
apply (simp-all)
done

lemma \( \text{H2-J-2-b} \):
assumes \( A: ([\lambda (A, A'). (P(A, A'[[ok := False]]) \longrightarrow P(A, A'[[ok := True]]))] \)
and \( B: (P ;; J)(a,b) \)
sows \( P(a,b) \)
using \( B \)
apply (auto simp: design-defs fun-eq-iff)
apply (case-tac ok b)
apply (subgoa-tac b = ba[[ok:=True]], auto intro: A[simplified, rule-format])
apply (rule-tac s=ba and t=ba\{ok:=False\} in subst, simp-all)
apply (subgoal-tac b = ba, simp-all)
apply (case-tac ok ba)
apply (subgoal-tac b = ba, simp-all)
apply (case-tac ok ba)
apply (subgoal-tac b = ba\{ok:=False\}, auto intro!: A[simplified, rule-format])
apply (rule-tac s=ba and t=ba\{ok:=False\} in subst, simp-all)
done

lemma H2-J-2 :
  assumes A: \[\lambda (A, A'). (P(A, A'\{ok := False\}) \longrightarrow P(A, A'\{ok := True\}))]]
  shows P is H2 healthy
apply (auto simp add: H2-def Healthy-def fun-eq-iff)
apply (simp add: H2-J-2-a)
apply (rule H2-J-2-b [OF A])
apply auto
done

lemma H2-J:
  \[\lambda (A, A'). P(A, A'\{ok := False\}) \longrightarrow P(A, A'\{ok := True\})] = P is H2 healthy
using H2-J-1 H2-J-2 by blast

lemma design-eq1: (P ⊢ Q) = (P ⊢ P ∧ Q)
by (rule ext) (auto simp: design-defs)

lemma H1-idem: H1 o H1 = H1
by (auto simp: design-defs fun-eq-iff)

lemma H1-idem2: (H1 (H1 P)) = (H1 P)
by (simp add: H1-idem[simplified fun-eq-iff Fun.comp-def, rule-format] fun-eq-iff)

lemma H2-idem: H2 o H2 = H2
by (auto simp: design-defs fun-eq-iff)

lemma H2-idem2: (H2 (H2 P)) = (H2 P)
by (simp add: H2-idem[simplified fun-eq-iff Fun.comp-def, rule-format] fun-eq-iff)

lemma H1-H2-commute: H1 o H2 = H2 o H1
by (auto simp: design-defs fun-eq-iff split: cond-splits)

lemma H1-H2-commute2: H1 (H2 P) = H2 (H1 P)

lemma alpha-d-eqD: r = r' \implies ok r = ok r' ∧ alpha-d.more r = alpha-d.more r'
by (auto simp: alpha-d.equality)

lemma design-H1: (P ⊢ Q) is H1 healthy
by (auto simp: design-defs fun-eq-iff)
10 Reactive processes

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

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

10.1 Preliminaries

**type-synonym** 'α trace = 'α list

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

**instantiation** list :: (type) minus

begin

**definition** list-minus : l1 − l2 ≡ the (list-diff l1 l2)

**instance** ..

end

**lemma** list-diff-empty [simp]: the (list-diff l []) = l

by (cases l) auto

**lemma** prefix-diff-empty [simp]: l − [] = l

by (induct l) (auto simp: list-minus)

**lemma** prefix-diff-eq [simp]: l − l = []
lemma prefix-diff [simp]: \((l@t) - l = t\)
by (induct l) (auto simp: list-minus)

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

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

lemma prefix-diff1 [simp]: \(((l@m)@t) - (l@m) = t\)
by (rule prefix-diff)

lemma prefix-diff2 [simp]: \((l@m@t) - (l@m) = t\)
apply (simp only: append-assoc [symmetric])
apply (rule prefix-diff1)
done

lemma prefix-diff3 [simp]: \((l@m) - (l@t) = (m - t)\)
by (induct l, auto simp: list-minus)

lemma prefix-diff4 [simp]: \((a#m) - (a#t) = (m - t)\)
by (auto simp: list-minus)

class ev-eq =
  fixes ev-eq :: 'a ⇒ 'a ⇒ bool
  assumes refl: ev-eq a a
  assumes comm: ev-eq a b = ev-eq b a

definition filter-chan-set a cs = (¬(∃e ∈ cs. ev-eq a e))

lemma in-imp-not-fcs:
  \(x ∈ S \Rightarrow ¬\ \text{filter-chan-set} \ x \ S\)
apply (auto simp: filter-chan-set-def)
apply (rule_tac bexI, auto simp: refl)
done

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))

lemma tr-filter-conc: \((tr-filter (a@b) cs) = ((tr-filter a cs) @ (tr-filter b cs))\)
by (induct a, auto)

lemma filter-chan-set-hd-tr-filter:
tr-filter l cs ≠ [] --- \implies \text{filter-chan-set} (hd (tr-filter l cs)) cs
by (induct l, auto)

\textbf{lemma \textit{tr-filter-conc-eq1}}:
(a\@b = (tr-filter (a\@c) cs)) \implies (b = (tr-filter c cs))
apply (induct a, auto)
apply (case-tac \text{tr-filter} (a2 \@ c) cs = [], simp-all)
apply (drule \text{filter-chan-set-hd-tr-filter}\{rule-format\})
apply (case-tac \text{tr-filter} (a2 \@ c) cs, simp-all)
done

\textbf{lemma \textit{tr-filter-conc-eq2}}:
(a\@b = (tr-filter (a\@c) cs)) \implies (a = (tr-filter a cs))
apply (induct a, auto)
apply (case-tac \text{tr-filter} (a2 \@ c) cs = [], simp-all)
apply (drule \text{filter-chan-set-hd-tr-filter}\{rule-format\})
apply (case-tac \text{tr-filter} (a2 \@ c) cs, simp-all)
apply (drule \text{filter-chan-set-hd-tr-filter}\{rule-format\})
apply (case-tac \text{tr-filter} (a2 \@ c) cs, simp-all)
done

\textbf{lemma \textit{tr-filter-conc-eq}}:
(a\@b = (tr-filter (a\@c) cs)) = (b = (tr-filter c cs) \& a = (tr-filter a cs))
apply (rule, rule)
apply (rule \text{tr-filter-conc-eq1}\{rule-format, af a\}, clarsimp)
apply (rule \text{tr-filter-conc-eq2}\{rule-format, af a b c\}, clarsimp)
apply (clarsimp simp: \text{tr-filter-conc})
done

\textbf{lemma \textit{tr-filter-conc-eq3}}:
(b = (tr-filter (a\@c) cs)) = (\exists b1 b2, b=b1\@b2 \& b2 = (tr-filter c cs) \& b1 = (tr-filter a cs))
by (rule, auto simp: \text{tr-filter-conc})

\textbf{lemma \textit{tr-filter-un}}:
\text{tr-filter} l (s1 \cup s2) = \text{tr-filter} (\text{tr-filter} l s1) s2
by (induct l, auto simp: \text{filter-chan-set-def})

\textbf{instantiation list :: (ev-eq) ev-eq}
begin
fun \text{ev-eq-list} where
  \text{ev-eq-list} [] [] = True
| \text{ev-eq-list} l [] = False
| \text{ev-eq-list} [] l = False
| \text{ev-eq-list} (x#xs) (y#ys) = (if (ev-eq x y) then (ev-eq-list xs ys) else False)
instance
proof
fix a::'a::ev-eq list show ev-eq a a
by (induct a, auto simp: ev-eq-class.refl)
next
fix a b::'a::ev-eq list show ev-eq a b = ev-eq b a
apply (cases a)
apply (cases b, simp-all add: ev-eq-class.comm)
apply (hyps:thin)
apply (induct b, simp-all add: ev-eq-class.comm)
apply (case-tac list = [], simp-all)
apply (case-tac b, simp-all)
apply (atomize)
apply (erule-tac x = hd list in allE)
apply (erule-tac x = tl list in allE)
apply (subst (asm) hd-Cons-tl, simp-all)
done
qed
end

10.2 Definitions

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

lemma list-diff-empty-eq: l1 − l2 = [] ⇒ l2 ≤ l1 ⇒ l1 = l2
by (auto simp: prefixeq-def)

The definitions of reactive process alphabets and healthiness conditions are
given in the following. The healthiness conditions of reactive processes are
declared by $R_1$, $R_2$, $R_3$ 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 Speciftt (_₄') where (P)₄' ≡ spec True True P

41
abbreviation Specifff (-{f}) where (P){f} ¨ spec False False P
abbreviation Specifff (-{f}) where (P){f} ¨ spec True False P
abbreviation Specifff (-{f}) where (P){f} ¨ 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 Πrea
where Πrea  ¨ λ(A, A′). (¬ok A ∧ tr A ≤ tr A′) ∨ (ok A′ ∧ tr A = tr A′ ∨ (wait A = wait A′) ∧ ref A = ref A′ ∧ more A = more A′)
definition R3::((,'ϑ,,'σ)alphabet-rp) Healthiness-condition
where R3 (P)  ¨ (Πrea o wait o fst ⊢ P)
definition R::((,'ϑ,,'σ)alphabet-rp) Healthiness-condition
where R  ¨ R3 o R2 o R1
lemmas rp-defs = R1-def R2-def Πrea-def R3-def R-def spec-def

10.3 Proofs

lemma tr-filter-empty [simp]: tr-filter l {} = l
by (induct l) (auto simp: filter-chan-set-def)

lemma trf-imp-filtercs: [xs = tr-filter ys cs; xs ≠ []] ⇒ filter-chan-set (hd xs) cs
apply (induct xs, auto)
apply (induct ys, auto)
apply (case-tac filter-chan-set a cs, auto)
done

lemma filtercs-imp-trf:
[filter-chan-set x cs; xs = tr-filter ys cs] ⇒ x#xs = tr-filter (x#ys) cs
by (induct xs) auto

lemma alpha-d-more-eqI:
assumes tr r = tr r′ wait r = wait r′ ref r = ref r′ more r = more r′
shows alpha-d more r = alpha-d more r′
using assms by (cases r, cases r′) auto

lemma alpha-d-more-eqE:
assumes alpha-d more r = alpha-d more r′
obtains tr r = tr r′ wait r = wait r′ ref r = ref r′ more r = more r′
using assms by (cases r, cases r') auto

lemma alpha-rp-eqE:
  assumes r = r'
  obtains ok r = ok r' tr r = tr r' wait r = wait r' ref r = ref r' more r = more r'
  using assms by (cases r, cases r') auto

lemma R-idem: R o R = R
  by (auto simp: rp-defs design-defs fun-eq-iff split: cond-splits)

lemma R-idem2: R (R P) = R P
  by (auto simp: rp-defs design-defs fun-eq-iff split: cond-splits)

lemma R1-idem: R1 o R1 = R1
  by (auto simp: rp-defs design-defs)

lemma R1-idem2: R1 (R1 x) = R1 x
  by (auto simp: rp-defs design-defs)

lemma R2-idem: R2 o R2 = R2
  by (auto simp: rp-defs design-defs fun-eq-iff prefixeq-def)

lemma R2-idem2: R2 (R2 x) = R2 x
  by (auto simp: rp-defs design-defs fun-eq-iff prefixeq-def)

lemma R3-idem: R3 o R3 = R3
  by (auto simp: rp-defs design-defs fun-eq-iff split: cond-splits)

lemma R3-idem2: R3 (R3 x) = R3 x
  by (auto simp: R3-idem[simplified Fun.comp-def fun-eq-iff] fun-eq-iff)

lemma R1-R2-commute: (R1 o R2) = (R2 o R1)
  by (auto simp: rp-defs design-defs fun-eq-iff prefixeq-def)

lemma R1-R3-commute: (R1 o R3) = (R3 o R1)
  by (auto simp: rp-defs design-defs fun-eq-iff prefixeq-def split: cond-splits)

lemma R2-R3-commute: R2 o R3 = R3 o R2
  by (auto simp: rp-defs design-defs fun-eq-iff prefixeq-def split: cond-splits elim: prefixE)

lemma R-abs-R1: R o R1 = R
  apply (auto simp: R-def)
  apply (subst R1-idem[symmetric]) back back
  apply (auto)
  done

lemma R-abs-R2: R o R2 = R

43
by (auto simp: rp-defs design-defs fun-eq-iff)

lemma R-abs-R3: \( R \circ R3 = R \)
by (auto simp: rp-defs design-defs fun-eq-iff split: cond-splits elim: prefixE)

lemma R-is-R1:
assumes A: \( P \) is \( R \) healthy
shows \( P \) is \( R1 \) healthy
proof -
  have \( R \ P = P \)
    using assms by (simp-all only: Healthy-def)
  moreover
  have \( (R \ P) \) is \( R1 \) healthy
    by (auto simp add: design-defs rp-defs fun-eq-iff split: cond-splits)
  ultimately show ?thesis by simp
qed

lemma R-is-R2:
assumes A: \( P \) is \( R \) healthy
shows \( P \) is \( R2 \) healthy
proof -
  have \( R \ P = P \)
    using assms by (simp-all only: Healthy-def)
  moreover
  have \( (R \ P) \) is \( R2 \) healthy
    by (auto simp add: design-defs rp-defs fun-eq-iff prefixeq-def prefix-def split: cond-splits)
  ultimately show ?thesis by simp
qed

lemma R-is-R3:
assumes A: \( P \) is \( R \) healthy
shows \( P \) is \( R3 \) healthy
proof -
  have \( R \ P = P \)
    using assms by (simp-all only: Healthy-def)
  moreover
  have \( (R \ P) \) is \( R3 \) healthy
    by (auto simp add: design-defs rp-defs fun-eq-iff split: cond-splits)
  ultimately show ?thesis by simp
qed

lemma R-disj:
assumes A: \( P \) is \( R \) healthy
assumes B: \( Q \) is \( R \) healthy
shows \( (P \lor Q) \) is \( R \) healthy
proof -
  have \( R \ P = P \) and \( R \ Q = Q \)
    using assms by (simp-all only: Healthy-def)

moreover have \(((R\ P) \lor (R\ Q))\) is R healthy
  by (auto simp add: design-defs rp-defs fun-eq-iff split: cond-splits)
ultimately show ?thesis by simp
qed

lemma R-disj2:  \(R\ (P \lor Q) = (R\ P \lor R\ Q)\)
apply (subst R-disj[simplified Healthy-def, where P=R P])
apply (simp-all add: R-idem2)
apply (auto simp: fun-eq-iff rp-defs split: cond-splits)
done

lemma R1-comp:
  assumes P is R1 healthy
  and Q is R1 healthy
  shows \((P; ; Q)\) is R1 healthy
proof –
  have \(R1\ P = P\) and \(R1\ Q = Q\)
  using assms by (simp-all only: Healthy-def)
moreover have \(((R1\ P); ; (R1\ Q))\) is R1 healthy
  by (auto simp add: design-defs rp-defs fun-eq-iff split: cond-splits)
ultimately show ?thesis by simp
qed

lemma R1-comp2:
  assumes A: P is R1 healthy
  assumes B: Q is R1 healthy
  shows \(R1\ (P; ; Q) = ((R1\ P); ; Q)\)
using A B
apply (subst R1-comp[simplified Healthy-def, symmetric])
apply (auto simp: fun-eq-iff rp-defs design-defs)
done

lemma J-is-R1: J is R1 healthy
by (auto simp: rp-defs design-defs fun-eq-iff elim: alpha-d-more-eqE)

lemma J-is-R2: J is R2 healthy
by (auto simp: rp-defs design-defs fun-eq-iff prefix-def prefixeq-def
  elim!: alpha-d-more-eqE intro!: alpha-d-more-eqI)

lemma R1-H2-commute2: R1 (H2 P) = H2 (R1 P)
by (auto simp add: H2-def R1-def J-def fun-eq-iff
  elim!: alpha-d-more-eqE intro!: alpha-d-more-eqI)

lemma R1-H2-commute: R1 o H2 = H2 o R1
by (auto simp: R1-H2-commute2)

lemma R2-H2-commute2: R2 (H2 P) = H2 (R2 P)
apply (auto simp add: fun-eq-iff rp-defs design-defs prefix-def)
apply (rule-tac b=ba(tr := tr a @ tr ba) in comp-intro)
apply (auto simp: fun-eq-iff prefix-def prefixeq-def
  elim!: alpha-d-more-eqE alpha-rp-eqE intro!: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac b=ba(tr := tr a @ tr ba) in comp-intro,
  auto simp: elim: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac b=ba(tr := tr a @ tr ba) in comp-intro,
  auto simp: elim: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac x=zs in extI, auto)+ done

lemma R2-H2-commute: R2 o H2 = H2 o R2
by (auto simp: R2-H2-commute2)

lemma R3-H2-commute2: R3 (H2 P) = H2 (R3 P)
apply (auto simp: fun-eq-iff rp-defs design-defs prefix-def
  elim: alpha-d-more-eqE split: cond-splits)
done

lemma R3-H2-commute: R3 o H2 = H2 o R3
by (auto simp: R3-H2-commute2)

lemma R-join:
  assumes x is R healthy
  and y is R healthy
  shows (x ⊓ y) is R healthy
proof -
  have R x = x and R y = y
  using assms by (simp-all only: Healthy-def)
moreover
  have ((R x) ⊓ (R y)) is R healthy
  by (auto simp add: design-defs rp-defs fun-eq-iff split: cond-splits)
ultimately show ?thesis by simp
qed

lemma R-meet:
  assumes A: x is R healthy
  and B: y is R healthy
  shows (x ⊔ y) is R healthy
proof -
  have R x = x and R y = y
  using assms by (simp-all only: Healthy-def)
moreover
  have ((R x) ⊔ (R y)) is R healthy
  by (auto simp add: design-defs rp-defs fun-eq-iff split: cond-splits)
ultimately show ?thesis by simp
qed
lemma R-H2-commute: \( R \circ H2 = H2 \circ R \)
apply (auto simp add: rp-defs design-defs fun-eq-iff split: cond-splits
            elim: alpha-d-more-eqE)
apply (rule-tac b=ba in comp-intro, auto split: cond-splits
            elim!: alpha-d-more-eqE alpha-rp-eqE intro!: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac s=ba in subst, auto intro!: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac s=ba in subst, auto elim!: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac s=ba in subst, auto elim: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality
            split: cond-splits)
apply (rule-tac s=ba in subst, auto elim: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality)
apply (rule-tac b=ba in subst,
            auto elim: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality
            split: cond-splits)
apply (rule-tac s=ba in subst,
            auto elim: alpha-d-more-eqE alpha-rp-eqE intro: alpha-d-more-eqI alpha-rp.equality)
done

lemma R-H2-commute2: \( R (H2 P) = H2 (R P) \)
by (auto simp: fun-eq-iff R-H2-commute[simplified fun-eq-iff Fun.comp-def])

end

11 CSP processes

theory CSP-Processes
imports Reactive-Processes
begin

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

11.1 Definitions

We introduce here the definitions of the CSP healthiness conditions.

