RAPPORT DE RECHERCHE

L R I

REPRESENTING CIRCUS OPERATIONAL SEMANTICS IN ISABELLE/HOL

FELIACHI A

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

08/2011

Rapport de Recherche N° 1544

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

Representing *Circus* Operational Semantics in Isabelle/HOL

Abderrahmane FELIACHI
Abderrahmane.Feliachi@lri.fr

Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France

Résumé *Circus* est un langage de spécification qui permet de spécifier des structures de données et des comportements complexes. La sémantique dénotationnelle et opérationnelle de *Circus* ont été définies précédemment, et une théorie de test a été proposée sur la base de ces sémantiques. Nous avons proposé, dans un travail précédent, une représentation cohérente et logique de la sémantique dénotationnelle dans le prouveur de théorèmes Isabelle/HOL [5]. Dans ce rapport, nous décrivons notre représentation des règles de la sémantique opérationnelle de *Circus* dans Isabelle/HOL. Nous discutons également de problèmes et de décisions relatifs à cette représentation. Nous proposons une formalisation de l'essentiel de la théorie de test de *Circus*, avec pour objectif le développement d'un outil de génération de tests.

Abstract *Circus* is a specification language that allows to specify complex data and complex behaviour. The denotational and operational semantics of *Circus* were previously defined, and a testing theory was proposed using these semantics. We proposed, in a previous work, a consistent and logically safe representation of the denotational semantics in the Isabelle/HOL theorem prover [5]. In this report, we describe our representation of the *Circus* operational semantics rules in Isabelle/HOL. We also discuss some problems and decisions related to this representation. We propose a formalisation of the main part of the *Circus* testing theory, in the perspective of a test generation framework.

Keywords: *Circus* operational semantics, Specification-based testing, Theorem prover-based testing, Isabelle/HOL.

1 Introduction

Circus is a formal specification language developed in the university of York [10], which integrates the notions of states and complex data types (in a Z-like style) and communicating parallel processes inspired from CSP. Moreover, the language comes with a formal notion of refinement and allows to take into account abstract specifications and their transitions to models of programs. *Circus* has a denotational semantics [9], which is based on UTP [7].

This report describes the advancement of the definition of testing strategies based on *Circus* and their implementation with the HOL-TestGen system [2], which is developed as an extension of the Isabelle/HOL theorem prover [8].

A testing theory of *Circus* has been given in [4]. It is built essentially on three sets called *cstraces*, *csinitials* and $\overline{csinitials}$ and on an operational semantics of *Circus* . The tests are defined using these three notions. The *cstraces* is the set of all constrained symbolic traces (event lists) a process may perform. The *csinitials* is the set of all constrained symbolic events the process may perform after performing a given trace. The $\overline{csinitials}$ is the set of constrained symbolic events the process should not perform after performing a given trace. Each element of these three sets is defined over a set of symbolic variables (values), and is associated with a constraint defined over these symbolic variables.

The definition of the *cstraces*, *csinitials* and $\overline{csinitials}$ is based on the operational (and denotational) semantics of *Circus*. We already have developed a representation of the denotational semantics of *Circus* in the Isabelle theorem prover. In order to represent the three sets above, we had then to represent the operational semantics of *Circus*. The operational semantics is defined with a set of rules over *Circus* processes given in [4].

The main results in this report are: the description of the transposition in Isabelle/HOL of the operational semantics of *Circus* with some related problems and decisions; the presentation of a test generation process for the tests defined in [4], which will serve as a basis for the development of various testing strategies from *Circus* specifications.

The report is organised as follows: in Section 2 we explain how a theorem prover can be used for test generation, and the principles of the HOL-TestGen system. Section 3 briefly recalls some technicalities on the UTP theories in Isabelle/HOL (the complete presentation can be found in [5]). Section 4 is the core of the report. In this Section we introduce our representation of the *Circus* operational semantics rules, including the definition of the notion of configurations: this notion captures in a unified logical way, abstract states, constraints and continuations. In Section 5 we describe our *cstraces* generation technique with a small example. Finally, Section 6 explains how *csinitials* and $\overline{csinitials}$ are generated.

2 Theorem prover based test generation

Test generation is subject of many research projects, and several approaches have been proposed. One important approach is the use of theorem provers for

test generation. The use of theorem provers is motivated by the fact that such kind of systems is verified and logically safe. In addition, test generation systems can greatly benefit from the automatic and interactive proof techniques to define automatic and interactive test generation techniques. Our work belong to this category of approaches. As mentioned in the introduction, we use the Isabelle theorem prover to define our theories and our test generation approach is based on the HOL-TestGen system.

HOL-TestGen [2] is a test generation system based on the Isabelle theorem prover. The main goal of this system is to use the facilities offered by Isabelle to help test generation. The principle is rather simple: the prover can be seen as a transition system. The states of the system corresponds to some formula. The transitions correspond to formula transformation. The initial state (formula) is called test specification. It is introduced into the system as a proof goal. The system will try to prove this theorem using different techniques and should fail. The system will then stay in an intermediate state (normal form) containing all the test cases.

The way of writing the test theorem is very important. For example if we want to generate tests for a program P, one test theorem will be: $\forall\, t \in \mathit{Tests}(P) \rightarrow \mathit{prog}(t)$ where *Tests* is the test generation method, t is a test and *prog* is a free variable that makes the system fail to prove the theorem. The system will try to solve this problem by simplifying the definition of *Tests*, and the resulting cases are given to *prog*. The premises will contains the test conditions, and the conclusion will contain the actual tests (protected by *prog*). The non-feasible cases are automatically removed, since they have a false premise and the system can prove it and remove these cases. The premises are simplified using case splitting and elimination rules defined for that purpose. The normal form that defines the tests contains a set of (sub-)theorems: the premises describe a constraint over the free variables used in the conclusion, i. e. some test case.

In our case, the test theorems will be written in the same way and the operational semantics rules will be used to generates the tests.

3 Isabelle/HOL and UTP

3.1 HOL Records

A HOL record can be functionally seen as a binding "$\mathit{name} \mapsto \mathit{value}$" characterized by two functions "*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. An instance of these functions is defined for each variable in the record. The name of the variable is used to distinguish the different instances: for the *select* function the name is used directly and for the *update* function the name is used as a prefix e.g. for a variable named "x" the names of the select and update functions are respectively x and x_update.

3.2 Predicates

An HOL predicate is a boolean function over some HOL free variables e.g. $P =_{def} \lambda\, x\, y.\, x = y$ and the type of P is "$\alpha \to \alpha \to bool$". It corresponds also to a set of possible values for the free variables.

A HOL relation is a predicate over some pair "$\alpha \times \beta \to bool$".

We define a UTP predicate as a set of record instances (bindings) of type "$\alpha\ set$", which is equivalent in HOL to "$\alpha \to bool$". The variable values are given by the *select* function e.g. for a record type A that contains the variables x and y, the expression $P =_{def} x = y$ of type "$A \to bool$" is a UTP predicate.

A UTP relation is a pair $(\alpha P, P)$, with a predicate P and its alphabet αP. In our representation, the alphabet is encoded internally with the record definition, and the predicate is a UTP predicate. The homogeneous relations (with the same input and output alphabet) are defined by a HOL relation over two instances of the same record type e.g. the relation $x' = x + 1$ is encoded by $\lambda(A, A').x\ A' = (x\ A) + 1$ of type $T \times T \to bool$ (T is a record type).

A UTP condition is a relation that contains only input (or only output) variables. As we represent relations by a set of pairs of bindings, a condition is represented by a set of bindings. In our theory, the conditions and the relations have different types, consequently the operators defined for relations cannot be applied for conditions (new operators may be defined in this case).

Hereafter, the *UTP theory* will refer to our representation of the UTP constructs defined in [5]. This theory contains definitions for *predicates*, *relations*, *designs*, *reactive processes* and *CSP processes*.

3.3 Alphabets

The main concern of our UTP theory is to represent "built-in" bindings for alphabets in a typed way. As a consequence, alphabets can be represented as records (in the sense of SML or Isabelle). The advantage of this representation is that any element of the alphabet is associated with a general HOL type. The inconvenience of this representations is that variables cannot be introduced "on the fly", they must be known statically i.e. at type inference time.

For certain constructs appearing in UTP and *Circus* this might appear as a restriction, since it is a common exercise in the UTP community to introduce "on the fly" (fresh) variables in logical or operational rules.

Our approach to represent "on the fly" variables is motivated by the classical and in compiler construction well-known distinction of the set of statically known variables names (with a given type) and the actual variables, whose number is unknown statically. This means that a variable x of type τ in the UTP is represented in the *Circus* operational semantics by a binding of the form $(x \rightsquigarrow \tau\ list)$. Of course, since variables underly a scoping discipline, the latter is represented by a stack discipline on variable instances (see 4.4).

4 *Circus* Operational Semantics

The configurations of the transition system for the semantics of *Circus* actions are triples $(c \mid s \models A)$ where c is a constraint (a UTP condition) over the symbolic variables in use, s an assignment of values to all *Circus* variables in scope (a UTP condition over output variables), and A a *Circus* action. The transition rules over configurations have the form: $(c_0 \mid s_0 \models A_0) \xrightarrow{e} (c_1 \mid s_1 \models A_1)$, where e is a label i.e. a pair (channel $\times$ symbolic variable) or ϵ.

The representation of constraints, state, actions and labels is given below as well as the corresponding transition rules.