definition CSP1::((′ϑ,′σ) alphabet-rp) Healthiness-condition
where CSP1 \( (P) \equiv P \lor (\lambda(A,A'). \neg ok A \land tr A \leq tr A') \)
definition J-csp
where J-csp \( \equiv \lambda(A,A'). (ok A \rightarrow ok A') \land tr A = tr A' \land wait A = wait A' \land ref A = ref A' \land more A = more A' \)
definition CSP2::((′ϑ,′σ) alphabet-rp) Healthiness-condition
where CSP2 \( (P) \equiv P ; \ J-csp \)
definition is-CSP-process::((′ϑ,′σ) relation-rp ⇒ bool) where is-CSP-process \( P \equiv P \text{ is CSP1 healthy} \land P \text{ is CSP2 healthy} \land P \text{ is R healthy} \)
lemmas csp-defs = CSP1-def J-csp-def CSP2-def is-CSP-process-def

lemma is-CSP-processE1 [elim?]:
  assumes is-CSP-process P
  obtains P is CSP1 healthy P is CSP2 healthy P is R healthy
  using assms unfolding is-CSP-process-def by simp

lemma is-CSP-processE2 [elim?]:
  assumes is-CSP-process P
  obtains CSP1 P = P CSP2 P = P R P = P
  using assms unfolding is-CSP-process-def by (simp add: Healthy-def')

11.2 Proofs

Theorems and lemmas relative to CSP processes are introduced here.

lemma CSP1-CSP2-commute: CSP1 o CSP2 = CSP2 o CSP1
by (simp add: csp-defs fun-eq-iff)

lemma CSP2-is-H2: H2 = CSP2
apply (clarsimp simp add: csp-defs design-defs rp-defs fun-eq-iff)
apply (erule_tac ![] comp-elim)
apply (erule_tac ![] b=ba in comp-intro)
apply (auto elim!: alpha-d-more-eqE intro!: alpha-d-more-eqI)
done

lemma H2-CSP1-commute: H2 o CSP1 = CSP1 o H2
apply (subst CSP2-is-H2[simplified Healthy-def])+
apply (rule CSP1-CSP2-commute[symmetric])
done

lemma H2-CSP1-commute2: H2 (CSP1 P) = CSP1 (H2 P)
by (simp add: H2-CSP1-commute[simplified Fun.comp-def fun-eq-iff, rule-format]
fun-eq-iff)

lemma CSP1-R-commute:
    CSP1 (R P) = R (CSP1 P)
by (auto simp: csp-defs rp-defs fun-eq-iff prefixeq-def split: cond-splits)

lemma CSP2-R-commute:
    CSP2 (R P) = R (CSP2 P)
apply (subst CSP2-is-H2[symmetric])+
apply (rule R-H2-commute2[symmetric])
done

lemma CSP1-idem: CSP1 = CSP1 o CSP1
by (auto simp: csp-defs fun-eq-iff)
lemma CSP2-idem: CSP2 = CSP2 o CSP2
by (auto simp: csp-defs fun-eq-iff)

lemma CSP-is-CSP1:  
  assumes A: is-CSP-process P  
  shows P is CSP1 healthy  
using A by (auto simp: is-CSP-process-def design-defs)

lemma CSP-is-CSP2:  
  assumes A: is-CSP-process P  
  shows P is CSP2 healthy  
using A by (simp add: design-defs prefixeq-def is-CSP-process-def)

lemma CSP-is-R:  
  assumes A: is-CSP-process P  
  shows P is R healthy  
using A by (simp add: design-defs prefixeq-def is-CSP-process-def)

lemma t-or-f-a: P(a, b) ⇒ ((P(a, b(ok := True))) ∨ (P(a, b(ok := False))))
apply (clarsimp simp: csp-defs design-defs rp-defs split: cond-splits elim: prefixE)
apply (case-tac ok ba)
apply (rule-tac t=b(ok := True) and s=ba in subst, simp-all)
by (subgoal-tac b=b(ok := False) and simp-all)

lemma CSP2-ok-a:  
  (CSP2 P)(a, b(ok := True)) ⇒ (P(a, b(ok := True)) ∨ (P(a, b(ok := False))))
apply (clarsimp simp: csp-defs design-defs rp-defs split: cond-splits elim: prefixE)
apply (case-tac ok ba)
apply (rule-tac t=b(ok := True) and s=ba in subst, simp-all)
apply (drule-tac b=b(ok := False) and a=ba in back-subst)
apply (auto intro: alpha-rp.equality)
done

lemma CSP2-ok-b:  
  (P(a, b(ok := True)) ∨ (P(a, b(ok := False)))) ⇒ (CSP2 P)(a, b(ok := True))
by (auto simp: csp-defs design-defs rp-defs)

lemma CSP2-ok:  
  (CSP2 P)(a, b(ok := True)) = (P(a, b(ok := True)) ∨ (P(a, b(ok := False))))
apply (rule iffl)
apply (simp add: CSP2-ok-a)
by (simp add: CSP2-ok-b)

lemma CSP2-notok-a: (CSP2 P)(a, b(ok := False)) ⇒ P(a, b(ok := False))
apply (clarsimp simp: csp-defs design-defs rp-defs)
apply (case-tac ok ba)
apply (rule-tac t=b(ok := True) and s=ba in subst, simp-all)
apply (drule-tac b=b(ok := False) and a=ba in back-subst)
apply (auto intro: alpha-rp.equality)
done
lemma CSP2-notok-b: P(a, b[ok:=False]) \implies (CSP2 P)(a, b[ok:=False]) 
by (auto simp: csp-defs design-defs rp-defs)

lemma CSP2-notok: (CSP2 P)(a, b[ok:=False]) = P(a, b[ok:=False])
apply (rule iffI)
apply (simp add: CSP2-notok-a)
by (simp add: CSP2-notok-b)

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
apply (rule B)
apply (rule disjI2)
apply (insert A)
apply (auto simp add: csp-defs design-defs rp-defs)
done

lemma disj-CSP1:
  assumes P is CSP1 healthy
  and Q is CSP1 healthy
  shows (P \lor Q) is CSP1 healthy
using assms by (auto simp: csp-defs design-defs rp-defs fun-eq-iff)

lemma disj-CSP2:
  P is CSP2 healthy \implies Q is CSP2 healthy \implies (P \lor Q) is CSP2 healthy
by (simp add: CSP2-is-H2[symmetric] Healthy-def' design-defs comp-ndet-l-distr)

lemma disj-CSP:
  assumes A: is-CSP-process P
  assumes B: is-CSP-process Q
  shows is-CSP-process (P \lor Q)
apply (simp add: is-CSP-process-def Healthy-def)
apply (subth disj-CSP2[simplified Healthy-def, symmetric])
apply (rule A[THEN CSP-is-CSP2, simplified Healthy-def])
apply (rule B[THEN CSP-is-CSP2, simplified Healthy-def], simp)
apply (subth disj-CSP1[simplified Healthy-def, symmetric])
apply (rule A[THEN CSP-is-CSP1, simplified Healthy-def])
apply (rule B[THEN CSP-is-CSP1, simplified Healthy-def], simp)
apply (subth R-disj[simplified Healthy-def])
apply (rule A[THEN CSP-is-R, simplified Healthy-def])
apply (rule B[THEN CSP-is-R, simplified Healthy-def], simp)
done

lemma seq-CSP1:
  assumes A: P is CSP1 healthy
  assumes B: Q is CSP1 healthy
shows \((P :: Q)\) is CSP1 healthy using \(A\) \(B\) by (auto simp: csp-defs design-defs rp-defs fun-eq-iff)

lemma seq-CSP2:
  assumes \(A:\ Q\) is CSP2 healthy
  shows \((P :: Q)\) is CSP2 healthy using \(A\)
  by (auto simp: CSP2-is-H2[symmetric] H2-J[symmetric])

lemma seq-R:
  assumes \(P\) is R healthy
  and \(Q\) is R healthy
  shows \((P :: Q)\) is R healthy
  proof –
    have \(R P = P\) and \(R Q = Q\)
      using assms by (simp-all only: Healthy-def)
    moreover have \((R P :: Q)\) is R healthy
      apply (auto simp add: design-defs rp-defs prefixeq-def fun-eq-iff split: cond-splits)
      apply (rule A[simplified Healthy-def])
      apply (rule CSP-is-CSP1[OF C, simplified Healthy-def])
      apply (simp add: Healthy-def, subst CSP1-idem, auto)
      done
  ultimately show \(?thesis\) by simp
qed

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 \((P :: Q)\)
  apply (auto simp add: is-CSP-process-def)
  apply (subst seq-CSP1[simplified Healthy-def])
  apply (rule A[simplified Healthy-def])
  apply (rule CSP-is-CSP1[OF C, simplified Healthy-def])
  apply (simp add: Healthy-def, subst CSP1-idem, auto)
  apply (rule A)[simplified Healthy-def])
  apply (rule CSP-is-CSP2[OF C, simplified Healthy-def])
  apply (simp add: Healthy-def, subst CSP2-idem, auto)
  apply (rule B[simplified Healthy-def])
  apply (rule CSP-is-R[OF C, simplified Healthy-def])
  apply (simp add: Healthy-def, subst R-idem2, auto)
  done

lemma rd-ind-wait: \((R\neg(P^f_j) \vdash (P^t_j))\)
  \(= (R\neg(\lambda (A, A') \ P (A, A'[ok := False])))\)
apply (auto simp: design-defs rp-defs fun-eq-iff split: cond-splits)
apply (subgoal-tac a(tr := [], wait := False) = a(tr := []), auto)
apply (subgoal-tac a(tr := [], wait := False) = a(tr := []), auto)
apply (subgoal-tac a(tr := [], wait := False) = a(tr := []), auto)
apply (subgoal-tac a(tr := [], wait := False) = a(tr := []), auto)
apply (subgoal-tac a(tr := [], wait := False) = a(tr := []), auto)
apply (rule-tac t=a(tr := [], wait := False) and s=a(tr := []) in subst, simp-all) done

lemma rd-H1: \( R((\neg (\lambda (A, A'). P (A, A'[\text{ok} := \text{False}])))) \)
\( \vdash (\lambda (A, A'). P (A, A'[\text{ok} := \text{True}])) ) \)
\( (R ((\neg H1 (\lambda (A, A'). P (A, A'[\text{ok} := \text{False}])))) = \)
\( (H1 (\lambda (A, A'). P (A, A'[\text{ok} := \text{True}]))) ) \)
by (auto simp: design-defs rp-defs fun-eq-iff split: cond-splits)

lemma rd-H1-H2: \( R((\neg H1 (\lambda (A, A'). P (A, A'[\text{ok} := \text{False}])))) \)
\( \vdash H1 (\lambda (A, A'). P (A, A'[\text{ok} := \text{True}])) ) \)
\( (R ((\neg(H1 o H2) (\lambda (A, A'). P (A, A'[\text{ok} := \text{False}])))) = \)
\( (H1 o H2) (\lambda (A, A'). P (A, A'[\text{ok} := \text{True}])) ) \)
apply (auto simp: design-defs rp-defs prefixeq-def fun-eq-iff split: cond-splits elim: alpha-d-move-eqE)
apply (subgoal-tac b(tr := zs, ok := False) = ba(ok := False), auto intro: alpha-d.equality)
apply (subgoal-tac b(tr := zs, ok := False) = ba(ok := False), auto intro: alpha-d.equality)
apply (subgoal-tac b(tr := zs, ok := False) = ba(ok := False), auto intro: alpha-d.equality)
apply (subgoal-tac b(tr := zs, ok := True) = ba(ok := True), auto intro: alpha-d.equality)
apply (subgoal-tac b(tr := zs, ok := True) = ba(ok := True), auto intro: alpha-d.equality)
done

lemma rd-H1-H2-R-H1-H2: \( R ((\neg (H1 o H2) (\lambda (A, A'). P (A, A'[\text{ok} := \text{False}])))) \)
\( \vdash (H1 o H2) (\lambda (A, A'). P (A, A'[\text{ok} := \text{True}])) ) \)
\( (R o H1 o H2) P \)
apply (auto simp: design-defs rp-defs fun-eq-iff split: cond-splits)
apply (erule notE) back
apply (rule-tac b=ba in comp-intro, auto)
apply (rule-tac t=ba(ok := False) and s=ba in subst, auto intro: alpha-d.equality)
apply (erule notE) back
apply (rule-tac b=ba in comp-intro, auto)
apply (rule-tac t=ba(ok := False) and s=ba in subst, auto intro: alpha-d.equality)
apply (case-tac ok ba)
apply (rule-tac b=ba in comp-intro, auto)
apply (rule-tac t=ba(ok := True) and s=ba in subst, auto)
apply (erule notE) back
apply (rule-tac b=ba in comp-intro, auto)
apply (rule-tac t=ba(ok := False) and s=ba in subst, auto intro: alpha-d.equality)
done

lemma CSP1-is-R1-H1:
assumes $P$ is $R_1$ healthy
shows $CSP_1 P = R_1 (H_1 P)$
using assms
by (auto simp: csp-defs design-defs rp-defs fun-eq-iff split: cond-splits)

lemma CSP1-is-R1-H1-2: $CSP_1 (R_1 P) = R_1 (H_1 P)$
by (auto simp: csp-defs design-defs rp-defs fun-eq-iff split: cond-splits)

lemma CSP1-R1-commute: $CSP_1 o R_1 = R_1 o CSP_1$
by (auto simp: csp-defs design-defs rp-defs fun-eq-iff split: cond-splits)

lemma CSP1-R1-commute2: $CSP_1 (R_1 P) = R_1 (CSP_1 P)$
by (auto simp: csp-defs design-defs rp-defs fun-eq-iff split: cond-splits)

lemma CSP1-is-R1-H1-b:
$(P = (R o R_1 o H_1 o H_2) P) = (P = (R o CSP_1 o H_2) P)$
apply (simp add: fun-eq-iff)
apply (subst H1-H2-commute2)
apply (subst R1-H2-commute2)
apply (subst CSP1-is-R1-H1-2[symmetric])
apply (subst H2-CSP1-commute2)
apply (subst R1-H2-commute2[symmetric])
apply (subst CSP1-R1-commute2)
apply (subst R-abs-R1[simplified Fun.comp-def fun-eq-iff])
apply (auto)
done

lemma CSP1-join:
assumes A: $x$ is $CSP_1$ healthy
and B: $y$ is $CSP_1$ healthy
shows $(x \cap y)$ is $CSP_1$ healthy
using A B
by (simp add: Healthy-def CSP1-def fun-eq-iff utp-defs)

lemma CSP2-join:
assumes A: $x$ is $CSP_2$ healthy
and B: $y$ is $CSP_2$ healthy
shows $(x \cap y)$ is $CSP_2$ healthy
using A B
apply (simp add: design-defs csp-defs fun-eq-iff)
apply (rule allI)
apply (rule allI)
apply (erule-tac x=a in allE)
apply (erule-tac x=a in allE)
apply (erule-tac x=b in allE)+
by (auto)

lemma CSP1-meet:
assumes A: $x$ is $CSP_1$ healthy
and $B$: $y$ is CSP1 healthy
shows $(x \sqcup y)$ is CSP1 healthy
using $A \quad B$
apply (simp add: Healthy-def CSP1-def fun-eq-iff utp-defs)
apply (rule allI)
apply (erule-tac $x=a$ in allE)
apply (erule-tac $x=a$ in allE)
apply (erule-tac $x=b$ in allE)+
by (auto)

lemma CSP2-meet:
assumes $A$: $x$ is CSP2 healthy
and $B$: $y$ is CSP2 healthy
shows $(x \sqcup y)$ is CSP2 healthy
using $A \quad B$
apply (simp add: Healthy-def CSP2-def fun-eq-iff)
apply (rule allI)+
apply (erule-tac $x=a$ in allE)
apply (erule-tac $x=a$ in allE)
apply (erule-tac $x=b$ in allE)+
apply (auto)
apply (rule-tac $b=ca$ in comp-intro)
apply (auto simp: J-csp-def)
done

lemma CSP-join:
assumes $A$: is-CSP-process $x$
and $B$: is-CSP-process $y$
shows is-CSP-process $(x \cap y)$
using $A \quad B$
by (simp add: is-CSP-process-def CSP1-join CSP2-join R-join)

lemma CSP-meet:
assumes $A$: is-CSP-process $x$
and $B$: is-CSP-process $y$
shows is-CSP-process $(x \sqcup y)$
using $A \quad B$
by (simp add: is-CSP-process-def CSP1-meet CSP2-meet R-meet)

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
by (auto simp: csp-defs design-defs rp-defs fun-eq-iff split: cond-splits elim: prefixE)

lemma rd-is-CSP2:
assumes A: \( \forall \ a \ b. \ r(a, b\{\text{ok} := \text{True}\}) \rightarrow r(a, b\{\text{ok} := \text{False}\}) \)
shows \( R(r \vdash p) \) is CSP2 healthy
apply (subst CSP2-is-H2[symmetric])
apply (simp add: Healthy-def)
apply (subst R-H2-commute2[symmetric])
apply (subst design-H2[simplified Healthy-def], auto simp: A)
done

lemma rd-is-CSP:
assumes A: \( \forall \ a \ b. \ r(a, b\{\text{ok} := \text{True}\}) \rightarrow r(a, b\{\text{ok} := \text{False}\}) \)
shows is-CSP-process \( R(r \vdash p) \)
apply (simp add: is-CSP-process-def Healthy-def fun-eq-iff)
apply (subst R-idem2)
apply (subst rd-is-CSP2[simplified Healthy-def, symmetric], rule A)
apply (subst rd-is-CSP1[simplified Healthy-def, symmetric], simp)
done

lemma CSP-is-rd:
assumes A: is-CSP-process \( P \)
shows \( P = (R(\neg (P f f) \vdash (P t f))) \)
apply (subst rd-ind-wait)
apply (subst rd-H1)
apply (subst rd-H1-H2)
apply (subst rd-H1-H2-R-H1-H2)
apply (subst R-abs-R1[symmetric])
apply (subst CSP1-is-R1-H1-b)
apply (subst CSP2-is-H2)
apply (simp)
apply (subst CSP-is-CSP2[OF A, simplified Healthy-def, symmetric])
apply (subst CSP-is-CSP1[OF A, simplified Healthy-def, symmetric])
apply (subst CSP-is-R[OF A, simplified Healthy-def, symmetric], simp)
done

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 (′ϑ::ev-eq, ′σ) action = {p::(′ϑ,′σ) relation-rp. is-CSP-process p}

morphism relation-of action-of

proof

have R (false ⊢ true) ∈ {p :: (′ϑ,′σ) relation-rp. is-CSP-process p}

by (auto intro: rd-is-CSP)

thus ?thesis by auto

qed

print-theorems

The type-definition introduces a new type by stating a set. In our case, it is the set of reactive processes that satisfy the healthiness-conditions for CSP-processes, isomorphic to the new type. Technically, this construct introduces two constants (morphism) definitions relation_of and action_of as well as the usual axioms expressing the bijection action_of (relation_of ?x) = ?x and ?y ∈ {p. is-CSP-process p} → relation_of (action_of ?y) = ?y.

lemma relation-of-CSP: is-CSP-process (relation_of x)

proof

have (relation_of x):{p. is-CSP-process p} by (rule relation_of)

then show is-CSP-process (relation_of x) ..

qed

lemma relation-of-CSP1: (relation_of x) is CSP1 healthy

by (rule CSP-is-CSP1[OF relation_of-CSP])

lemma relation-of-CSP2: (relation_of x) is CSP2 healthy

by (rule CSP-is-CSP2[OF relation_of-CSP])

lemma relation-of-R: (relation_of x) is R healthy

by (rule CSP-is-R[OF relation_of-CSP])

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:
∀ a b. (relation_of y → relation_of x) (a, b) →
(relation_of y)ʃ f (a|tr := [], b) →
(relation_of x)ʃ f (a|tr := [], b)

by (auto simp: spec_def)

lemma relation-of-spec-t-f:
∀ a b. (relation_of y → relation_of x) (a, b) →
(relation-of y) \_f \ (a\{tr := []; b) \implies 
(relation-of x) \_f \ (a\{tr := []; b)

\text{by (auto simp: spec-def)}

\text{instantiation action::(ev-eq, type) below}

\text{begin}

\text{definition ref-def : } P \subseteq Q \equiv [(relation-of Q) \implies (relation-of P)]

\text{instance ..}

\text{end}

\text{instance action :: (ev-eq, type) po}

\text{proof}

\text{fix x y z::('a, 'b) action}

\{ 
\text{show } x \subseteq x \text{ by (simp add: ref-def utp-defs)}
\text{next}
\text{assume } x \subseteq y \text{ and } y \subseteq z \text{ then show } x \subseteq z
\text{ by (simp add: ref-def utp-defs)}
\text{next}
\text{assume } A:x \subseteq y \text{ and } B:y \subseteq x \text{ then show } x = y
\text{ by (auto simp add: ref-def relation-of-inject[symmetric] fun-eq-iff)}
\}

\text{qed}

\text{instantiation action :: (ev-eq, type) lattice}

\text{begin}

\text{definition inf-action : } \inf P Q \equiv \text{action-of} ((relation-of P) \cap (relation-of Q))
\text{definition sup-action : } \sup P Q \equiv \text{action-of} ((relation-of P) \cup (relation-of Q))
\text{definition less-eq-action : } \text{less-eq} (P::('a, 'b) action) Q \equiv P \subseteq Q
\text{definition less-action : } \text{less} (P::('a, 'b) action) Q \equiv P \subseteq Q \land \neg Q \subseteq P

\text{instance}

\text{proof}

\text{fix x y z::('a, 'b) action}

\{ 
\text{show } (x < y) = (x \leq y \land \neg y \leq x)
\text{ by (simp add: less-action less-eq-action)}
\text{next}
\text{show } (x \leq x) \text{ by (simp add: less-action less-eq-action)}
\text{next}
\text{assume } x \leq y \text{ and } y \leq z
\text{ then show } x \leq z
\text{ by (simp add: less-action ref-def utp-defs)}
\text{next}
\text{assume } x \leq y \text{ and } y \leq x
\text{ then show } x = y
\text{ by (auto simp add: less-action ref-def relation-of-inject[symmetric] utp-defs)}
\text{next}

\}

57
show \( \inf x y \leq x \)
apply (auto simp add: less-eq-action inf-action ref-def csp-defs design-defs rp-defs)
apply (subst action-of-inverse, simp add: Healthy-def)
apply (insert relation-of-CSP[where \( x=z \)])
apply (insert relation-of-CSP[where \( x=y \)])
apply (simp-all add: CSP-join)
done

next
show \( \inf x y \leq y \)
apply (auto simp add: less-eq-action inf-action ref-def csp-defs)
apply (subst action-of-inverse, simp add: Healthy-def)
apply (insert relation-of-CSP[where \( x=x \)])
apply (insert relation-of-CSP[where \( x=y \)])
apply (simp-all add: CSP-join)
done

next
assume \( x \leq y \) and \( x \leq z \)
thenshow \( x \leq \inf y z \)
apply (auto simp add: less-eq-action inf-action ref-def impl-def csp-defs)
apply (erule-tac \( x=a \) in allE, erule-tac \( x=a \) in allE)
apply (erule-tac \( x=b \) in allE)+
apply (subst (asm) action-of-inverse)
apply (simp add: Healthy-def)
apply (insert relation-of-CSP[where \( x=z \)])
apply (insert relation-of-CSP[where \( x=y \)])
apply (auto simp add: CSP-join)
done

next
show \( x \leq \sup x y \)
apply (auto simp add: less-eq-action sup-action ref-def impl-def csp-defs)
apply (subst (asm) action-of-inverse)
apply (simp add: Healthy-def)
apply (insert relation-of-CSP[where \( x=x \)])
apply (insert relation-of-CSP[where \( x=y \)])
apply (auto simp add: CSP-meet)
done

next
show \( y \leq \sup x y \)
apply (auto simp add: less-eq-action sup-action ref-def impl-def csp-defs)
apply (subst (asm) action-of-inverse)
apply (simp add: Healthy-def)
apply (insert relation-of-CSP[where \( x=x \)])
apply (insert relation-of-CSP[where \( x=y \)])
apply (auto simp add: CSP-meet)
assumed \( y \leq x \) and \( z \leq x \)

then show \( \sup y z \leq x \)

apply (auto simp add: less-eq-action sup-action ref-def impl-def csp-defs)
apply (erule-tac x=a in allE)
apply (erule-tac x=a in allE)
apply (erule-tac x=b in allE)
apply (subst action-of-inverse)
apply (simp add: Healthy-def)
apply (insert relation-of-CSP [where \( x=z \)])
apply (insert relation-of-CSP [where \( x=y \)])
apply (auto simp add: CSP-meet)
done

qed

lemma bot-is-action: \( R (\text{false } \vdash \text{true}) \in \{ \text{p. is-CSP-process p} \} \)
by (auto intro: rd-is-CSP)

lemma bot-eq-true: \( R (\text{false } \vdash \text{true}) \) = \( R \text{ true} \)
by (auto simp: fun-eq-iff design-defs rp-defs split: cond-splits)

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

definition bot-action : \( \text{bot} :: (\text{'}a, \text{'}b) \text{ action} \) \( \equiv \) \( \text{action-of} \ (R(\text{false } \vdash \text{true})) \)
definition top-action : \( \text{top} :: (\text{'}a, \text{'}b) \text{ action} \) \( \equiv \) \( \text{action-of} \ (R(\text{true } \vdash \text{false})) \)

instance
proof
fix \( \text{'}a, \text{'}b \text{ action} \)
{
  show \( \text{bot} \leq x \)
  unfolding bot-action
  apply (auto simp add: less-action less-eq-action ref-def bot-action)
  apply (subst action-of-inverse) apply (rule bot-is-action)
  apply (subst bot-eq-true)
  apply (subst (asm) CSP-is-rd)
  apply (rule relation-of-CSP)
  apply (auto simp add: csp-defs rp-defs fun-eq-iff split: cond-splits)
done

next
show \( x \leq \text{top} \)
  apply (auto simp add: less-action less-eq-action ref-def top-action)
  apply (subst (asm) action-of-inverse)
  apply (simp)
}
apply (rule rd-is-CSP)
apply auto
apply (subst action-of-cases[where x=x], simp-all)
apply (subst action-of-inverse, simp-all)
apply (subst CSP-is-rd[where P=y], simp-all)
apply (auto simp: rp-defs design-defs fun-eq-iff split: cond-splits)
done

qed
end

lemma relation-of-top: relation-of top = R(true ⊢ false)
apply (simp add: top-action)
apply (subst action-of-inverse)
apply (simp)
apply (rule rd-is-CSP)
apply (auto simp add: utp-defs design-defs rp-defs)
done

lemma relation-of-bot: relation-of bot = R true
apply (simp add: bot-action)
apply (subst action-of-inverse)
apply (simp add: bot-is-action[simplified], rule bot-eq-true)
done

lemma non-emptyE: assumes A ≠ {} obtains x where x : A
using assms by (auto simp add: ex-in-conv [symmetric])

lemma CSP1-Inf:
assumes "*:A ≠ {}"
shows (Π relation-of ' A) is CSP1 healthy
proof –
  have (Π relation-of ' A) = CSP1 (Π relation-of ' A)
  proof
    fix P
    note * then
  apply (intro iffI)
  apply (simp-all add: csp-defs)
  apply (rule disj-introC, simp)
  apply (erule disj-elim, simp-all)
  apply (cases P, simp-all)
  apply (erule non-emptyE)
  apply (rule-tac x=Collect (relation-of x) in exI, simp)
  apply (rule conjI)
  apply (rule-tac x=(relation-of x) in exI, simp)
  apply (subst CSP-is-rd, simp add: relation-of-CSP)
  done
apply (auto simp add: csp-defs prefixeq-def design-defs rp-defs fun-eq-iff split: cond-splits)
done
qed
then show (∏ relation-of ' A) is CSP1 healthy by (simp add: design-defs)
qed

lemma CSP2-Inf:
assumes *:A ≠ {} shows (∏ relation-of ' A) is CSP2 healthy
proof –
have (∏ relation-of ' A) = CSP2 (∏ relation-of ' A)
proof
fix P
note * then
apply (intro iffI)
apply (simp-all add: csp-defs)
apply (cases P, simp-all)
apply (erule exE)
apply (rule-tac b=b in comp-intro, simp-all)
apply (rule-tac x=x in exI, simp)
apply (erule comp-elim, simp-all)
apply (erule exE | erule conjE)+
apply (simp-all)
apply (rule-tac x=Collect Pa in exI, simp)
apply (rule conjI)
apply (rule-tac x=Pa in exI, simp)
apply (erule Set.imageE, simp add: relation-of)
apply (subst CSP-is-rd, simp add: relation-of-CSP)
apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP)
apply (auto simp add: csp-defs rp-defs prefixeq-def design-defs fun-eq-iff split: cond-splits)
apply (subgoal-tac b{tr := zs, ok := False}) = c{tr := zs, ok := False}, auto)
apply (subgoal-tac b{tr := zs, ok := False}) = c{tr := zs, ok := False}, auto)
apply (subgoal-tac b{tr := zs, ok := False}) = c{tr := zs, ok := False}, auto)
apply (subgoal-tac b{tr := zs, ok := False}) = c{tr := zs, ok := False}, auto)
apply (subgoal-tac b{tr := zs, ok := False}) = c{tr := zs, ok := False}, auto)
apply (subgoal-tac b{tr := zs, ok := True}) = c{tr := zs, ok := True}, auto)
apply (subgoal-tac b{tr := zs, ok := True}) = c{tr := zs, ok := True}, auto)
donedef
qed
then show (∏ relation-of ' A) is CSP2 healthy by (simp add: design-defs)
qed

lemma R-Inf:
assumes *:A ≠ {}
shows ($\bigcap$ relation-of ' $A$) is R healthy
proof –
  have ($\bigcap$ relation-of ' $A$) = R ($\bigcap$ relation-of ' $A$)
  proof
  fix $P$
  show ($P \in \bigcup \{ \{ p . P p \} \mid P . P \in$ relation-of ' $A \} \} = R (\lambda Aa . Aa \in \bigcup \{ p . P p \} \mid P . P \in$ relation-of ' $A \} \} P$
    apply (cases $P$, simp-all)
    apply (rule)
    apply (simp-all add: csp-defs rp-defs split: cond-splits)
    apply (erule exE)
    apply (erule exE | erule conjE)+
    apply (simp-all)
    apply (erule Set.imageE, simp add: relation-of)
    apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP)
    apply (simp add: csp-defs prefixeq-def design-defs rp-defs fun-eq-iff split: cond-splits)
    apply (rule-tac $x=x$ in exI, simp)
    apply (rule conjI)
    apply (rule-tac $x=\text{relation-of } xa$ in exI, simp)
    apply (subst CSP-is-rd, simp add: relation-of-CSP)
    apply (simp add: csp-defs prefixeq-def design-defs rp-defs fun-eq-iff split: cond-splits)
    apply (erule exE | erule conjE)+
    apply (simp-all)
    apply (erule Set.imageE, simp add: relation-of)
    apply (rule-tac $x=x$ in exI, simp)
    apply (rule conjI)
    apply (rule-tac $x=\text{relation-of } xa$ in exI, simp)
    apply (subst CSP-is-rd, simp add: relation-of-CSP)
    apply (simp add: csp-defs prefixeq-def design-defs rp-defs fun-eq-iff split: cond-splits)
    apply (erule exE | erule conjE)+
    apply (simp-all)
    apply (erule Set.imageE, simp add: relation-of)
    apply (rule-tac $x=x$ in exI, simp)
    apply (rule conjI)
    apply (rule-tac $x=\text{relation-of } xa$ in exI, simp)
    apply (subst CSP-is-rd, simp add: relation-of-CSP)
    apply (simp add: csp-defs prefixeq-def design-defs rp-defs fun-eq-iff split: cond-splits)
    apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP)
    apply (simp add: csp-defs prefixeq-def design-defs rp-defs fun-eq-iff split: cond-splits)
  done
qed
then show ($\bigcap$ relation-of ' $A$) is R healthy by (simp add: design-defs)
qed

lemma CSP-Inf:
  assumes $A \neq \{\}$
  shows is-CSP-process ($\bigcap$ relation-of ' $A$)
  unfolding is-CSP-process-def
  using assms CSP1-Inf CSP2-Inf R-Inf
  by auto

62
lemma Inf-is-action: \( A \neq \{\} \implies \bigcap \text{relation-of} \ ' A \in \{ \text{p. is-CSP-process} \ p \} \)
by (auto dest!: CSP-Inf)

lemma CSP1-Sup: \( A \neq \{\} \implies (\bigcup \text{relation-of} \ ' A) \text{ is CSP1 healthy} \)
apply (auto simp add: design-defs csp-defs fun-eq-iff)
apply (subst CSP-is-rd, simp add: relation-of-CSP)
apply (simp add: csp-defs prefixeq-def design-defs rp-defs split: cond-splits)
done

lemma CSP2-Sup: \( A \neq \{\} \implies (\bigcup \text{relation-of} \ ' A) \text{ is CSP2 healthy} \)
apply (simp add: design-defs csp-defs fun-eq-iff)
apply (rule allI)+
apply (rule)
apply (rule-tac b=b in comp-intro, simp-all)
apply (erule comp-elim, simp-all)
apply (rule allI)
apply (erule-tac x in allE)
apply (rule_impl)
apply (case-tac (\( \exists \ P. \ x = \text{Collect} \ P \ & \ P \in \text{relation-of} \ ' A \), simp-all)
apply (erule exE | erule conjE)+
apply (simp-all)
apply (erule Set.imageE, simp add: relation-of)
apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP, subst CSP-is-rd, simp add: relation-of-CSP)
apply (auto simp add: csp-defs design-defs rp-defs split: cond-splits)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := False\] = \( c[tr := tr \ c - tr \ aa, ok := False\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := False\] = \( c[tr := tr \ c - tr \ aa, ok := False\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := False\] = \( c[tr := tr \ c - tr \ aa, ok := False\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := False\] = \( c[tr := tr \ c - tr \ aa, ok := False\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := False\] = \( c[tr := tr \ c - tr \ aa, ok := False\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := False\] = \( c[tr := tr \ c - tr \ aa, ok := False\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := True\] = \( c[tr := tr \ c - tr \ aa, ok := True\], simp-all)
apply (subgoal-tac ba\[tr := tr \ c - tr \ aa, ok := True\] = \( c[tr := tr \ c - tr \ aa, ok := True\], simp-all)
done

lemma R-Sup: \( A \neq \{\} \implies (\bigcup \text{relation-of} \ ' A) \text{ is R healthy} \)
apply (simp add: rp-defs design-defs csp-defs fun-eq-iff)
apply (rule allI)+
apply (rule)
apply (simp split: cond-splits)
apply (case-tac wait a, simp-all)
apply (erule non-emptyE)
apply (erule-tac x=Collect (relation-of x) in allE, simp-all)
apply (case-tac relation-of x (a, b), simp-all)
apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP)
apply (simp add: csp-defs design-defs rp-defs split: cond-splits)
apply (erule-tac x=(relation-of x) in allE, simp-all)
apply (rule conjI)
apply (rule allI)
apply (erule-tac x=Collect (relation-of x) in allE, simp-all)
apply (case-tac 3 P. x = Collect P & P ∈ relation-of ' A), simp-all)
apply (erule exE | erule conjE) +
apply (simp-all)
apply (erule Set.imageE, simp add: relation-of)
apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP, subst CSP-is-rd, simp add: relation-of-CSP)
apply (simp add: csp-defs design-defs rp-defs split: cond-splits)
apply (erule non-emptyE)
apply (erule-tac x=Collect (relation-of x) in allE, simp-all)
apply (case-tac relation-of x (a, b), simp-all)
apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP)
apply (simp add: csp-defs design-defs rp-defs split: cond-splits)
apply (erule-tac x=(relation-of x) in allE, simp-all)
apply (simp split: cond-splits)
apply (rule allI)
apply (rule impI)
apply (erule Set.imageE, simp add: relation-of)
apply (subst (asm) CSP-is-rd, simp add: relation-of-CSP, subst CSP-is-rd, simp add: relation-of-CSP)
apply (simp add: csp-defs design-defs rp-defs split: cond-splits)
apply (rule allI)
apply (rule impI)
apply (erule Set.imageE, simp add: relation-of)
apply (subst (asm) CSP-is-rd)
apply (case-tac relation-of xa (a|tr := [], b|tr := tr b − tr a)), simp-all)
apply (subst (asm) CSP-is-rd) back back back
apply (simp add: relation-of-CSP, subst CSP-is-rd, simp add: relation-of-CSP)
apply (simp add: csp-defs design-defs rp-defs split: cond-splits)
apply (erule-tac x=P in allE, simp-all)
done

lemma CSP-Sup: A ≠ {} ⇒ is-CSP-process (⨆ relation-of ' A)
unfolding is-CSP-process-def using CSP1-Sup CSP2-Sup R-Sup by auto
lemma Sup-is-action: \( A \neq \{\} \Rightarrow \bigcup \text{relation-of} \ A \in \{p \text{-is-CSP-process} p\} \)
by (auto dest!: CSP-Sup)

lemma relation-of-Sup:
\( A \neq \{\} \Rightarrow \text{relation-of} (\bigcup \text{relation-of} \ A) = \bigcup \text{relation-of} \ A \)
by (auto simp: action-of-inverse dest!: Sup-is-action)