4.1 Constraints

As mentioned in the last section (3.3), our UTP theory does not allow dynamic variable introduction. However, this feature is important for constraints since symbolic variables are introduced dynamically in the *Circus* operational semantics rules. One possible solution is to introduce symbolic variables as HOL variables; this will also allow simplifications when solving the constraints. Constraints are represented by HOL predicates over these symbolic variables. A syntactic type called "Constraint" is defined to avoid the simplification of the constraints and to allow transformations on constraints (DNF for example). Two functions are defined: "*freeze*" that transforms predicates to "Constraints" and "*eval*" that transforms the "Constraint" to a predicate.

The constraints are built in the operational semantics rules in two ways, either implicitly or explicitly. By explicit I mean the conjunction of equalities or type restrictions. In this case every equality is transformed (protected) to the syntactic type and a syntactic conjunction is used to keep trace of the conjunction. The type restrictions are not considered because type conformity is enforced internally by the system. The implicit (and more interesting) way is to build constraints from predicates that come from some *Circus* constructs such as specification statements, Schema expressions or Guards. To deal with those cases, a syntactically equivalent definition of these constructs should be provided.

Since we are using HOL predicates and not UTP conditions to represent the constraints, expressions like "(s; v = e)" cannot be expressed directly, we should define a semantically equivalent expressions according to our representation of the state; this point is discussed in section 4.4.

4.2 Actions

In our UTP theory [5] we provide a definition of the *Circus* action type as the set of reactive CSP processes (CSP-healthy reactive designs). Our theories contains also definitions for basic *Circus* actions (e.g. Skip) and *Circus* operators over actions (e.g. ;). The *Circus* action type is given by "$(\Theta, \sigma)\, action$" where "$\Theta$" and "$\sigma$" are polymorphic type parameters for "*channels*" and "*alphabet*"; these type parameters are instantiated for concrete processes.

In the configurations, the third component corresponds to a *Circus* action of type "(Θ, σ) *action*" as described above. The denotational semantics of the actions is not used by the rule inference system, so they will be seen as syntactic entities. When the operational semantics rules are applied, the system matches syntactically the concrete actions with the actions of the rules to find which rules are applicable. The rules contain also some special actions that are not part of the *Circus* language, but are introduced by the operational semantics (e.g. $\boxplus$). These actions are considered here as syntactic entities since it turns out that their denotational semantics is not used in the operational semantics rules.

To represent the recursion operator "μ" over actions, we use the universal "μ" defined in HOLCF (the extension of Church's Higher-Order Logic with Scott's Logic for Computable Functions). This operator is inherited from the "*pcpo class*" under some conditions, and all theorems defined over this operator can be reused as well (e.g. the unfolding theorem, see 4.5).

4.3 Labels

All the transitions over configurations are decorated with labels to keep a trace of the events that the system may perform. A label may refer to a communication with an input or output value, a synchronization (without communication) or an internal (silent) transition. In *Circus* an input communication is represented by "*chan*?*var*" and an output communication by "*chan*!*val*" where *chan* is a channel, *var* is a variable and *val* is a value. A synchronization is represented by "*chan*" where *chan* is a channel. The labels keep track of these events as "*chan*?*symbvar*" for inputs, "*chan*!*symbvar*" for outputs and "*chan*" for synchronizations (symbvar is a symbolic variable). Silent transitions are labeled by a special label "ϵ", that corresponds to an internal evolution of the system.

The channels type is represented in our theory as a *datatype* called "*Channels*"; every channel is a function defined from the channel type to the type *Channels*. For example, if we have two channels in our process $chan1$ (that communicates natural values) and $chan2$ (without communication), the *Channels* type will be defined by $datatype\ Channels = chan1\ nat \mid chan2$.

Labels are also defined as a *datatype* as follow:
$datatype\ TransitionLabels = Inp\ Channels \mid Out\ Channels \mid Sync\ Channels \mid \epsilon$. The transitions are labeled by one of these labels, the pair $(channel, symbvar)$ is represented by the function application $channel(symbvar)$. For example, the label $chan1!w_0$ is represented by $out(chan1(w_0))$ where w_0 is a symbolic variable.

4.4 State

The state records, after each rule transition, the mapping between the variable names and their current symbolic values. This notion is usually used and well-know in compilation as a stack mechanism. We present, in the following, our representation of the stack, with variable introduction, removal and nested scopes.

The representation In the operational semantics the symbolic state is represented by a UTP condition over output (dashed) variables. Concretely, it consists of an assignment of values to all *Circus* variables. The state is updated in the operational semantics rules by composing one of the three relations: $var\, x$, $end\, x$ or $x := e$, that corresponds to variable scope delimiters and variable value update. The state is also used to build the constraints, but there it is used only in expressions of the form "$s;\ v = expr$" that corresponds to an evaluation of the expression $expr$ in the context (binding) of s.

The symbolic state is conceptually the set of all the bindings (concrete states) of (*Circus*) variables of the form (variable $\mapsto$ symbolic variable). In our representation, symbolic variables corresponds to some HOL variables bounded by a meta universal quantifier "Λ". The (*Circus*) actions variables are the only free variables in the state; the binding is then concrete. The set of mappings is represented by the meta quantification over symbolic variables.

As a consequence, instead of using the set of all bindings (variable $\mapsto$ symbolic variable) we use a function (symbolic variables $\rightarrow$ binding). Since the symbolic variables are HOL variables, the state corresponds to a single symbolic binding (variable $\mapsto$ symbolic variable). The state can be updated by usual bindings update functions and the evaluation of expressions (in the constraints part) is done using select functions over this binding. This new representation will avoid to the automatic simplifier a lot of superfluous simplifications.

Variable scopes We explained in the introduction that the main idea of representing alphabets as records raises the limitation that alphabets cannot be modified. Consequently, introducing, removing or renaming variables cannot be expressed. This limitation forced us to find a new representation of variables in the state that preserves the semantic of variable scopes and assignments in *Circus*.

Since the state is a mapping between variables and symbolic values, a nested scope of a variable should not affect this mapping, and the value of a global variable should be the same before *var* and after *end* of a local variable. The solution is to use a stack, in this case the state is no more a mapping variable/value but rather variable/(vector of values), which keeps track of nested statements. For example, if we have a *Circus* action containing $var\, x\,;\ x := 1;\ var\, x\,;\ x := 2$, the state must contain the mapping between x and the vector [1,2]. To implement the vector of values we used lists, the first element of the list is the value of the variable in the current scope, *var* and *end* correspond to adding and removing the first item of this list. The following example lists the mappings variable/values in different states.

$var\ x$	$x \mapsto [?]$
$var\ x\ ;\ x := 1$	$x \mapsto [1]$
$var\ x\ ;\ x := 1;\ var\ x$	$x \mapsto [?, 1]$
$var\ x\ ;\ x := 1;\ var\ x\ ;\ x := 2$	$x \mapsto [2, 1]$
$var\ x\ ;\ x := 1;\ var\ x\ ;\ x := 2;\ end\ x$	$x \mapsto [1]$
$var\ x\ ;\ x := 1;\ var\ x\ ;\ x := 2;\ end\ x\ ;\ end\ x$	$x \mapsto []$

This representation affects slightly the semantics of assignments and evaluations over the state in our encoding. The value of a variable is no more the corresponding value in the mapping but the first element of it. The *select* and *update* functions are modified to preserve their original semantics. The definitions of *var*, *end*, *select* and *update* are given as follow:

- $var\ x\ e \quad \equiv \quad x := e \# x$
- $end\ x \quad \equiv \quad x := tl\ x$
- $x\_update'\ e \equiv \quad (hd\ x) := e$
- $x\_select' \quad \equiv \quad (hd\ x)$

Where # is the list constructor, *tl* gives the tail of a list and *hd* gives the head of a list. These operators are only defined and used in our encoding of the operational semantics; their semantics differ from the usual UTP operators.

This modification will affect also the actions because the state uses the same alphabet as the actions. The problem is not related to the alphabet itself, but to the bindings, since our representation does not make a difference between alphabets and bindings (names and values). Since the binding is now referring to a list of values, we cannot anymore use the same binding for the actions. To deal with this problem we can either change the way we use our variables in the actions or use two different alphabets (bindings) for state and actions and introduce transformations between them.

In the first solution we can use the same alphabet (binding) in the sate and the actions, but we should use the new definitions of *select* and *update* functions (*select′* and *update′*) in the actions (including schema expressions). This solution is simple and easy to implement and the modifications are transparent.

The second solution is to use different mappings in the state and the actions, by keeping the binding (variable $\mapsto$ value) for the actions. The alphabets of actions and state are then different because in our UTP theory names and binding are correlated. We can define a new alphabet based on the first one but with a binding (variable $\mapsto$ list of values). In this case the *select* and *update* functions are used in the actions while *select′* and *update′* functions are used for the state. Transformations between these functions should be defined and used in the rules that manipulate variables (e.g. the assignment rule).

4.5 Operational Semantics rules

The operational semantics is defined by a set of (inductive) inference rules of the form:

$$\frac{A}{(c_0 \mid s_0 \models A_0) \xrightarrow{e} (c_1 \mid s_1 \models A_1)}$$

Where $(c_0 \mid s_0 \models A_0)$ and $(c_1 \mid s_1 \models A_1)$ are configurations, e is a label.

The soundness of these rules w.r.t the denotational semantics has to be proved: it is the subject of another on-going work in the *Circus* group. In our work, we assume the soundness of these rules, we can then introduce them as axioms in our theories.

In Section 2 we explained that the theorem prover based test generation relies essentially on elimination rules, these rules should be introduced in our theories. It is not safe to introduce elimination rules as axioms: we should be able to infer them form introduction rules. The *inductive set* definition is a convenient way to represent the operational semantics. The introduction rules are used to define the inductive set, and the system will automatically generate and induction rule over these rules. The system offers also the possibility to generate elimination rules using this induction rule. Thus, we only need to provide some knowledge on the processes from the denotational semantics.