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

definition Sup-action :
\( (\text{Sup} (S::(a',b)\text{ action set}) \equiv \begin{cases} \text{bot} & \text{if } S=\{\} \\
\text{action-of} (\bigcup \text{relation-of} S) & \text{else} \end{cases} \)\)
definition Inf-action :
\( (\text{Inf} (S::(a',b)\text{ action set}) \equiv \begin{cases} \text{top} & \text{if } S=\{\} \\
\text{action-of} (\bigcap \text{relation-of} S) & \text{else} \end{cases} \)\)

instance
proof
fix A::(a',b) action set and z::(a',b) action
{  
  fix x::(a',b) action
  assume x\in A
  then show Inf A \leq x
  apply (auto simp add: less-action less-eq-action Inf-action ref-def)
  apply (subst (asm) action-of-inverse)
  apply (auto intro: Inf-is-action[simplified])
  done
}
note rule1 = this

{  
  assume *: \( \forall x. x \in A \Rightarrow z \leq x \)
  show z \leq Inf A
  proof (cases A = \{\})
    case True
    then show \( \text{thesis} \) by (simp add: Inf-action)
  next
    case False
    show \( \text{thesis} \)
    using *
    apply (auto simp add: Inf-action)
    using A \neq \{\}
    apply (simp add: less-eq-action Inf-action ref-def)
    apply (subst (asm) action-of-inverse)
    apply (subst (asm) ex-in-cone[symmetric])
    apply (erule exE)
    apply (auto intro: Inf-is-action[simplified])
    done
  qed
}

fix x::(a',b) action
assume x\in A
then show $x \leq (\operatorname{Sup} A)$
  apply (auto simp add: less-action less-eq-action Sup-action ref-def)
  apply (subst (asm) action-of-inverse)
  apply (auto intro: Sup-is-action[simplified])
done
}

note rule2 = this
{
  assume $\forall x. \ x \in A \Rightarrow x \leq z$
  then show $\operatorname{Sup} A \leq z$
    apply (auto simp add: Sup-action)
    apply atomize
    apply (case-tac A = {}, simp-all)
    apply (insert rule2)
    apply (auto simp add: less-action less-eq-action Sup-action ref-def)
    apply (subst (asm) action-of-inverse)
    apply (auto intro: Sup-is-action[simplified])
    apply (subst (asm) action-of-inverse)
    apply (auto intro: Sup-is-action[simplified])
done
}

{ show $\operatorname{Inf} (\{\} :: (\') a, \') b) \operatorname{action set} = \operatorname{top}$ by (simp add: Inf-action )
}{ show $\operatorname{Sup} (\{\} :: (\') a, \') b) \operatorname{action set} = \operatorname{bot}$ by (simp add: Sup-action )
qed

end
end

13 Circus variables

theory Var-list
imports Main
begin

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

type-synonym $(\') a, \') \sigma$ var-list $= ((\') \sigma \Rightarrow \') a list) \times ((\') a list \Rightarrow \') a list) \Rightarrow \') \sigma \Rightarrow \') \sigma$

The select function returns the top value of the stack.

definition select :: $(\') a, \') r) var-list \Rightarrow \') r \Rightarrow \') a$
  where select $f \equiv \lambda A. \operatorname{hd} ((\operatorname{fst} f) A)$

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

definition increase :: $(\') a, \') r) var-list \Rightarrow \') a \Rightarrow \') r$
  where increase $f \ val \equiv (\operatorname{snd} f) (\lambda l. \ val\#l)$
The \textit{increase0} function pushes an arbitrary value to the top of the stack.

\textbf{definition} \textit{increase0} :: (\text{'a}, \text{'r}) \text{var-list} \Rightarrow \text{'r} \Rightarrow \text{'r}
\text{where} \textit{increase0} f \equiv (\text{snd} f) (\lambda l. ((\text{SOME val. True})\#l))

The \textit{decrease} function pops the top value of the stack.

\textbf{definition} \textit{decrease} :: (\text{'a}, \text{'r}) \text{var-list} \Rightarrow \text{'r} \Rightarrow \text{'r}
\text{where} \textit{decrease} f \equiv (\text{snd} f) (\lambda l. (\text{tl} l))

The \textit{update} function updates the top value of the stack.

\textbf{definition} \textit{update} :: (\text{'a}, \text{'r}) \text{var-list} \Rightarrow (\text{'a} \Rightarrow \text{'a}) \Rightarrow \text{'r} \Rightarrow \text{'r}
\text{where} \textit{update} f upd \equiv (\text{snd} f) (\lambda l. (\text{upd} (\text{hd} l))\#(\text{tl} l))

The \textit{update0} function initializes the top of the stack with an arbitrary value.

\textbf{definition} \textit{update0} :: (\text{'a}, \text{'r}) \text{var-list} \Rightarrow \text{'r} \Rightarrow \text{'r}
\text{where} \textit{update0} f \equiv (\text{snd} f) (\lambda l. ((\text{SOME upd. True}) (\text{hd} l))\#(\text{tl} l))

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

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

\textbf{syntax} \text{-VAR-LIST} :: \text{id} \Rightarrow (\text{'a}, \text{'r}) \text{var-list} (\text{VAR-\textit{LIST} -})
\textbf{translations} \text{VAR-LIST} x =\rangle (x, -\text{-update-name x})

\textbf{end}

\section{Denotational semantics of Circus actions}

\textbf{theory} \textit{Denotational-Semantics}
\textbf{imports} \textit{Circus-Actions \text{Var-list}}
\textbf{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.

\subsection{Skip}

\textbf{definition} \textit{Skip} :: (\text{'\emptyset::ev-eq}, \sigma) \text{action} \text{where}
\textit{Skip} \equiv \text{action-of}
(R (\text{true} \vdash \lambda(A, A'). \text{tr} A' = \text{tr} A \land \neg \text{wait} A' \land \text{more} A = \text{more} A'))

\textbf{lemma} \textit{Skip-is-action:}
(R (\text{true} \vdash \lambda(A, A'). \text{tr} A' = \text{tr} A \land \neg \text{wait} A' \land \text{more} A = \text{more} A')) \in \{p. \text{is-CSP-process p}\}
\textbf{apply} \text{ (simp)}
\textbf{apply} \text{ (rule rd-is-CSP)}
\textbf{by} \text{ auto}
lemmas Skip-is-CSP = Skip-is-action[simplified]

lemma relation-of-Skip:
relation-of Skip =
  \( R (\text{true} \vdash \lambda (A, A'). tr A' = tr A \land \neg \text{wait} A' \land \text{more} A = \text{more} A') \)
by (simp add: Skip-def action-of-inverse Skip-is-CSP)

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

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

lemma Skip-is-CSP3: (relation-of Skip) is CSP3 healthy
apply (auto simp: relation-of-Skip rp-defs design-defs fun-eq-iff CSP3-def)
apply (split cond-splits, simp-all)+
apply (rule-tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
prefer 3
apply (split cond-splits, simp-all)+
apply (auto simp add: prefixeq-def)
done

lemma Skip-is-CSP4: (relation-of Skip) is CSP4 healthy
apply (auto simp: relation-of-Skip rp-defs design-defs fun-eq-iff CSP4-def)
apply (split cond-splits, simp-all)+
apply (rule-tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
prefer 3
apply (split cond-splits, simp-all)+
apply (auto simp add: prefixeq-def)
done

lemma Skip-comp-absorb: (relation-of Skip ;; relation-of Skip) = relation-of Skip
apply (auto simp: relation-of-Skip fun-eq-iff rp-defs true-def design-defs)
apply (clarsimp simp: cond-splits)+
apply (case-tac ok aa, simp-all)
apply (erule disjE)+
apply (clarsimp simp: prefixeq-def)
apply (clarsimp simp: prefixeq-def)
apply (erule disjE)+
apply (clarsimp simp: prefixeq-def)
apply (clarsimp simp: prefixeq-def)
apply (erule disjE)+

68
apply (clarsimp simp: prefixeq-def)
apply (clarsimp simp: prefixeq-def)
apply (case-tac ok aa, simp-all)
apply (clarsimp simp: prefixeq-def)
apply (clarsimp split: cond-splits)+
apply (rule-tac b=a in comp-intro)
apply (clarsimp split: cond-splits)+
apply (rule-tac b=a in comp-intro)
apply (clarsimp split: cond-splits)+
done

14.2 Stop

definition Stop :: ('ϑ::ev-eq,'σ) action
where Stop ≡ action-of \((\lambda (A', A)\cdot \text{tr } A' = \text{tr } A \land \text{wait } A')\)

lemma Stop-is-action:
\((\lambda (A', A)\cdot \text{tr } A' = \text{tr } A \land \text{wait } A')\) ∈ \{\text{p. is-CSP-process } p\}
apply (simp)
apply (rule rd-is-CSP)
by auto

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

lemma relation-of-Stop:
relation-of Stop = \((\lambda (A', A)\cdot \text{tr } A' = \text{tr } A \land \text{wait } A')\)
by (simp add: Stop-def action-of-inverse Stop-is-CSP)

lemma Stop-is-CSP3: (relation-of Stop) is CSP3 healthy
apply (auto simp: relation-of-Stop relation-of-Skip rp-defs design-defs fun-eq-iff
CSP3-def)
apply (rule-tac b=a in comp-intro)
apply (split cond-splits, auto)
apply (split cond-splits)+
apply (simp-all)
apply (case-tac ok aa, simp-all)
apply (case-tac tr aa ≤ tr ba, simp-all)
apply (case-tac ok ba, simp-all)
apply (case-tac tr ba ≤ tr c, simp-all)
apply (rule disjI1)
apply (simp add: prefixeq-def)
apply (erule exE)+
apply (rule-tac x=zs@zsa in exI, simp)
apply (rule disjI1)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule-tac x=zs@zsa in exI, simp)
apply (split cond-splits)+
apply (simp-all add: true-def)
apply (erule disjE)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule_tac x@zs@zsa in exI, simp)
apply (auto simp add: prefixeq-def)
done

lemma Stop-is-CSP4; (relation-of Stop) is CSP4 healthy
apply (auto simp: relation-of-Stop relation-of-Skip rp-defs design-defs fun-eq-iff CSP4-def)
apply (rule_tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
apply (case-tac ok aa, simp-all)
apply (case-tac tr aa ≤ tr ba, simp-all)
apply (case-tac ok ba, simp-all)
apply (case-tac tr ba ≤ tr c, simp-all)
apply (erule disjI)
apply (simp add: prefixeq-def)
apply (erule exE)+
apply (rule_tac x@zs@zsa in exI, simp)
apply (rule disjI1)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule_tac x@zs@zsa in exI, simp)
apply (split cond-splits)+
apply (simp-all add: true-def)
apply (erule disjE)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule tac x@zs@zsa in exI, simp)
apply (auto simp add: prefixeq-def)
done

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}
apply (simp)
apply (rule rd-is-CSP)
by auto

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

lemma relation-of-Chaos; relation-of Chaos = (R(false ⊢ true))
by (simp add: Chaos-def action-of-inverse Chaos-is-CSP)
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 ⇒ ('τ::ev-eq, σ) action ⇒ ('τ, 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}
apply (simp)
apply (rule seq-CSP)
apply (rule rd-is-CSP1)
apply (auto simp: R-idem2 Healthy-def relation-of-CSP)
done

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)
by (simp add: state-update-before-def action-of-inverse state-update-before-is-CSP)

lemma mono-state-update-before: mono (state-update-before sc)
by (auto simp: mono-def less-eq-action ref-def relation-of-state-update-before design-defs
rp-defs fun-eq-iff
split: cond-splits dest: relation-of-spec-f-f[|simplified|
relation-of-spec-t-f[|simplified|])

lemma state-update-before-is-CSP3: relation-of (state-update-before sc Ac) is CSP3
healthy
apply (auto simp: relation-of-state-update-before relation-of-Skip rp-defs design-defs
fun-eq-iff CSP3-def)
apply (rule-tac b=aa in comp-intro)
apply (split cond-splits, auto)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro)
apply (split cond-splits, simp-all)+
apply (case-tac ok aa, simp-all)
apply (case-tac tr aa ≤ tr ab, simp-all)
apply (case-tac ok ab, simp-all)
apply (case-tac tr ab ≤ tr bb, simp-all)
apply (rule disjI1)
apply (simp add: prefixeq-def)
apply (erule exE)+
apply (rule-tac $x @ zsa$ in exI, simp)
apply (rule-tac $b = bb$ in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule disjI1)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule-tac $x @ zsa$ in exI, simp)
apply (rule-tac $b = bb$ in comp-intro)
apply (split cond-splits, simp-all)+
apply (simp-all add: true-def)
apply (erule disjE)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule-tac $x @ zsa$ in exI, simp)
apply (auto simp add: prefixeq-def)
done

lemma state-update-before-is-CSP4:
  assumes A: relation-of Ac is CSP4 healthy
  shows relation-of (state-update-before sc Ac) is CSP4 healthy
apply (auto simp: relation-of-state-update-before relation-of-Skip rp-defs design-defs fun-eq-iff CSP4-def)
apply (rule-tac $b = c$ in comp-intro)
apply (rule-tac $b = ba$ in comp-intro, simp-all)
apply (split cond-splits, simp-all)
apply (rule-tac $b = bb$ in comp-intro, simp-all)
apply (subst A[ simplified design-defs rp-defs CSP4-def relation-of-Skip])
apply (auto simp: rp-defs)
done

definition state-update-after :: $\sigma$ relation $\Rightarrow$ ('$\theta$: ev-eq,'$\sigma$) action $\Rightarrow$ ('$\theta$,'$\sigma$) action
where state-update-after sc Ac = action-of(relation-of Ac ; ; $R (true \mapsto \lambda (A, A'). sc (more A, more A') & \sim$wait A' & tr A = tr A'))

lemma state-update-after-is-action:
  (relation-of Ac ; ; $R (true \mapsto \lambda (A, A'). sc (more A, more A') & \sim$wait A' & tr A = tr A')) $\in \{ p. is-CSP-process p \}$
apply (simp)
apply (rule seq-CSP)
apply (auto simp: relation-of-CSP[simplified is-CSP-process-def])
apply (rule rd-is-CSP, auto)
done

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

lemma relation-of-state-update-after:
  relation-of (state-update-after sc Ac) = (relation-of Ac ; ; $R (true \mapsto \lambda (A, A'). sc$
(more $A$, more $A'$) & \neg \text{wait} A' & \text{tr} A = \text{tr} A' )))

by (simp add: state-update-after-def action-of-inverse state-update-after-is-CSP)

lemma mono-state-update-after: mono (state-update-after sc)
by (auto simp: mono-def less-eq-action ref-def relation-of-state-update-after design-defs
    rp-defs fun-eq-iff
    split: cond-splits dest: relation-of-spec-f-f[simplified]
    relation-of-spec-t-f[simplified])

lemma state-update-after-is-CSP3:
  assumes $A : \text{relation-of } Ac \text{ is CSP3 healthy}$
  shows $\text{relation-of } (\text{state-update-after sc } Ac) \text{ is CSP3 healthy}$
apply (auto simp: relation-of-state-update-after relation-of-Skip rp-defs design-defs
    fun-eq-iff CSP3-def)
apply (rule-tac b=aa in comp-intro)
apply (split cond-splits, auto)
apply (rule-tac b=bb in comp-intro, simp-all)
apply (subst A[simplified design-defs rp-defs CSP3-def relation-of-Skip])
apply (auto simp: rp-defs)
done

lemma state-update-after-is-CSP4: relation-of (state-update-after sc Ac) is CSP4 healthy
apply (auto simp: relation-of-state-update-after relation-of-Skip rp-defs design-defs
    fun-eq-iff CSP4-def)
apply (rule-tac b=c in comp-intro)
apply (rule-tac b=ba in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (case-tac ok bb, simp-all)
apply (case-tac tr bb \leq tr c, simp-all)
apply (case-tac ok ca, simp-all)
apply (case-tac tr ca \leq tr c, simp-all)
apply (simp add: prefixeq-def)
apply (erule exE)+
apply (erule-tac x=zs@zsa in allE, simp)
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all add: true-def)+
apply (case-tac ok ca, simp-all)
apply (case-tac tr ca \leq tr c, simp-all)
apply (simp add: prefixeq-def)
apply (erule exE | erule conjE)+
apply (rule-tac x=zsa@zs in exI, simp)
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (case-tac tr bb \leq tr c, simp-all)
apply (simp add: prefixeq-def)
y
done