In the definition of inductive sets, types have to be explicitly specified. This could be problematic in our case, since the variable types are not known a-priori. This problem can be avoided by adding some parameters to our inductive set. We have explored two different solutions depending on these parameters. The first solution is to give as parameters the variable select and update functions, this functions will be instantiated for each variable in the concrete alphabet. This solution requires many changes on the rules, so it may make rules unreadable or introduce inconsistence.

We opted for a second solution, that consists of passing variable types as parameters. This solution is more transparent, the rules are not changed and type inference is done automatically for each concrete variable type. Once the introduction and elimination rules are defined, we can introduce the definitions of *cstraces*, *csinitials* and $\overline{csinitials}$.

4.6 Representing the rules

In this section we consider all the rules of the operational semantics given in [4] and we present their corresponding representation in our theory. The rules are written as introduction rules but, as seen above, the test generation uses elimination rules to simplify the premises. As a consequence, elimination rules should be derived from the introduction rules of the operational semantics. We describe here our representation of the operational semantics rules as introduction rules and sketch some elimination rules in the next subsection.

We provide a representation for the main operational semantics rules. However, some rule are not represented in our theory because they are irrelevant for our work or they are not covered yet at this stage of work. The irrelevant rules concerns essentially divergent behavior, which is not considered in the testing theory. The represented rules are labeled by (+).

Rules (1) and (2) of [4] are not represented because they are defined on processes, in our work we define only rules for actions.

The assignment rule (3) is very tricky because both the state and the constraint are updated. If we are using the same alphabet (binding) for the state and the actions, the expression e contains only $select'$ functions and can be evaluated directly in the state (binding). If the alphabet is different, we need to transform every x_select function in e to the corresponding x_select' function; after that the evaluation can be made in the current state. In both cases the state is updated by a v_update' function. Note that w_0 is the HOL free variable that corresponds to the symbolic variable. The application of the expression e to the current state s replaces in e the variables by their corresponding symbolic variables in the state s.

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\epsilon} (c \wedge (s;\ w_0 = e) \mid s;\ v := w_0 \models \mathit{Skip})} \tag{3}$$

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\epsilon} (c \wedge \mathit{freeze}\,(w_0 = e\,s) \mid v\_\mathit{update}'\ w_0\ s \models \mathit{Skip})} \tag{+}$$

For schema expressions we need to evaluate the *Pre* operator in the current state, then the new constraint is built by evaluating the schema operation in the updated state. Note the we do not need any substitution because the variables are already bound to fresh symbolic variables in the new state.

$$\frac{c \wedge (s;\ \mathrm{pre}\ \mathit{Op})}{(c \mid s \models \mathit{Op}) \xrightarrow{\epsilon} (c \wedge (s;\ \mathit{Op}[w_0/v']) \mid s;\ v := w_0 \models \mathit{Skip})}\quad v = \mathit{out}\alpha s \tag{4}$$

$$\frac{c \wedge (\mathrm{pre}\ \mathit{Op}(s))}{(c \mid s \models \mathit{Op}) \xrightarrow{\epsilon} (c \wedge \mathit{freeze}(\mathit{Op}(v\_\mathit{update}'\ w_0\ s)) \mid v\_\mathit{update}'\ w_0\ s \models \mathit{Skip})} \tag{+}$$

For outputs the state is not changed, only the constraint is updated in the same way as for assignment. The transition is labeled by the corresponding label "$\mathit{Out}\,(d\ w_0)$".

$$\frac{c}{(c \mid s \models d!e\ \rightarrow\ A) \xrightarrow{d!w_0} (c \wedge (s;\ w_0 = e) \mid s \models A)} \tag{5}$$

$$\frac{c}{(c \mid s \models d!e\ \rightarrow\ A) \xrightarrow{\mathit{Out}\,(d\,w_0)} (c \wedge \mathit{freeze}(w_0 = e\,s) \mid s \models A)} \tag{+}$$

The input communication rule introduces a variable to the state with the assumption that this variable is not already in the state. With our representation of the state, we don't need to express this assumption because our definition of *var* takes into account this kind of nested scope (by using a stack). As mentioned in the introduction, type checking constraints are also removed, the type checking is done statically by the system. The transition is labeled by "$Inp\,(d\ w_0)$".

$$\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(c \mid s \models d?x : T \rightarrow A) \xrightarrow{d?w_0} (c \wedge w_0 \in T \mid s;\ \mathbf{var}\ x := w_0 \models \textbf{\textit{let}}\ x \bullet A)} \tag{6}$$

$$\frac{c}{(c \mid s \models d?x \rightarrow A) \xrightarrow{Inp\,(d\,w_0)} (c \mid \mathbf{var}\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