14.5 Sequential composition
definition

lemma Seq-is-action: (relation-of P ;; relation-of Q) ∈ {p. is-CSP-process p}
apply (simp)
y

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

lemma relation-of-Seq: relation-of (P ;; Q) = (relation-of P ;; relation-of Q)
by (simp add: Seq-def action-of-inverse Seq-is-CSP)

lemma mono-Seq: mono (op ;; P)
by (auto simp: mono-def less-eq-action ref-def relation-of-Seq)

done

lemma CSP3-imp-left-Skip:
  assumes A: relation-of P is CSP3 healthy
  shows (Skip ;; P) = P
apply (subst relation-of-inject[symmetric])
y

lemma CSP4-imp-right-Skip:
  assumes A: relation-of P is CSP4 healthy
  shows (P ;; Skip) = P
apply (subst relation-of-inject[symmetric])
y

lemma Seq-associ: (A ;; (B ;; C)) = ((A ;; B) ;; C)
by (auto simp: relation-of-inject[symmetric] fun-eq-iff relation-of-Seq rp-defs design-defs)

lemma Skip-absorb: (Skip ;; Skip) = Skip
by (auto simp: Skip-comp-absorb relation-of-inject[symmetric] relation-of-Seq)

done

14.6 Internal choice
definition

74
\( Ndet::(\forall v, e, q, \sigma) \text{ action} \Rightarrow (\forall v, \sigma) \text{ action} \Rightarrow (\forall v, \sigma) \text{ action} \) (infix \( \sqcap \) \ 18)

where \( P \sqcap Q \equiv \text{action-of} ((\text{relation-of} P) \lor (\text{relation-of} Q)) \)

**Lemma** \( Ndet-is-action\): \((\text{relation-of} P) \lor (\text{relation-of} Q)) \in \{ p. \text{is-CSP-process p}\}

apply (simp)
apply (rule disj-CSP)
apply (simp-all add: relation-of-CSP)
done

**Lemmas** \( Ndet-is-CSP = Ndet-is-action\)[simplified]

**Lemma** \( relation-of-Ndet\): \(\text{relation-of} (P \sqcap Q) = ((\text{relation-of} P) \lor (\text{relation-of} Q))\)
by (simp add: Ndet-def action-of-inverse Ndet-is-CSP)

**Lemma** \( mono-Ndet\): \( mono (op \sqcap P) \)
by (auto simp: mono-def less-eq-action ref-def relation-of-Ndet)

### 14.7 External choice

**Definition**
\( Det::(\forall v, e, q, \sigma) \text{ action} \Rightarrow (\forall v, \sigma) \text{ action} \Rightarrow (\forall v, \sigma) \text{ action} \) (infix \( \sqcup \) \ 18)

where \( P \sqcup Q \equiv \text{action-of}(R((\neg((\text{relation-of} P)^{\downarrow} f) \land \neg((\text{relation-of} Q)^{\downarrow} f)) \vdash (((\text{relation-of} P)^{\downarrow} r \land ((\text{relation-of} Q)^{\downarrow} r)) \land ((\text{relation-of} P)^{\downarrow} r \lor ((\text{relation-of} Q)^{\downarrow} r)))) \in \{ p. \text{is-CSP-process p}\}

apply (simp add: spec-def)
apply (rule rd-is-CSP)
apply (auto)
done

**Lemmas** \( Det-is-CSP = Det-is-action\)[simplified]

**Lemma** \( relation-of-Det\):
\(\text{relation-of} (P \sqcup Q) = (R((\neg((\text{relation-of} P)^{\downarrow} f) \land \neg((\text{relation-of} Q)^{\downarrow} f)) \vdash (((\text{relation-of} P)^{\downarrow} r \land ((\text{relation-of} Q)^{\downarrow} r)) \land ((\text{relation-of} P)^{\downarrow} r \lor ((\text{relation-of} Q)^{\downarrow} r)))) \in \{ p. \text{is-CSP-process p}\}

apply (unfold Det-def)
apply (rule action-of-inverse)
apply (rule Det-is-action)
done

lemma mono-Det: mono \((\text{op} \ [\implies] \ P)\)
by (auto simp: mono-def less-eq-action ref-def relation-of-Det design-defs rp-defs
fun-eq-iff
split: cond-splits dest: relation-of-spec-f-f[simplified]
relation-of-spec-t-f[simplified])

14.8 Reactive design assignment

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

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

lemmas rd-assign-is-CSP = rd-assign-is-action[simplified]

lemma relation-of-rd-assign:
relation-of \((\text{rd-assign} \ s) = \)
\((R \ (true \vdash \lambda(A, A'). \ \text{ref} \ A' = \text{ref} \ A \land \text{tr} \ A' = \text{tr} \ A \land \neg \text{wait} \ A' \land \text{more} \ A' = s))\)
by (simp add: rd-assign-def action-of-inverse rd-assign-is-CSP)

14.9 Local state external choice

definition Loc:: \('\varphi:'ev-eq,'\sigma) \Rightarrow \('\varphi,'\sigma) \Rightarrow \('\varphi,'\sigma) \Rightarrow \('\varphi,'\sigma) \Rightarrow \('\varphi,'\sigma) \Rightarrow \('\varphi,'\sigma) \Rightarrow \('\varphi,'\sigma)
where \((\text{loc} \ s1 \bullet P) \oplus (\text{loc} \ s2 \bullet Q) \equiv \ ((\text{rd-assign} \ s1)'; \ 'P) \uplus ((\text{rd-assign} \ s2)'; \ 'Q)\)

14.10 Schema expression

definition Schema :: \('\varphi)'relation \Rightarrow \('\varphi::ev-eq,'\sigma) \Rightarrow \('\varphi::ev-eq,'\sigma) \Rightarrow \('\varphi::ev-eq,'\sigma) \Rightarrow \('\varphi::ev-eq,'\sigma) \Rightarrow \('\varphi::ev-eq,'\sigma) \Rightarrow \('\varphi::ev-eq,'\sigma)
where Schema \ sc \equiv \text{action-of}\ ((R \ ((\lambda(A, A'). \ (\text{Pre} \ sc \ (\text{more} \ A))) \vdash \ (\lambda(A, A'). \ sc \ (\text{more} \ A, \ \text{more} \ A') \land \neg \text{wait} \ A' \land \text{tr} \ A = \text{tr} \ A')))\)

lemma Schema-is-action:
\((R \ ((\lambda(A, A'). \ (\text{Pre} \ sc \ (\text{more} \ A))) \vdash \ (\lambda(A, A'). \ sc \ (\text{more} \ A, \ \text{more} A') \land \neg \text{wait} A' \land \text{tr} A = \text{tr} A')))\) \(\in \ \{p. \ \text{is-CSP-process} \ p\}\)
apply (simp)
apply (rule rd-is-CSP)

76
apply (auto)
done

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

lemma relation-of-Schema:
relation-of (Schema sc) = (R ((λ(A, A'). (Pre sc) (more A)) ⊢
(λ(A, A'). sc (more A, more A') ∧ ¬wait A' ∧ tr A = tr A')))
by (simp add: Schema-def action-of-inverse Schema-is-CSP)

lemma Schema-is-state-update-before: Schema u = state-update-before u Skip
apply (subst relation-of-inject[symmetric])
apply (auto simp: relation-of-Schema relation-of-state-update-before relation-of-Skip
rp-defs fun-eq-iff
design-defs)
apply (split cond-splits, simp-all)
apply (rule comp-intro)
apply (split cond-splits, simp-all)+
apply (rule comp-intro)
apply (split cond-splits, simp-all)+
prefer 3
apply (split cond-splits, simp-all)+
apply (auto simp: prefixeq-def)
done

14.11 Parallel composition

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

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

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

fun ParMergel :: ''σ:ev-eq list ⇒ ''σ list ⇒ ''σ list ⇒ ''σ 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)))))

77
else (if (filter-chan-set b cs)
    then (a # (ParMerge tr1 (b # tr2) cs))
    else (a # (ParMerge tr1 (b # tr2) cs)))
\( \otimes (b \# (ParMerge (a \# tr1) tr2 cs))) \)

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

**lemma** set-Cons1: tr1 ∈ set l ⇒ a ≠ tr1 ∈ op # a  ' set l
by (auto)

**lemma** tr-in-set-eq: (tr1 ∈ op # b  ' set l) = (tr1 ≠ [] ∧ hd tr1 = b ∧ tl tr1 ∈ set l)
by (induct l) auto

definition M-par::(('θ::ev-eq, 'σ) alpha-rp-scheme ⇒ ('σ ⇒ 'σ ⇒ 'σ)
⇒ ('θ, 'σ) alpha-rp-scheme ⇒ ('σ ⇒ 'σ ⇒ 'σ)
⇒ ('θ set ⇒ ('θ, 'σ) relation-rp where
M-par s1 x1 s2 x2 cs =
\(((\lambda(S, S')). ((\text{diff-tr} S') S) \in \text{ParMerge} (\text{diff-tr} s1 S) (\text{diff-tr} s2 S) cs \&
\text{ev-eq} (\text{tr-filter} (tr s1) cs) (\text{tr-filter} (tr s2) cs)) \&
((\lambda(S, S')). (wait s1 \lor wait s2) \&
ref S' \subseteq (((\text{ref s1}) \cup (\text{ref s2}) \cap cs) \cup (\text{ref s1} \cap (\text{ref s2})^{-}cs)))
< wait o snd >
((\lambda(S, S')). (\neg \text{wait} s1 \lor \neg \text{wait} s2) \& \text{MergeSt} ((\text{more s1}, x1) ((\text{more s2}, x2))) )

**definition** Par::(('θ::ev-eq, 'σ) action ⇒ ('σ ⇒ 'σ ⇒ 'σ) ⇒ 'θ set ⇒ ('σ ⇒ 'σ ⇒ 'σ)
⇒ ('θ, 'σ) action ⇒ ('θ, 'σ) action (\text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} \text{-} where
\begin{align*}
A1 \quad [\text{ns1} | \text{cs}] \quad [\text{ns2} | \text{cs}] \quad A2 \equiv (\text{action-of} (R ((\lambda (S, S')).
\neg (\exists tr1 tr2. ((\text{relation-of} A1)^f ; ((\lambda (S, S')). tr1 = (tr S)) (S, S'))
\& (\text{spec False} (wait S) (\text{relation-of} A2) ; (\lambda (S, -). tr2 = (tr S))) (S, S'))
\& ((\text{tr-filter} tr1 cs) = (\text{tr-filter} tr2 cs))) \&
\neg (\exists tr1 tr2. (\text{spec False} (wait S) (\text{relation-of} A1); (\lambda (S, -). tr1 = tr S)) (S, S'))
\& ((\text{relation-of} A2)^f ; ((\lambda (S, S'). tr2 = (tr S))) (S, S'))
\& ((\text{tr-filter} tr1 cs) = (\text{tr-filter} tr2 cs)))
\end{align*}

**lemma** Par-is-action: (R ((\lambda (S, S')).
\neg (\exists tr1 tr2. ((\text{relation-of} A1)^f ; ((\lambda (S, S'). tr1 = (tr S))) (S, S'))
\& (\text{spec False} (wait S) (\text{relation-of} A2) ; (\lambda (S, S'). tr2 = (tr S))) (S, S'))
\& ((\text{tr-filter} tr1 cs) = (\text{tr-filter} tr2 cs)))
\neg (\exists tr1 tr2. (\text{spec False} (wait S) (\text{relation-of} A1); (\lambda (S, -). tr1 = tr S)) (S, S'))
\& ((\text{relation-of} A2)^f ; ((\lambda (S, S'). tr2 = (tr S))) (S, S'))
\& ((\text{tr-filter} tr1 cs) = (\text{tr-filter} tr2 cs)))
\))
(\lambda (S, S'). (\exists s1 s2. ((\lambda (A, A'). (\text{relation-of} A1)^f (A, s1))
\& ((\text{relation-of} A2)^f (A, s2))) ; M-par s1 ns1 s2 ns2 cs (S, S'))))


87
\( (\text{relation-of } A2)^f_f (A, s2) ) \); \( M\text{-par } s1 \; ns1 \; s2 \; ns2 \; cs \) \( (S, S') ) ) \) ) \in \{ p. \; is-CSP\text{-process } p \} \\
\text{apply (simp)} \\
\text{apply (rule rd-is-CSP)} \\
\text{apply (blast)} \\
\text{done }

\text{lemmas } Par\text{-is-CSP} = Par\text{-is-action}[\text{simplified}]

\text{lemma } relation\text{-of-Par:}
relation\text{-of} \( (A1 \ [ \ ns1 \ | \ cs \ | \ ns2 \ ] \ A2) = (R (\lambda (S, S') \).
\sim (\exists \ tr1 \ tr2. ((\text{relation-of } A1)^f_f ; ) \; \lambda (S, S'). \; tr1 = (tr S) ) ) (S, S') \\
\sim (\text{spec False } (wait S) (\text{relation-of } A2) ; ) \; \lambda (S, S'). \; tr2 = (tr S) ) ) (S, S') \\
\sim ((\text{tr-filter } tr1 \ cs) = (\text{tr-filter } tr2 \ cs)) ) ) \wedge \\
\sim (\exists \ tr1 \ tr2. (\text{spec False } (wait S) (\text{relation-of } A1) ; ) \; \lambda (S, -. \; tr1 = tr S) ) (S, S') \\
\wedge ((\text{relation-of } A2)^f_f ; ) \; \lambda (S, S'). \; tr2 = (tr S) ) ) (S, S') \\
\wedge ((\text{tr-filter } tr1 \ cs) = (\text{tr-filter } tr2 \ cs)) ) ) ) \wedge \\
\lambda (S, S'). \; (\exists s1 s2. (\lambda (A, A'). (\text{relation-of } A1)^f_f (A, s1) \\
\wedge ((\text{relation-of } A2)^f_f (A, s2))) ; ) \; M\text{-par } s1 \; ns1 \; s2 \; ns2 \; cs ) (S, S') ) ) ) ) ) \\
\text{apply (unfold Par-def)} \\
\text{apply (rule action\text{-of-inverse)} \\
\text{apply (rule Par\text{-is-action)} \\
\text{done}

\text{lemma } mono\text{-Par: } mono (\lambda Q. \; P \ [ \ ns1 \ | \ cs \ | \ ns2 \ ] \ Q) \\
\text{apply (auto simp: mono-def less-eq-action ref-def relation\text{-of-Par} design-defs} \\
\text{fun-eq-iff rp-defs} \\
\text{split: cond-splits)} \\
\text{apply (auto simp: rp-defs dest: relation\text{-of-spec-f-f}[simplified] relation\text{-of-spec-t-f}[simplified])} \\
\text{apply (erule-tac x=tr ba in allE, auto)} \\
\text{apply (erule notE)} \\
\text{apply (auto dest: relation\text{-of-spec-f-f relation\text{-of-spec-t-f})} \\
\text{done}

14.12 Local parallel block

\text{definition } ParLoc::'(\sigma \Rightarrow ' \sigma \Rightarrow ' \sigma) \Rightarrow (\emptyset::ev-eq, ' \sigma) \text{ action } ' \emptyset \; \text{ set } ' \sigma \Rightarrow (\sigma \Rightarrow ' \sigma \Rightarrow ' \sigma) \Rightarrow (\emptyset, ' \sigma) \text{ action } \\
'(\emptyset, ' \sigma, ' \varphi) \Rightarrow (\emptyset, ' \sigma, ' \varphi) \text{ action } \\
(' (\emptyset)\text{-par | - \bullet - }') [ - ] ' ((' (\emptyset)\text{-par | - \bullet - }')) \\
\text{where } \\
(\text{par } s1 \; \bullet P) \ [ \ cs \ ] (\text{par } s2 \; \bullet Q) \equiv ((\text{rd\text{-assign } s1}); \; ' P) \ [ \ ns1 \ | \ cs \ | \ ns2 \ ] ((\text{rd\text{-assign } s2}); \; ' Q) \\

14.13 Assignment

\text{definition } ASSIGN::'(v, ' \sigma) \text{ var-list } (\sigma \Rightarrow ' v) \Rightarrow (\emptyset::ev-eq, ' \sigma) \text{ action where } \\
\text{ASSIGN } x \; e \equiv \text{action-of} \ (R (true \Rightarrow (\lambda (S, S'). \; tr S' = tr S \wedge \sim\text{-wait } S' \wedge \\
\text{(more } S' = (\text{update } x \ (\lambda -. \; (e \ (\text{more } S)) (\text{more } S)))))))
syntax -assign::id ⇒ (′σ ⇒ ′v) ⇒ (′ϑ, ′σ) action {- := - }
translations y ′:=′ vv ⇒ CONST ASSIGN (VAR y) vv

lemma Assign-is-action:
\( R \ (true \vdash (λ (S, S'). tr S' = tr S \land \neg wait S' \land (more S' = (update x (λ-. (e (more S)))) (more S)))) \in \{ p. is-CSP-process p \} \)
apply (simp)
apply (rule rd-is-CSP)
apply (blast)
done

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

lemma relation-of-Assign:
relation-of (ASSIGN x e) = \( R \ (true \vdash (λ (S, S'). tr S' = tr S \land \neg wait S' \land (more S' = (update x (λ-. (e (more S)))) (more S)))) \)
by (simp add: ASSIGN-def action-of-inverse Assign-is-CSP)

lemma Assign-is-state-update-before: ASSIGN x e = state-update-before (λ (s, s') . s' = (update x (λ-. (e s))) s) Skip
apply (subst relation-of-inject[ symmetric])
Pre-def update-def design-defs)
apply (split cond-splits, simp-all)+
apply (rule-tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=b in comp-intro)
apply (split cond-splits, simp-all)+
deref
apply (split cond-splits, simp-all)+
pref 3
apply (split cond-splits, simp-all)+
apply (auto simp add: prefixeq-def)
done