The variable introduction rule is similar to inputs rule. It defines a variable scope. Similarly, we do not need any dynamic type checking for the symbolic variable, its type is statically instantiated from the context (same type as x).

$$\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(c \mid s \models \mathbf{var}\ x : T \bullet A) \xrightarrow{\epsilon} (c \wedge w_0 \in T \mid s;\ \mathbf{var}\ x := w_0 \models \textbf{\textit{let}}\ x \bullet A)} \tag{7}$$

$$\frac{c}{(c \mid s \models \mathbf{var}\ x \bullet A) \xrightarrow{\epsilon} (c \mid \mathbf{var}\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

The first rule for variable scopes (8) is kept unchanged.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \textbf{\textit{let}}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{\textit{let}}\ x \bullet A_2)} \tag{8}$$

For the second rule (9), our *end* function (see section 4.4) is used to close the scope of the variable x.

$$\frac{c}{(c \mid s \models \textbf{\textit{let}}\ x \bullet \mathit{Skip}) \xrightarrow{\epsilon} (c \mid s;\ \mathbf{end}\ x \models \mathit{Skip})} \tag{9}$$

$$\frac{c}{(c \mid s \models \textbf{\textit{let}}\ x \bullet \mathit{Skip}) \xrightarrow{\epsilon} (c \mid \mathbf{end}\ x\ s \models \mathit{Skip})} \tag{+}$$

The sequence rules (10 and 11) and the internal choice rules (12) are unchanged since they don't explicitly contain any variables manipulation as in rule (8).

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1 \,;\, B) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \,;\, B)} \tag{10}$$

$$\frac{c}{(c \mid s \models \mathit{Skip} \,;\, A) \xrightarrow{\epsilon} (c \mid s \models A)} \tag{11}$$

$$\frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\epsilon} (c \mid s \models A_1)} \qquad \frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\epsilon} (c \mid s \models A_2)} \tag{12}$$

The external choice rule (13) and local blocks rules (14, 15 and 16) are not changed.

$$\frac{c}{(c \mid s \models A_1 \,\Box\, A_2) \xrightarrow{\epsilon} (c \mid s \models (\mathsf{loc}\ c \mid s \bullet A_1) \boxplus (\mathsf{loc}\ c \mid s \bullet A_2))} \tag{13}$$

$$\frac{c_1}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet \mathit{Skip}) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A)) \xrightarrow{\epsilon} (c_1 \mid s_1 \models \mathit{Skip})}$$

$$\frac{c_2}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet \mathit{Skip})) \xrightarrow{\epsilon} (c_2 \mid s_2 \models \mathit{Skip})} \tag{14}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\epsilon} (c_3 \mid s_3 \models A_3)}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right)}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{\epsilon} (c_3 \mid s_3 \models A_3)}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \end{array}\right) \end{array}\right)} \tag{15}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l \neq \epsilon}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)) \xrightarrow{l} (c_3 \mid s_3 \models A_3)}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l \neq \epsilon}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)) \xrightarrow{l} (c_3 \mid s_3 \models A_3)} \tag{16}$$

The rules of parallelism (17, 18, 19, 20, 21 and 22) are not integrated in theory yet. The state merge operation $((\exists\, x_2' \bullet s_1) \wedge (\exists\, x_1' \bullet s_2))$ should be defined first.

$$\frac{c}{(c \mid s \models A_1 \,[\![\, x_1 \mid cs \mid x_2 \,]\!]\, A_2) \xrightarrow{\epsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c \mid s \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ c \mid s \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{17}$$

$$\frac{c_1 \wedge c_2}{\left(\begin{array}{l} c_1 \wedge c_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_1 \mid s_1 \mid x_1 \bullet \mathit{Skip}) \\ [\![cs]\!] \\ (\mathsf{par}\ c_2 \mid s_2 \mid x_2 \bullet \mathit{Skip}) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} c_1 \wedge c_2 \mid (\exists\, x_2' \bullet s_1) \wedge (\exists\, x_1' \bullet s_2) \\ \models \\ \quad \mathit{Skip} \end{array}\right)} \tag{18}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \epsilon \vee \mathit{chan}\, l \notin cs \quad c_1 \wedge c_2 \quad c_3 \wedge c_2}{\left(\begin{array}{l} c_1 \wedge c_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_1 \mid s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ c_2 \mid s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \wedge c_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_3 \mid s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ c_2 \mid s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{19}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \epsilon \vee \mathit{chan}\, l \notin cs \quad c_1 \wedge c_2 \quad c_3 \wedge c_1}{\left(\begin{array}{l} c_1 \wedge c_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_1 \mid s_1 \mid x_1 \bullet A_1) \\ \|[cs]\| \\ (\mathsf{par}\ c_2 \mid s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \wedge c_1 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_1 \mid s_1 \mid x_1 \bullet A_1) \\ \|[cs]\| \\ (\mathsf{par}\ c_3 \mid s_3 \mid x_2 \bullet A_3) \end{array}\right) \end{array}\right)} \tag{20}$$