14.14 Variable scope

definition Var::(′v, ′σ) var-list ⇒ (′ϑ, ′σ) action ⇒ (′ϑ::ev-eq, ′σ) action where
Var v A ≡ action-of(\( R(\text{true} \vdash (λ (A, A'). \exists a. tr A' = tr A \land \neg wait A' \land more A' = (increase v a (more A))))))) ;
(relation-of A; ;
(\( R(\text{true} \vdash (λ (A, A'). tr A' = tr A \land \neg wait A' \land more A' = (decrease v (more A)))))))
syntax -var::idt ⇒ (′ϑ, ′σ) action ⇒ (′ϑ, ′σ) action (var - • [1000] 999)
translations var y • Act => CONST Var (VAR-LIST y) Act

lemma Var-is-action:
((R(true ⊢ (λ (A, A'). ∃ a. tr A' = tr A ∧ ¬wait A' ∧ more A' = (increase v a (more 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}
apply (simp)
apply (rule seq-CSP)
prefer 3
apply (auto simp: relation-of-CSP1 relation-of-R)
apply (rule rd-is-CSP)
apply (auto simp: csp-defs rp-defs design-defs fun-eq-iff prefixeq-def increase-def decrease-def
  split: cond-splits)
done

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

lemma relation-of-Var:
relation-of (Var v A) =
  ((R(true ⊢ (λ (A, A'). ∃ a. tr A' = tr A ∧ ¬wait A' ∧ more A' = (increase v a (more A)))); ;
  (relation-of A); ;
  (R(true ⊢ (λ (A, A'). tr A' = tr A ∧ ¬wait A' ∧ more A' = (decrease v (more A)))))))
apply (simp only: Var-def)
apply (rule action-of-inverse)
apply (rule Var-is-action)
done

lemma mono-Var : mono (Var x)
  by (auto simp: mono-def less-eq-action ref-def relation-of-Var)

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

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

lemma Let-is-action:
  (relation-of A; ;

81
\[ \left( R(\text{true} \vdash (\lambda (A, A'). \text{tr} A' = \text{tr} A \land \neg \text{wait} A' \land \text{more} A' = (\text{decrease} v \ (\text{more} A')))) \right) \in \{ p. \text{is-CSP-process} \ p \} \]

apply (simp)
apply (rule seq-CSP)
apply (auto simp: relation-of-CSP1 relation-of-R)
apply (rule rd-is-CSP)
apply (auto)
done

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

lemma relation-of-Let:
relation-of (Let v A) =
\( (\text{relation-of } A; ; \)
\( (R(\text{true} \vdash (\lambda (A, A'). \text{tr} A' = \text{tr} A \land \neg \text{wait} A' \land \text{more} A' = (\text{decrease} v \ (\text{more} A')))) \))
by (simp add: Let-def action-of-inverse Let-is-CSP)

lemma mono-Let : mono (Let x)
by (auto simp: mono-def less-eq-action ref-def relation-of-Let)

lemma Var-is-state-update-before: Var v A = state-update-before (\( \lambda (s, s'). \exists a. \ s' = \text{increase} v a s \)) (Let v A)
apply (subst relation-of-inject[symmetric])
apply (auto simp: rp-defs fun-eq-iff Pre-def design-defs)
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (split cond-splits, simp-all)+
apply (split cond-splits, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (rule-tac b=bb in comp-intro, simp-all)
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (case-tac ∃ A′ a. A′ = increase v a (alpha-rp.more aa), simp-all add: true-def)
apply (erule-tac x=increase v a (alpha-rp.more aa) in allE)
apply (erule-tac x=a in allE, simp)
apply (rule-tac b=bb in comp-intro, simp-all)+
apply (rule-tac b=ab in comp-intro)
apply (split cond-splits, simp-all)+
apply (case-tac ∃ A′ a. A′ = increase v a (alpha-rp.more aa), simp-all add: true-def)
apply (erule-tac x=increase v a (alpha-rp.more aa) in allE)
apply (erule-tac x=a in allE, simp)
done

lemma Let-is-state-update-after: Let v A = state-update-after (λ (s, s′). s′ = decrease v s) A
apply (subst relation-of-inject[ symmetric])
apply (auto simp: rp-defs fun-eq-iff Pre-def design-defs)
apply (auto split: cond-splits)
done

14.15 Guarded action

definition Guard::σ predicate ⇒ (σ:ev-eq, σ) action ⇒ (σ, σ) action (¬ ‘&’ -)
where g ‘&’ 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})
by (auto simp add: spec-def intro: rd-is-CSP)

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

lemma relation-of-Guard:
relation-of (g ‘&’ P) = (R (((g o more o fst) → ¬ ((relation-of P)’f)) ⊨
(((g o more o fst) ∧ ((relation-of P)’f)) ∨
\[
((\neg (g \circ \text{more} \circ \text{fst})) \land (\lambda (A, A'). \text{tr} A' = \text{tr} A \land \text{wait} A')))\]

apply (unfold Guard-def)
apply (subst action-of-inverse)
apply (simp-all only: Guard-is-action)
done

lemma mono-Guard : mono (Guard g)
apply (auto simp: mono-def less-eq-action ref-def rp-defs design-defs relation-of-Guard
split: cond-splits)
apply (auto dest: relation-of-spec-f-f relation-of-spec-t-f)
done

lemma false-Guard: false ' & P = Stop
apply (subst relation-of-inject[symmetric])
apply (subst relation-of-Stop)
apply (subst relation-of-Guard)
apply (simp add: fun-eq-iff utp-defs csp-defs design-defs rp-defs)
done

lemma false-Guard1: \(\forall a b. g (\text{alpha-rp}. \text{more} a) = \text{False} \Rightarrow \) (relation-of \((g ' & P)) (a, b) = (relation-of \text{Stop}) (a, b)
apply (subst relation-of-Guard)
apply (subst relation-of-Stop)
apply (auto simp: fun-eq-iff csp-defs design-defs rp-defs split: cond-splits)
done

lemma true-Guard: true ' & P = P
apply (subst relation-of-inject[symmetric])
apply (subst relation-of-Guard)
apply (subst CSP-is-rd[OF relation-of-CSP]) back back
apply (simp add: fun-eq-iff utp-defs csp-defs design-defs rp-defs)
done

lemma true-Guard1: \(\forall a b. g (\text{alpha-rp}. \text{more} a) = \text{True} \Rightarrow \) (relation-of \((g ' & P)) (a, b) = (relation-of \text{P}) (a, b)
apply (subst relation-of-Guard)
apply (subst CSP-is-rd[OF relation-of-CSP]) back back
apply (auto simp: fun-eq-iff csp-defs design-defs rp-defs split: cond-splits)
done

lemma Guard-is-state-update-before: g ' & P = state-update-before (\lambda (s, s'). g s) P
apply (subst relation-of-inject[symmetric])
apply (auto simp: relation-of-Guard relation-of-state-update-before relation-of-Skip rp-defs fun-eq-iff
Pre-def update-def design-defs)
apply (rule tac b=a in comp-intro)
apply (split cond-splits, simp-all)
apply (subst CSP-is-rd)
apply (simp-all add: relation-of-CSP rp-defs design-defs fun-eq-iff)
apply (split cond-splits, simp-all)+
apply (auto)
apply (subst (asm) CSP-is-rd)
apply (simp-all add: relation-of-CSP rp-defs design-defs fun-eq-iff)
apply (split cond-splits, simp-all)+
apply (subst CSP-is-rd)
apply (simp-all add: relation-of-CSP rp-defs design-defs fun-eq-iff)
apply (split cond-splits, simp-all)+
apply (auto) defer
apply (split cond-splits, simp-all)+
apply (subst CSP-is-rd)
apply (simp-all add: relation-of-CSP rp-defs design-defs fun-eq-iff)
apply (split cond-splits, simp-all)+
apply (auto) defer
apply (rule disjI1) defer
apply (case-tac g (alpha-rp. more aa), simp-all)
apply (rule)+
apply (simp add: impl-def) defer
oops

14.16 Prefixed action
definition do where
do e ≡ (λ(A, A'). tr A = tr A' ∧ (e (more A)) ⟨ wait o snd ⟩ (λ(A, A'). tr A' = (tr A ⊦[[e (more A)]])))
definition do-I ('σ ⇒ 'θ) ⇒ 'θ set ⇒ ('θ, 'σ) relation-rp
where do-I c S ≡ ((λ(A, A'). tr A = tr A' & S ∩ (ref A') = {}) ⟨ wait o snd ⟩
(λ(A, A'). hd (tr A' − tr A) ∈ S & (c (more A) = (last (tr A')))))
definition iPrefix ('σ ⇒ 'θ::ev-eq) ⇒ ('σ relation) ⇒ (''θ, 'σ) action ⇒ (''θ, 'σ) action ⇒ ('σ ⇒ ''θ set) ⇒ ('''θ, 'σ) action ⇒ ('''θ, 'σ) action where
iPrefix c i j S P ≡ action-of(R(true ⊦ (λ (A, A'). (do-I c (S (more A)) (A, A') & more A' = more A))); ' P
definition oPrefix ('σ ⇒ 'θ) ⇒ (''θ::ev-eq, 'σ) action ⇒ (''θ, 'σ) action where
\( o\text{Prefix}\ c\ P \equiv \text{action-of}(R(\text{true} \vdash (\text{do}\ c) \land (\lambda (A, A').\ \text{more}\ A' = \text{more}\ A)))\); ' P

definition \( \text{Prefix0}
\vdash \theta \Rightarrow (\theta::\text{ev-eq}, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{Prefix0}\ c\ P \equiv \text{action-of}(R(\text{true} \vdash (\text{do}\ (\lambda -.\ c)) \land (\lambda (A, A').\ \text{more}\ A' = \text{more}\ A)))\); ' P

definition read::'v \Rightarrow \theta \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{read}\ c\ x\ P \equiv \text{iPrefix}\ (\lambda A.\ c (\text{select}\ x\ A))\) \( (\lambda (s, s').\ \exists\ a.\ s' = \text{increase}\ x\ a\ s)\) \( (\lambda x)\) \( (\lambda s.\ s')\) \( P\)

definition read1::'v \Rightarrow \theta \Rightarrow (\sigma \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{readI}1\ c\ x\ S\ P \equiv \text{iPrefix}\ (\lambda A.\ c (\text{select}\ x\ A))\) \( (\lambda (s, s').\ \exists\ a.\ a\in S\ s'\ &\ s' = \text{increase}\ x\ a\ s)\) \( (\lambda s.\ s')\) \( P\)

definition write1::'v \Rightarrow \theta \Rightarrow (\theta::\text{ev-eq}, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{writeI}1\ c\ a\ P \equiv \text{oPrefix}\ (\lambda A.\ c (a\ A))\) \( P\)

definition write0::\theta \Rightarrow (\theta::\text{ev-eq}, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{writeI}0\ c\ P \equiv \text{Suffix}\ c\ P\)

definition read::[id, pttrn, (\theta, \sigma) action] \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{readS}::[id, pttrn, (\theta, \sigma) action] \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{readSS}::[id, pttrn, (\theta, \sigma) action] \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where
\( \text{writeS}::[id, \theta, (\theta, \sigma) action] \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow (\theta, \sigma) \Rightarrow \theta\) action where

translations
\( \text{read}\ c\ p\ P \equiv \text{CONST}\ \text{read}\ c\ (\text{VAR-LIST}\ p)\) \( P\)
\( \text{readS}\ c\ p\ b\ P \equiv \text{CONST}\ \text{readI}1\ c\ (\text{VAR-LIST}\ p)\) \( (\lambda b)\) \( P\)
\( \text{readSS}\ c\ p\ b\ P \equiv \text{CONST}\ \text{readI}1\ c\ (\text{VAR-LIST}\ p)\) \( b\) \( P\)
\( \text{write}\ c\ p\ P \equiv \text{CONST}\ \text{writeI}1\ c\ p\ P\)
\( \text{writeS}\ a\ P \equiv \text{CONST}\ \text{writeI}0\ a\ P\)

lemma \( \text{Prefix-is-action}::(R(\text{true} \vdash (\text{do}\ c) \land (\lambda (A, A').\ \text{more}\ A' = \text{more}\ A)))\in\{\text{p. is-CSP-process}\ p\}\)
by \(\text{auto intro: rd-is-CSP}\)

lemma \( \text{PrefixI-is-action}::(R(\text{true} \vdash (\lambda (A, A').\ \text{do-I}\ c\ (S\ (\lambda (\text{alpha-rp}\ .\ \text{more}\ A).
\text{alpha-rp}\ .\ \text{more}\ A' = \text{alpha-rp}\ .\ \text{more}\ A)))\) \in\{\text{p. is-CSP-process}\ p\}\)
by \(\text{auto intro: rd-is-CSP}\)
lemma Prefix0-is-action:
\[ (R(\text{true} \vdash (\text{do} \ C) \land (\lambda (A, A'). \text{more} A' = \text{more} A))) \in \{p. \text{-CSP-process } p\} \]
by (auto intro: rd-is-CSP)

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 \((i\text{Prefix } c \ i \ j \ S \ P)\) = 
\[ (\lnot \exists s. (\text{diff-tr} S' S) = (\text{tr-filter} (s - (\text{tr} S)) cs) \land \text{relation-of } P) \]
by (simp add: iPrefix-def relation-of-Seq action-of-inverse Prefix1-is-CSP)

lemma relation-of-oPrefix:
relation-of \((o\text{Prefix } c \ P)\) = 
\[ (\lnot \exists s. (\text{diff-tr} S' S) = (\text{tr-filter} (s - (\text{tr} S)) cs) \land \text{relation-of } P) \]
by (simp add: oPrefix-def relation-of-Seq action-of-inverse Prefix-is-CSP)

lemma relation-of-Prefix0:
relation-of \((\text{Prefix0 } c \ P)\) = 
\[ (\lnot \exists s. (\text{diff-tr} S' S) = (\text{tr-filter} (s - (\text{tr} S)) cs) \land \text{relation-of } P) \]
by (simp add: Prefix0-def relation-of-Seq action-of-inverse Prefix0-is-CSP)

lemma mono-iPrefix : mono \((i\text{Prefix } c \ i \ j \ s)\)
by (auto simp: mono-def less-eq-action ref-def relation-of-iPrefix)

lemma mono-oPrefix : mono \((o\text{Prefix } c)\)
by (auto simp: mono-def less-eq-action ref-def relation-of-oPrefix)

lemma mono-Prefix0 : mono \((\text{Prefix0 } c)\)
by (auto simp: mono-def less-eq-action ref-def relation-of-Prefix0)

14.17 Hiding

definition Hide::\(\theta::\text{ev-ev}, \sigma\) action ⇒ \(\theta \ set \ ⇒ \ (\theta, \sigma) \ action\ (\text{infixl} \ 18)\)
where
\[ P \ \set \equiv \ \text{action-of}(R(\lambda (S, S'). \exists s. (\text{diff-tr} S' S) = (\text{tr-filter} (s - (\text{tr} S)) cs) \land \text{relation-of } P)(S, S'\{tr := s, ref := (\text{ref} S') \cup cs \})); \ (\text{relation-of Skip}) \]

definition hid \(P cs \equiv (R(\lambda (S, S'). \exists s. (\text{diff-tr} S' S) = (\text{tr-filter} (s - (\text{tr} S)) cs) \land \text{relation-of } P)(S, S'\{tr := s, ref := (\text{ref} S') \cup cs \})); \ (\text{relation-of Skip}) \)
lemma hid-is-R: hid P cs is R healthy
apply (simp add: hid-def)
apply (rule seq-R)
apply (simp add: Healthy-def R-idem2)
apply (rule CSP-is-R)
apply (rule relation-of-CSP)
done

lemma hid-Skip: hid P cs = (hid P cs ; ; relation-of Skip)
by (simp add: hid-def comp-assoc[symmetric] Skip-comp-absorb)

lemma hid-is-CSP1: hid P cs is CSP1 healthy
apply (auto simp add: CSP1-def hid-def rp-defs fun-eq-iff)
apply (rule-tac b=a in comp-intro)
apply (clarsimp split: cond-splits)
apply (subst CSP-is-rd, auto simp: rp-defs design-defs true-def split: cond-splits)
apply (rule-tac x=[] in exI, auto)
done

lemma hid-is-CSP2: hid P cs is CSP2 healthy
apply (simp add: hid-def)
apply (rule seq-CSP2)
apply (rule CSP-is-CSP2)
apply (rule relation-of-CSP)
done

lemma hid-is-CSP: is-CSP-process (hid P cs)
by (auto simp: csp-defs hid-is-CSP1 hid-is-R hid-is-CSP2)

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)) \in
\{ p. is-CSP-process p \}
by (simp add: hid-is-CSP[simplified hid-def])

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))
by (simp add: Hide-def action-of-inverse Hide-is-CSP)

lemma mono-Hide : mono(\lambda P. P \ cs)
by (auto simp: mono-def less-eq-action ref-def prefixeq-def atp-defs relation-of-Hide rp-defs)
14.18 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. The operator "lfp" is inherited from the "Complete Lattice class" under some conditions. All theorems defined over this operator can be reused.