$$\frac{\left(\begin{array}{l} (c_1 \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c_2 \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \\ \qquad\qquad\qquad\qquad\qquad\qquad \vee \\ (c_1 \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c_2 \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \\ \qquad\qquad\qquad\qquad\qquad\qquad \vee \\ (c_1 \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c_2 \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \end{array}\right) \quad d \in cs \quad c_1 \wedge c_2 \quad c_3 \wedge c_4 \wedge w_1 = w_2}{\left(\begin{array}{l} c_1 \wedge c_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_1 \mid s_1 \mid x_1 \bullet A_1) \\ \|[cs]\| \\ (\mathsf{par}\ c_2 \mid s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d!w_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_3 \wedge w_1 = w_2 \mid s_3 \mid x_1 \bullet A_3) \\ \|[cs]\| \\ (\mathsf{par}\ c_4 \wedge w_1 = w_2 \mid s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{21}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \quad (c_2 \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \quad d \in cs \quad c_1 \wedge c_2 \quad c_3 \wedge c_4 \wedge w_1 = w_2}{\left(\begin{array}{l} c_1 \wedge c_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_1 \mid s_1 \mid x_1 \bullet A_1) \\ \|[cs]\| \\ (\mathsf{par}\ c_2 \mid s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d?w_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ c_3 \wedge w_1 = w_2 \mid s_3 \mid x_1 \bullet A_3) \\ \|[cs]\| \\ (\mathsf{par}\ c_4 \wedge w_1 = w_2 \mid s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{22}$$

In guarded actions the guard g is evaluated in the current state, then "frozen" and added to the constraint. The expression $(g\ s)$ evaluates the guard g in the current state s, it replaces the variables with their corresponding symbolic variables.

$$\frac{c \wedge (s;\ g)}{(c \mid s \models g\ \&\ A) \xrightarrow{\epsilon} (c \wedge (s;\ g) \mid s \models A)} \tag{23}$$

$$\frac{c \wedge (g\ s)}{(c \mid s \models g\ \&\ A) \xrightarrow{\epsilon} (c \wedge \mathit{freeze}(g\ s) \mid s \models A)} \quad (+)$$

The hiding rules are not changed, the definition for *chan* is given as partial function from *Labels* to *Channels* as follow:

if $(\mathit{label} = \epsilon)$
  then undefined
  elseif $(\mathit{label} = \mathit{Inp\ channel} \vee \mathit{label} = \mathit{Inp\ channel} \vee \mathit{label} = \mathit{Inp\ channel})$
    then channel

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l \neq \epsilon \quad \mathit{chan}\ l \notin cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \setminus cs)} \quad (24)$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l = \epsilon \vee \mathit{chan}\ l \in cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{\epsilon} (c_2 \mid s_2 \models A_2 \setminus cs)} \quad (25)$$

$$\frac{c}{(c \mid s \models \mathit{Skip} \setminus cs) \xrightarrow{\epsilon} (c \mid s \models \mathit{Skip})} \quad (26)$$

The recursion rules are removed, a new rule is introduced to express the HOL μ unfolding. No process environment is needed and the variables renaming is avoided with the new state representation. The soundness is justified by the denotational semantics: μ is the least fixed point operator (see section 4.2).

$$\frac{c}{(c \mid s \models \mu\ \ X \bullet A, \delta) \xrightarrow{\epsilon} (c \mid s \models A, \delta \oplus \{X \mapsto A\})} \quad (27)$$

$$\frac{c}{(c \mid s \models X, \delta) \xrightarrow{\epsilon} (c \mid s \models \delta\ X, \delta)} \quad (28)$$

$$\frac{(c \mid s \models \mu\ \ X \bullet A(X)) \xrightarrow{l} (c1 \mid s1 \models A1)}{(c \mid s \models A(\mu\ \ X \bullet A(X))) \xrightarrow{l} (c1 \mid s1 \models A1)} \quad (+)$$

4.7 Elimination rules

For each operator, at least one elimination rule is introduced. These rules are essentially extracted from the introduction rules. An example of the elimination rules for *Skip* and non-deterministic choice are given below:

$$\frac{(c \mid s \models \mathit{Skip}) \xrightarrow{l} c1 \qquad \begin{array}{c}[\mathit{False}]\\ \vdots \\ Q\end{array}}{Q}\ \mathit{Skip}\mathcal{E}$$

$$\frac{(c \mid s \models A \sqcap B) \xrightarrow{l} c1 \qquad \begin{array}{c}[c1 = (c \mid s \models A) \quad l = \epsilon]\\ \vdots \\ Q\end{array} \qquad \begin{array}{c}[c1 = (c \mid s \models B) \quad l = \epsilon]\\ \vdots \\ Q\end{array}}{Q}\ \mathit{Ndet}\mathcal{E}$$

The first rule eliminates the transitions of *Skip* to *False*, because there is no introduction rules for *Skip*. This rule allows to stop the application of the rules since this subgoal will be removed. The second rule eliminates the transitions for $A \sqcap B$ and produces two continuations (subgoals). With a silent transitions, the process may perform A or B, corresponding to the introduction rules.