In the Circus-Actions theory, we presented the proof that Circus actions form a complete lattice. The Knaster-Tarski Theorem (in its simplest formulation) states that any monotone function on a complete lattice has a least fixed-point. This is a consequence of the basic boundary properties of the complete lattice operations. Instantiating the complete lattice class allows one to inherit these properties with the definition of the least fixed-point for monotonic functions over Circus actions.

```
syntax -MU::[idt, idt ⇒ (‘σ, ‘σ) action] ⇒ (‘σ, ‘σ) action (μ - • -)
translations -MU X P == CONST lfp (λ X. P)
```

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 ⇒ ‘a) ⇒ ‘r ⇒ ‘r 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 ⟨⟨
structure VARs-Data = Proof-Data
```
```
nonterminal circus-action and circus-schema

syntax
- circus-action :: 'a => circus-action (-)
- circus-schema :: 'a => circus-schema (-)

parse-translation ⟨⟨
let
fun antiquote-tr ctxt =
  let
    val {State-vars=sv, Alpha-vars=av} = VARs-Data.get ctxt
    fun get-selector x =
      let val c = Consts.intern (Proof-Contextconsts-of ctxt) x
      in
        if member (op =) av x then SOME (Const (Circus-Syntax.list-select, dummyT) $ (Syntax.const c)) else
        if member (op =) sv x then SOME (Syntax.const c) else NONE end;

    fun get-update x =
      let val c = Consts.intern (Proof-Contextconsts-of ctxt) x
      in
        if member (op =) av x then SOME (Const (Circus-Syntax.list-update-const, dummyT) $ (Syntax.const c Record.updateN)) else
        if member (op =) sv x then SOME (Const (Circus-Syntax.update-const, dummyT) $ (Syntax.const c Record.updateN)) else NONE end;

    fun print text = (fn x => let val - = writeln text; in x end);

    val rel-op-type = @{typ ('a × 'b => bool) => ('b × 'c => bool) => 'a × 'c => bool};

    fun tr i (t as Free (x, -)) =
      (case get-selector x of
        SOME c => c $ Bound (i + 1)
      | NONE =>
        (case try (unsuffix ') x of
          SOME y =>
            (case get-selector y of SOME c => c $ Bound i | NONE => t)
          | NONE => t))
      | tr i (t as (Const (-synt-assign, -) $ Free (x, -) $ r)) =
        (case get-update x of
  ⟩⟩
⟩⟩
⟩⟩
\[ \text{SOME } c = \Rightarrow \ c \ (\text{tr } i \ r) \ (\text{Const } (\text{Product-Type.Pair}, \text{dummyT}) \ $ Bound \ (i + 1) \ $ Bound i) \ |
\text{NONE } = \Rightarrow \ t) \]  
  (* \[ \text{tr } i \ (t as (\text{Const } (c, \text{rel-op-type}) \ $ l $ r) = \text{print } c \]  
  ((\text{Syntax.const } @\{\text{const-name prod-case}\} $ \]  
  Abs (B, dummyT, Abs (B', dummyT, Const (c, rel-op-type))) $ tr i \]  
  l $ tr i r) \ $ (\text{Const } (\text{Product-Type.Pair}, \text{dummyT}) $ Bound \ (i + 1) $ Bound i)\)

\[ \text{tr } i \ (t \ u) = \text{tr } i \ t \ $ \text{tr } i \ u \]  
  \[ \text{tr } i \ (\text{Abs } (x, T, t)) = \text{Abs } (x, T, \text{tr } (i + 1) \ t) \]  
  \[ \text{tr } - a = a; \]  
  \[ \text{in } \text{tr } 0 \ \text{end}; \]

\[ \text{fun quote-tr ctxt } [t] = \]  
  \[ \text{Syntax.const } @\{\text{const-name case-prod}\} \]  
  \[ \text{Abs } (A, \text{dummyT}, \text{Abs } (A', \text{dummyT}, \text{antiquote-tr ctxt \ (Term.incr-boundvars 2 } t)) \]  
  \[ \text{quote-tr } - ts = \text{raise TERM } \text{quote-tr}, ts); \]
  \[ \text{in } \text{quote-tr } [\{\{\text{syntax-const -circus-schema}\}, \text{quote-tr}\} \end \]

### ML

\[ \text{fun get-fields } (\text{SOME } (\{\text{fields, parent, ...}\}: \text{Record.info})) \ thy = \]  
  \[ (\text{case parent of} \]  
  \[ \text{SOME } (-, y) = \Rightarrow \text{fields } @\{\text{get-fields } (\text{Record.get-info } thy y) \thy \]  
  \[ \text{| NONE } = \Rightarrow \text{fields} \]  
  \[ \text{| get-fields } \text{NONE } = \Rightarrow [] \]

\[ \text{val dummy } = \text{Term.dummy-pattern dummyT}; \]
\[ \text{fun mk-eq } (l, r) = \text{HOLogic.Trueprop } $ (\text{HOLogic.eq-const dummyT} $ l $ r) \]

\[ \text{fun add-datatype } (\text{params, binding}) \text{ constr-specs thy } = \]  
  \[ \text{let} \]  
  \[ \text{val } [(\text{dt-name}], \thy') = \thy \]  
  \[ \]  
  \[ \]  
  \[ \]  
  \[ \text{val constr-names } = \]  
  \[ \text{map fst } (\text{the-single } (\text{map } \#3 o \text{snd}) \]  
  \[ (\#\text{descr } (\text{BNF-LFP-Compat.the-info } thy' [\text{BNF-LFP-Compat.Keep-Nesting}] \]  
  \[ \text{dt-name}))); \]
  \[ \text{fun constr } (c, Ts) = (\text{Const } (c, \text{dummyT}), \text{length } Ts); \]
  \[ \text{val constrs } = \text{map } \#1 \text{ constr-specs } \text{~~ map constr } (\text{constr-names } \text{~~ map } \#2 \]  
  \[ \text{constr-specs}); \]
  \[ \text{in } [(\text{dt-name}, \text{constrs}, \thy') \text{ end}; \]

\[ \text{fun define-channels } (\text{params, binding}) \text{ typesyn channels thy } = \]  

91
case typesyn of
NONE =>
  let
    val dt-binding = Binding.suffix-name -channels binding;

  val constr-specs = map (fn (b, opt-T) => (b, the-list opt-T, NoSyn)) channels;
  val ((dt-name, constrs), thy1) =
    add-datatype (params, dt-binding) constr-specs thy;

  val T = Type (dt-name, []);

  val fun-name = ev-eq ^ ^ Long-Name.base-name dt-name;

  val ev-equ = Free (fun-name, T => T => HOLogic.boolT);

  val eqs = map-product (fn (-, (c, n)) => (fn (-, (c1,n1)) =>
    let
      val t = Term.list-comb (c, replicate n dummy);
      val t1 = Term.list-comb (c1, replicate n1 dummy);
      in
        if c = c1 then mk-eq ((ev-equ $ t $ t1), @{term True}) else mk-eq ((ev-equ $ t $ t1), @{term False})
    end)
  constrs constrs;

fun case-tac x ctxt
  = rtac (Drule.instantiate' [] [SOME x]
              (#exhaust (BNF-LFP-Compat.the-info (Proof-Context.theory-of ctxt) [BNF-LFP-Compat.Keep-Nesting]
                         dt-name)));

fun proof ctxt = (Class.intro-classes-tac ctxt [] THEN
  Subgoal.FOCUS (fn {context = ctxt', params = [(-, x)], ...} =>
    (case-tac x ctxt) 1
  ) THEN auto-tac ctxt') ctxt 1 THEN
  Subgoal.FOCUS (fn {context = ctxt', params = [(-, x), (-, y)],
    ...} =>
    ((case-tac x ctxt) THEN-ALL-NEW (case-tac y
ctxt)) 1
  ) THEN auto-tac ctxt') ctxt 1);

val thy2 =
  thy1 |
  > Class.instantiation ([dt-name], params, @{sort ev-eq})
  > Function-Fun.add-fun ([Binding.name fun-name, NONE, NoSyn])
    (map (pair Attrib.empty-binding) eqs) Function-Fun.fun-config
  > Local-Theory.restore
  > Class.prove-instantiation-exit (fn ctxt =>> proof ctxt);
in
  ((dt-name, constrs), thy2)
end |
(SOME typn) =>>
let
val dt-binding = Binding.suffix-name -channels binding;

val (dt-name, thy1) = 
    thy
    |> Named-Target.theory-init
    |> (fn ctxt => Typedecl.abbrev (dt-binding, map fst params, NoSyn))
    (Proof-Context.read-typ ctxt typn) ctxt;

val thy2 = thy1 |> Local-Theory.exit-global;
in
((dt-name, []), thy2)
end;

fun define-chanset binding channel-constrs (name, chans) thy = 
let
  val constrs = 
    filter (fn (b, -) => exists (fn a => a = Binding.name-of b) chans) channel-constrs;
  val bad-chans = 
    filter-out (fn a => exists (fn (b, -) => a = Binding.name-of b) channel-constrs) chans;
  val - = null bad-chans orelse error (Bad elements " commas-quote bad-chans " in chanset: " quote
    (Binding.print name));
  val base-name = Binding.name-of name;
  val cs = map (fn (-, (c, n)) => Term.list-comb (c, replicate n (Const (@{const-name undefined}, dummyT)))) constrs;
  val chanset-eq = mk-eq ((Free (base-name, dummyT)), (HOLogic.mk-set dummyT cs));
in
thy
  |> Named-Target.theory-init
  |> Specification.definition
      (SOME (Binding.qualified true base-name binding, NONE, NoSyn),
       (Attrib.empty-binding, chanset-eq))
  |> snd |> Local-Theory.exit-global
end;

fun define-nameset binding (rec-binding, alphabet) (ns-binding, names) thy = 
let
  val all-selectors = get-fields (Record.get-info thy (Sign.full-name thy rec-binding))
  thy
  val bad-names = 
    filter-out (fn a => exists (fn (b, -) => String.isSuffix a b) all-selectors) names;
  val - = null bad-names orelse error (Bad elements " commas-quote bad-names " in nameset: " quote
    (Binding.print ns-binding));
  val selectors =
fun define-schema binding (ex-binding, expr) (alph-bind, alpha, state) thy =
let
  val fields-names = (map (fn (x, T) => (Binding.name-of x, T)) (alpha @ state));
  val alpha' = (map (fn (x, T) => (Binding.name-of x, T)) alpha);
  val state' = (map (fn (x, T) => (Binding.name-of x, T)) state);
  val all-selectors = get-fields (Record.get-info thy (Sign.full-name thy alph-bind))
thy
  val base-name = Binding.name-of ex-binding;
  val ctxt = Proof-Context.init-global thy;
  val term = Syntax.read-term
    (ctxt
      |> VARs-Data.put {{State-vars=(map fst state'), Alpha-vars=(map fst alpha')}})
    |> Config.put Syntax.root @{nonterminal circus-schema}) expr;
  val sc-eq = mk-eq ((Free (base-name, dummyT)), term);
  in
thy

filter (fn (b, -) => exists (fn a => String.isSuffix a b) names) all-selectors;
val update = map (fn x => (fst x, ((suffix Record.updateN) o fst) x)) selectors;
val select = map (fn x => (fst x, Const(fst x, dummyT))) selectors;
val update = map (fn (x, y) => (x, Const(y, dummyT))) updates;
val l =
  map (fn (b, -) => Binding.name-of b) alphabet;
val formulas = map2 (fn (nx, x) =>
  fn (ny, y) =>
    if (exists (fn b => String.isSuffix b nx) l)
      then Abs (A, dummyT, (Const(Circus-Syntax.list-update, dummyT) $ x)
        $ (Abs (-, dummyT, (Const(Circus-Syntax.list-select, dummyT) $ y) $ (Bound 1)))))
    else Abs (A, dummyT, x $ (Abs (-, dummyT, y $ (Bound 1)))))
end;

fun define-schema binding (ex-binding, expr) (alph-bind, alpha, state) thy =
let
  val fields-names = (map (fn (x, T) => (Binding.name-of x, T)) (alpha @ state));
  val alpha' = (map (fn (x, T) => (Binding.name-of x, T)) alpha);
  val state' = (map (fn (x, T) => (Binding.name-of x, T)) state);
  val all-selectors = get-fields (Record.get-info thy (Sign.full-name thy alph-bind))
thy
  val base-name = Binding.name-of ex-binding;
  val ctxt = Proof-Context.init-global thy;
  val term = Syntax.read-term
    (ctxt
      |> VARs-Data.put {{State-vars=(map fst state'), Alpha-vars=(map fst alpha')}})
    |> Config.put Syntax.root @{nonterminal circus-schema}) expr;
  val sc-eq = mk-eq ((Free (base-name, dummyT)), term);
  in
thy

94
fun define-action binding (ex-binding, expr) alph-bind chan-bind thy =
  let
    val base-name = Binding.name-of ex-binding;
    val ctxt = Proof-Context.init-global thy;
    val actT = Circus-Actions.action;
    val action-eq = mk-eq
      ((Free (base-name, Type (actT, [Proof-Context.read-type-name {proper=true, strict=false} ctxt (Sign.full-name thy chan-bind)])),
        (Proof-Context.read-type-name {proper=true, strict=false} ctxt (Sign.full-name thy alph-bind))),
        (Syntax.parse-term ctxt expr));
  in
    thy
    |> Named-Target.theory-init
    |> Specification.definition
      (SOME (Binding.qualified true base-name binding, NONE, NoSyn), (Attrib.empty-binding, sc-eq))
    |> snd
    |> Local-Theory.exit-global
    end;

fun define-expr binding (alph-bind, alpha, state) chan-bind (ex-binding, (is-schema, expr)) =
  if is-schema then define-schema binding (ex-binding, expr) alph-bind alpha state
  else define-action binding (ex-binding, expr) alph-bind chan-bind;

fun prep-field prep-typ (b: binding, raw-T) ctxt =
  let
    val T = prep-typ ctxt raw-T;
    val ctxt' = Variable.declare-typ T ctxt;
  in ((b, T), ctxt') end;

fun prep-constr prep-typ (b: binding, raw-T) ctxt =
let
    val T = Option.map (prep-typ ctxt) raw-T;
    val ctxt' = fold Variable.declare-typ (the-list T) ctxt;
  in ((b, T), ctxt') end;

fun gen-circus-process prep-constraint prep-typ
  (raw-params, binding) raw-alphabet raw-state (typesyn, raw-channels) namesets
  chansets
  exprs act thy =
  let
    val ctxt = Proof-Context.init-global thy;

(* internalize arguments *)
    val params = map (prep-constraint ctxt) raw-params;
    val ctxt0 = fold (Variable.declare-typ o TFree) params ctxt;

    val (alphabet, ctxt1) = fold-map (prep-field prep-typ) raw-alphabet ctxt0;
    val (state, ctxt2) = fold-map (prep-field prep-typ) raw-state ctxt1;
    val (channels, ctxt3) = fold-map (prep-constr prep-typ) raw-channels ctxt2;

    val params' = map (Proof-Context.check-tfree ctxt3) params;

(* type definitions *)
    val fields =
      map (fn (b, T) => (b, T, NoSyn)) (map (apsnd HOLogic.listT) alphabet @ state);

    val thy1 = thy |
      not (null fields) ?
        Record.add-record (params', Binding.suffix-name -alphabet binding) NONE
    fields;

    val (channel-constrs, thy2) =
      if not (null channels) orelse is-some typesyn
      then apfst snd (define-channels (params', binding) typesyn channels thy1)
      else ([], thy1);
    val thy3 = thy2 |
      not (null chansets) ? fold (define-chanset binding channel-constrs) chansets
      not (null namesets) ?
        fold (define-nameset binding ((Binding.suffix-name -alphabet binding), alphabet)) namesets
      not (null exprs) ?
        fold (define-expr binding ((Binding.suffix-name -alphabet binding), alphabet, state))
        (Binding.suffix-name -channels binding)) exprs
| > define-action binding (binding, act)
  (Binding.suffix-name -alphabet binding) (Binding.suffix-name -channels binding);
  in
  thy3
end;

fun circus-process x = gen-circus-process (K I) Syntax.check-typ x;
fun circus-process-cmd x = gen-circus-process (apsnd o Typedecl.read-constraint) Syntax.read-typ x;

local

val fields =
  @{keyword []} |-- Parse.enum1 , (Parse.binding -- (@{keyword ::} |-- Parse!!! Parse.typ))
  --| @{keyword []};

val constrs =
  (@{keyword []} |-- Parse.enum1 , (Parse.binding -- Scan.option Parse.typ)
  --| @ [keyword []]) >> pair NONE
  || Parse.typ >> (fn b => (SOME b, []));

val names =
  @{keyword []} |-- Parse.enum1 , Parse.name --| @{keyword []};

in

val - =
  Outer-Syntax.command @{command-keyword circus-process} Circus process specification
  ((Parse.type-args-constrained -- Parse.binding -- | @{keyword =}) --
    Scan.optional (@{keyword alphabet} |-- Parse!!! (@{keyword =} |-- fields))
  [] --
    Scan.optional (@{keyword state} |-- Parse!!! (@{keyword =} |-- fields))
  [] --
    Scan.optional (@{keyword channel} |-- Parse!!! (@{keyword =} |-- constrs)) (NONE, []) --
    Scan.repeat (@{keyword nameset} |-- Parse!!! ((Parse.binding -- | @{keyword =})
      -- names)) --
    Scan.repeat (@{keyword chanset} |-- Parse!!! ((Parse.binding -- | @{keyword =})
      -- names)) --
    Scan.repeat (@{keyword schema} |-- Parse!!! ((Parse.binding -- | @{keyword =})
      -- (Parse.term >> pair true))) ||
    (@{keyword action} |-- Parse!!! ((Parse.binding -- | @{keyword =})
      -- (Parse.term >> pair false)))) --
    (Parse.where- -- Parse!!! Parse.term)
  >> (fn ((((((a, b), c), d), e), f), g), h) =>
    Toplevel.theory (circus-process-cmd a b c d e f g h));

97
16 Refinement and Simulation

definition Simul S b = extend (make (ok b) (wait b) (tr b) (ref b)) (S (more b))

definition Simulation::(′ϑ::ev-eq, ′σ::ev-eq) action ⇒ (′σ1 ⇒ ′σ) ⇒ (′ϑ, ′σ1) action ⇒ bool (¬ ≤ -)

where
A ≤S B ≡ ∀ a b. (relation-of B) (a, b) → (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: Stop ≤S Stop
by (auto simp: Simulation-def relation-of-Stop rp-defs design-defs Simul-def alpha-rp.defs

    split: cond-splits)

lemma Skip-Sim: Skip ≤S Skip
by (auto simp: Simulation-def relation-of-Skip design-def rp-defs Simul-def alpha-rp.defs

    split: cond-splits)

lemma Chaos-Sim: Chaos ≤S Chaos
by (auto simp: Simulation-def relation-of-Chaos rp-defs design-defs Simul-def alpha-rp.defs

    split: cond-splits)
lemma Ndet-Sim:
  assumes A: \( P \preceq_S Q \) and B: \( P' \preceq_S Q' \)
  shows \( (P \cap P') \preceq_S (Q \cap Q') \)
by (insert A B, auto simp: Simulation-def relation-of-Ndet)

lemma Det-Sim:
  assumes A: \( P \preceq_S Q \) and B: \( P' \preceq_S Q' \)
  shows \( (P \land P') \preceq_S (Q \land Q') \)
by (auto simp: Simulation-def relation-of-Det design-def rp-defs Simul-def alpha-rp-defs)

spec-def
split: cond-splits

lemma Schema-Sim:
  assumes A: \( \forall a. (\text{Pre sc1}) (S a) \Rightarrow (\text{Pre sc2}) a \)
  and B: \( \forall a b. ([\text{Pre sc1} (S a) ; \text{sc2} (a, b)] \Rightarrow \text{sc1} (S a, S b) \)
  shows \( (\text{Schema sc1}) \preceq_S (\text{Schema sc2}) \)
by (auto simp: Simulation-def Simul-def relation-of-Schema rp-defs design-defs alpha-rp-defs)

lemma SUb-Sim:
  assumes A: \( \forall a. (\text{Pre sc1}) (S a) \Rightarrow (\text{Pre sc2}) a \)
  and B: \( \forall a b. ([\text{Pre sc1} (S a) ; \text{sc2} (a, b)] \Rightarrow \text{sc1} (S a, S b) \)
  and C: \( P \preceq_S Q \)
  shows \( (\text{state-update-before sc1} P) \preceq_S (\text{state-update-before sc2} Q) \)
apply (auto simp: Simulation-def Simul-def relation-of-state-update-before rp-defs design-defs alpha-rp-defs)

spec-def
split: cond-splits

apply (erule notE)
back
apply (drule C[simplified Simulation-def, rule-format])
apply (rule-tac b=Simul S ba in comp-intro)
apply (auto simp: A B Simul-def alpha-rp-defs)
apply (clarsimp split: cond-splits)+
apply (erule notE)
back
apply (drule C[simplified Simulation-def, rule-format])
apply (rule-tac b=Simul S ba in comp-intro)
apply (auto simp: A B Simul-def alpha-rp-defs)
apply (clarsimp split: cond-splits)+
apply (erule notE)
back
apply (drule C[simplified Simulation-def, rule-format])
apply (rule-tac b=Simul S ba in comp-intro)
apply (auto simp: A B Simul-def alpha-rp.defs)
apply (clarsimp split: cond-splits)+
apply (drule C[simplified Simulation-def, rule-format])
apply (rule tac b=Simul S ba in comp-intro)
apply (auto simp: A B Simul-def alpha-rp.defs)
apply (clarsimp split: cond-splits)+
apply (rule B, auto)
done

lemma Seq-Sim:
  assumes A: P ⪯ S Q and B: P' ⪯ S Q'
  shows (P '; P') ⪯ S (Q'; Q')
by (auto simp: Simulation-def relation-of-Seq dest:
  A[simplified Simulation-def, rule-format]
  B[simplified Simulation-def, rule-format])

lemma Par-Sim:
  assumes A: P ⪯ S Q and B: P' ⪯ S Q'
  and C: \( \forall a b. S (ns'2 a b) = ns2 (S a) (S b) \)
  and D: \( \forall a b. S (ns'1 a b) = ns1 (S a) (S b) \)
  shows (P \[ ns1 | cs | ns2 \] P') ⪯ S (Q \[ ns'1 | cs | ns'2 \] Q')
apply (auto simp: Simulation-def relation-of-Par fun-eq-iff rp-defs Simul-def
design-defs spec-def
  alpha-rp.defs
  dest: A[simplified Simulation-def Simul-def, rule-format]
  B[simplified Simulation-def Simul-def, rule-format])
apply (clarsimp split: cond-splits)+
apply (simp, erule disjE, rule disjI1, simp, rule disjI2, simp-all, rule impI)
apply (auto)
apply (erule-tac x=tr ba in allE, auto)
apply (erule notE) back
apply (rule tac b=Simul S ba\[ok := False\] in comp-intro)
apply (auto simp: Simul-def alpha-rp.defs dest: A[simplified Simulation-def Simul-def,
  rule-format])
apply (erule-tac x=tr bb in allE, auto)
apply (erule notE) back
apply (rule tac b=Simul S bb\[ok := False\] in comp-intro)
apply (auto simp: Simul-def alpha-rp.defs dest: B[simplified Simulation-def Simul-def,
  rule-format])
apply (erule-tac x=tr ba in allE, auto)
apply (erule notE) back
apply (rule tac b=Simul S ba\[ok := False\] in comp-intro)
apply (auto simp: Simul-def alpha-rp.defs dest: A[simplified Simulation-def Simul-def,
  rule-format])
apply (erule-tac x=tr bb in allE, auto)
apply (erule notE) back
apply (rule tac b=Simul S bb\[ok := False\] in comp-intro)
apply (auto simp: Simul-def alpha-rp.defs dest: B[simplified Simulation-def Simul-def,
apply \( \text{rule-tac } x = \text{Simul } S \ s_1 \ \text{in } \text{exI} \)
apply \( \text{rule-tac } x = \text{Simul } S \ s_2 \ \text{in } \text{exI} \)
apply (auto simp: \text{Simul-def alpha-rp.defs}
    dest!: B[\text{simplified Simulation-def Simul-def}, \text{rule-format}]
    A[\text{simplified Simulation-def Simul-def}, \text{rule-format}]
split: \text{cond-splits})
apply (rule-tac \(x = \text{Simul } S \ ba\) \ \text{in } \text{comp-intro} )
apply (auto simp add: \text{M-par-def alpha-rp.defs difference-def fun-eq-iff ParMerge-def Simul-def}
    split : \text{cond-splits} )
apply (rule-tac \(b = \text{Simul } S \ ba\) \ \text{in } \text{comp-intro} )
apply (subst D \{ \text{where } a = (\text{alpha-rp.eqE}) \ \text{dest }! : \text{B}
    \text{simplified Simulation-def Simul-def}, \text{rule-format} \}
    \text{split : \text{cond-splits} })
apply (rule-tac \(b = (| \text{ok } = \text{ok } bb, \text{wait } = \text{wait } bb, \text{tr } = \text{tr } bb, \text{ref } = \text{ref } bb,
    \ldots = S (\text{alpha-rp.eqE}) \}) \ \text{in } \text{comp-intro}, \text{auto} )
apply (\text{subtst } D \{ \text{where } a = (\text{alpha-rp.eqE}) \ \text{and } b = (\text{alpha-rp.eqE}) \}
    \text{symmetric}, \text{simp} )
apply (rule-tac \(b = (| \text{ok } = \text{ok } bb, \text{wait } = \text{wait } bb, \text{tr } = \text{tr } bb, \text{ref } = \text{ref } bb,
    \ldots = S (\text{alpha-rp.eqE}) \}) \ \text{in } \text{comp-intro}, \text{auto} )
apply (\text{subtst } D \{ \text{where } a = (\text{alpha-rp.eqE}) \ \text{and } b = (\text{alpha-rp.eqE}) \}
    \text{symmetric}, \text{simp} )
apply (\text{subtst } C \{ \text{where } a = (\text{alpha-rp.eqE}) \ \text{and } b = (\text{alpha-rp.eqE}) \}
    \text{symmetric}, \text{simp} )
done

lemma \text{Assign-Sim}: 
assumes \( A : \bigwedge A. \text{vy } A = \text{vx } (S A) \)
and  \( B : \bigwedge \text{ff } A. (S (\text{y-update ff A})) = (\text{x-update ff } (S A)) \)
shows \((x \ ':= \ ' x) \preceq S (y \ ':= \ ' vy)\)
by (auto simp: \text{Simulation-def relation-of-Assign update-def rp-defs design-defs Simul-def A B}
    \text{alpha-rp.defs split : \text{cond-splits} })

lemma \text{Var-Sim}: 
assumes \( A : P \preceq S \ Q \ \text{and } B : \bigwedge \text{ff } A. (S ((\text{snd } b) \ \text{ff } A)) = (\text{snd } a) \ \text{ff } (S A) \)
shows \((\text{Var } a P) \preceq S (\text{Var } b \ Q) \)
apply (auto simp: \text{Simulation-def relation-of-Var rp-defs design-defs fun-eq-iff Simul-def B}
    \text{alpha-rp.defs increase-def decrease-def} )
apply (rule-tac \(b = \text{Simul } S \ ba\) \ \text{in } \text{comp-intro} )
apply (\text{split cond-splits} )+
apply (auto simp: \text{B alpha-rp.defs Simul-def elim!: alpha-rp-eqE} )
apply (rule-tac \(b = \text{Simul } S \ bb\) \ \text{in } \text{comp-intro} )
apply (\text{split cond-splits} )+
apply (auto simp: \text{B alpha-rp.defs Simul-def elim!: alpha-rp-eqE dest!: A[\text{simplified Simulation-def Simul-def}, \text{rule-format}]} )
apply (\text{split cond-splits} )+
apply (simp add: \text{alpha-rp.defs} )
apply (erule disjE, rule disjI1, simp, rule disjI2, simp)
apply (simp-all add: alpha-rp.defs true-def)
apply (rule impI, (erule conjE | simp)+)
apply (simp add: B)
apply (split cond-splits)+
apply (simp add: alpha-rp.defs)
apply (erule disjE, rule disjI1, simp, rule disjI2, simp-all)
apply (rule impI, (erule conjE | simp)+)
apply (simp add: B)
done

lemma Guard-Sim:
  assumes A: P ⪯ S Q and B: □ A. h A = g (S A)
  shows (g ’&’ P) ⪯ S (h ’&’ Q)
apply (auto simp: Simulation-def)
apply (case-tac h (alpha-rp.more a))
defer
apply (case-tac g (S (alpha-rp.more a)))
apply (auto simp: true-Guard1 false-Guard1 Simul-def alpha-rp.defs Simulation-def)
B
dest!: A[simplified, rule-format] Stop-Sim[simplified, rule-format])
done

lemma Write0-Sim:
  assumes A: P ⪯ S Q
  shows a → P ⪯ S a → Q
  using A
apply (auto simp: Simulation-def write0-def relation-of-Prefix0 design-defs rp-defs)
apply (erule-tac x = ba in allE)
apply (erule-tac x = ca in allE, auto)
apply (rule-tac b = Simul S ba in comp-intro)
apply (auto split: cond-splits simp: Simul-def alpha-rp.defs do-def)
done

lemma Read-Sim:
  assumes A: P ⪯ S Q and B: □ A. (d A) = c (S A)
  shows a’? c → P ⪯ S a’? d → Q
  using A
apply (auto simp: Simulation-def read-def relation-of-iPrefix design-defs rp-defs)
apply (erule-tac x = ba in allE, erule-tac x = ca in allE, simp)
apply (rule-tac b = Simul S ba in comp-intro)
apply (auto split: cond-splits simp: Simul-def alpha-rp.defs do-I-def select-def B)
done

lemma Read1-Sim:
  assumes A: P ⪯ S Q and B: □ A. (d A) = c (S A)
  shows a’? c’ : s → P ⪯ S a’? d’ : s → Q
  using A
apply (auto simp: Simulation-def read1-def relation-of-iPrefix design-defs rp-defs)
apply (erule-tac \( x = ba \) in \( \text{allE} \), erule-tac \( x = ca \) in \( \text{allE} \), simp)
apply (rule-tac \( b = \text{Simul} S ba \) in \( \text{comp-intro} \))
apply (auto split: cond-splits simp: \( \text{Simul-def} \) alpha-rp.defs do-I-def select-def B)
done

lemma Read1S-Sim:
  assumes \( A: \ P \preceq S Q \) and \( B: \bigwedge A. (d A) = c (S A) \) and \( C: \bigwedge A. (s' A) = s (S A) \)
  shows \( a' \preceq s' \rightarrow P \preceq S a' \preceq s' \rightarrow Q \)
  using \( A \)
apply (auto simp: Simulation-def read1-def relation-of-iPrefix design-defs rp-defs)
apply (erule-tac \( x = ba \) in \( \text{allE} \), erule-tac \( x = ca \) in \( \text{allE} \), simp)
apply (rule-tac \( b = \text{Simul} S ba \) in \( \text{comp-intro} \))
apply (auto split: cond-splits simp: \( \text{Simul-def} \) alpha-rp.defs)
done

lemma Write-Sim:
  assumes \( A: \ P \preceq S Q \) and \( B: \bigwedge A. (d A) = c (S A) \)
  shows \( a' \rightarrow P \preceq S a' \rightarrow Q \)
  using \( A \)
apply (auto simp: Simulation-def write1-def relation-of-oPrefix design-defs rp-defs)
apply (erule-tac \( x = ba \) in \( \text{allE} \), erule-tac \( x = ca \) in \( \text{allE} \), simp)
apply (rule-tac \( b = \text{Simul} S ba \) in \( \text{comp-intro} \))
apply (auto split: cond-splits simp: \( \text{Simul-def} \) alpha-rp.defs)
done

lemma Hide-Sim:
  assumes \( A: \ P \preceq S Q \)
  shows \( (P \setminus cs) \preceq S (Q \setminus cs) \)
apply (auto simp: Simulation-def relation-of-Hide design-defs rp-defs Simul-def alpha-rp.defs)
apply (rule-tac \( b = \text{Simul} S ba \) in \( \text{comp-intro} \))
apply (split cond-splits)+
apply (auto simp: \( \text{Simul-def} \) alpha-rp.defs Simulation-def dest!: \( \text{dest}!: A[\text{simplified}, \text{rule-format}] \) Skip-Sim[\text{simplified}, \text{rule-format}]])
apply (rule-tac \( x = s \) in \( exI \), auto simp: \( \text{diff-tr-def} \))
done

lemma lfp-Siml:
  assumes \( A: \bigwedge X. (X \preceq S Q) \rightarrow ((P X) \preceq S Q) \) and \( B: \text{mono} P \)
  shows \( (\text{lfp} P) \preceq S Q \)
apply (rule lfp-ordinal-induct , auto simp: B A)
apply (auto simp add: \( \text{Simulation-def} \) Sup-action \( \text{relation-of-bot} \) relation-of-Sup[\text{simplified}])
apply (subst (asm) \( \text{CSP-is-rd} \) OF relation-of-CSP)
apply (auto simp: rp-defs fun-eq-iff Simul-def alpha-rp.defs decrease-def split: cond-splits)
done
lemma Mu-Sim:
assumes A: \( \forall X Y. X \preceq_S Y \Rightarrow (P X) \preceq_S (Q Y) \)
and B: mono P and C: mono Q
shows \( (\text{lfp } P) \preceq_S (\text{lfp } Q) \)
apply (rule lfp-Siml, drule A)
apply (subst lfp-unfold, simp-all add: B C)
done

lemma bot-Sim: \( \text{bot} \preceq_S \text{bot} \)
by (auto simp: Simulation-def rp-defs Simul-def relation-of-bot alpha-rp-defs split: cond-splits)

lemma sim-is-ref: \( P \sqsubseteq Q = P \preceq (\text{id }) Q \)
apply (auto simp: ref-def Simulation-def Simul-def alpha-rp-defs)
apply (erule-tac x)
apply (erule-tac x)
apply (erule-tac x)
apply (erule-tac x)
done

lemma ref-eq: \( ((P::\{s::ev-eq,'b\} action) = Q) = (P \sqsubseteq Q \& Q \sqsubseteq P) \)
apply (erule-tac)
apply (erule-tac)
apply (erule-tac)
done

lemma rd-ref:
assumes A: \( R (P \parallel Q) \in \{ p. \text{is-CSP-process } p \} \)
and B: \( R (P' \parallel Q') \in \{ p. \text{is-CSP-process } p \} \)
and C: \( \forall a b. P (a, b) \Rightarrow P' (a, b) \)
and D: \( \forall a b. Q' (a, b) \Rightarrow Q (a, b) \)
shows \( (\text{action-of } (R (P \parallel Q))) \subseteq (\text{action-of } (R (P' \parallel Q'))) \)
apply (auto simp: rd-def)
apply (erule-tac)
apply (erule-tac)
done

lemma rd-impl:
assumes A: \( R (P \parallel Q) \in \{ p. \text{is-CSP-process } p \} \)
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 \( \text{Sim} \).

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 \( \forall v \bullet (\text{Schema FIG.Init}'; X) \)

\[ \mu X \bullet (((\text{req} \rightarrow (\text{Schema FIG.Out}'))'; \text{out}!'(\text{hd} o v) \rightarrow \text{Skip})) \]
\[ \Box (\text{ret}?'x \rightarrow (\text{Schema FIG.Remove}'))'; X) \]

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 \( \forall v \bullet (\text{Schema DFIG.Init}'; X) \)
\[\mu X \cdot ((\langle reg \to (\text{Schema DFIG}.\text{Out}) \rangle \langle out \to (\text{Schema DFIG}.\text{Remove})) \rangle \langle x \to (\text{Schema DFIG}.\text{Remove})) \rangle X)\]

**definition Sim where**

Sim A = FIG-alphabet.make (DFIG-alphabet.v A) (DFIG-alphabet.x A)

\(\mathbf{let} \quad \{ a. a < (DFIG-alphabet.max A) \land a \notin (DFIG-alphabet.retidS A) \}\)  

**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) \preceq Sim (Schema DFIG.Init)**

apply (auto simp: Sim-def Pre-def design-defs DFIG.Init-def FIG.Init-def rp-defs)

apply (rule-tac x = A \(|\max := 0, \text{retidS} := \{\}\) in exI, simp)

done

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

apply (rule Schema-Sim)

apply (auto simp: Pre-def DFIG-alphabet.defs FIG-alphabet.defs)

apply (rule-tac x = a \(|\max := [DFIG-alphabet.max a], \max := (Suc (DFIG-alphabet.max a))\), 

\(\text{retidS} := \text{retidS} a - \{DFIG-alphabet.max a\}\) in exI, simp)

apply (rule-tac x = a \(|\max := [DFIG-alphabet.max a], \max := (Suc (DFIG-alphabet.max a))\), 

\(\text{retidS} := \text{retidS} a - \{DFIG-alphabet.max a\}\) in exI, simp)

done

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

apply (rule Schema-Sim)

apply (auto simp: Pre-def DFIG-alphabet.defs FIG-alphabet.defs)

apply (clarsimp simp add: DFIG.Remove-def FIG.Remove-def)

apply (rule-tac x = a \(|\\text{retidS} := \text{insert} (hd (DFIG-alphabet.x a)) (\text{retidS} a)\) in exI, simp)

apply (auto simp add: DFIG.Remove-def FIG.Remove-def)

done

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

by (auto simp: DFIG.DFIG-def FIG.FIG-def mono-Seq SimRemove SimOut SimInit Sim-def DFIG-alphabet.defs)

introl: Var-Sim Seq-Sim Mu-Sim Det-Sim Write0-Sim Write-Sim Read-Sim Skip-Sim)

done

end
References