5 Generating *cstraces*

The *cstraces* is the set of all constrained symbolic traces a process may perform. A *cstrace* is a list of events associated to a constraint. Events are given by the labels of the operational semantics transitions. Some additional rules are defined in order to build those lists from the operational semantics rules. We introduce a relation noted "$\Longrightarrow$" and defined by :

$$\frac{}{c_1 \overset{[]}{\Longrightarrow} c_2} \qquad \frac{c_1 \xrightarrow{\epsilon} c_2 \quad c_2 \overset{st}{\Longrightarrow} c_3}{c_1 \overset{st}{\Longrightarrow} c_3} \qquad \frac{c_1 \xrightarrow{e} c_2 \quad c_2 \overset{st}{\Longrightarrow} c_3 \quad e \neq \epsilon}{c_1 \overset{e\#st}{\Longrightarrow} c_3} \qquad (*)$$

Given # is the list construction operation, and [] the empty list.

The *cstraces* set is then defined using this relation by:

$$\mathit{cstraces}(c_1, s_1, A_1) = \left\{ \begin{array}{l} st, c_2 \mid (c_1 \mid s_1 \models A_1) \overset{st}{\Longrightarrow} (c_2 \mid s_2 \models A_2) \\ \bullet\ (st, (\exists\, \alpha c_2 \setminus \alpha st) \bullet c_2) \end{array} \right\}$$

We introduced in our theories the introduction rules corresponding to (*) in the same way as for the operational semantics rules (using an inductive set definition). The *cstraces* definition is introduced using these rules. Starting from our rules, a trace generation tactic is defined to allow automatic trace generation.

5.1 Trace generation tactic

As explained in the first section, a test theorem is written for trace generation. This test theorem is given below:

$$\frac{\forall\, tr \in cstraces(c_1, s_1, A_1)}{Prog(tr)}$$

The premise is simplified using a trace generation tactic and resulting traces are stored by *Prog*. The trace generation tactic is described by the following algorithm:

Data: k : the maximum length of traces

Simplify *cstraces* by its definition;

while *length* $\leq k \wedge$ *more traces can be generated* **do**

  Apply the elimination rules of the "$\Longrightarrow$" relation on the current goal;

  Apply the elimination rules of the operational semantics on the resulting subgoals;

end

Algorithm 1: Trace generation tactic

The premise is first simplified using the definition of *cstraces*, the resulting premise contain only predicates over the "$\Longrightarrow$" relation. The application of the rules (*) on these predicates generates three possible continuations:

- The empty trace,
- A trace preceded by an ϵ,
- An event followed by a trace.

The first resulting subgoal allows the building of partial traces by giving empty continuations. The other two cases allow to build the traces element by element using the transition rule of the operational semantics for the first element and the "$\Longrightarrow$" relation for the continuation.

The elimination rules of the operational semantics are applied to the two last subgoals in order to instantiate the first element. The system will try to match the transition of the subgoal with the transitions of the operational semantics. If a matching is found, the transition is replaced by an instantiation of the configurations. If at least two matchings are found (e.g. internal choice), the system introduces a new subgoal for each possible matching. If no matching is found, the subgoal is removed because the system can prove it (False premise).

The rules are applied repeatedly until a given depth, or if there is no possible continuation. The list of subgoals corresponds to the list of all possible traces of a given depth. To see in details how the trace generation tactic works, we give the following example:

5.2 Example

If we want to generate the *cstraces* of the action "$sync \rightarrow Skip$", the test theorem is written:

$$\frac{\forall\, tr \in cstraces(true, s_0, sync \rightarrow Skip)}{Prog\ tr}$$

This theorems is simplified with the definition of *cstraces* and the result is:

$$\frac{\forall\, st\ c_2 \bullet \exists\, s_2\ A_2 \bullet (true \mid s_0 \models sync \rightarrow Skip) \stackrel{st}{\Longrightarrow} (c_2 \mid s_2 \models A_2)}{Prog\ (st, c_2)}$$

The application of the elimination rules of the relation "$\Longrightarrow$" gives three subgoals:

$$\frac{\forall\, st\ c_2 \bullet \exists\, s_2\ A_2 \bullet (true \mid s_0 \models sync \rightarrow Skip) \stackrel{[]}{\Longrightarrow} (true \mid s_0 \models sync \rightarrow Skip)}{Prog\ ([], true)}$$

$$\frac{\begin{array}{c}\forall\, st\ c_2 \bullet \exists\, s_2 ... \bullet (true \mid s_0 \models sync \rightarrow Skip) \stackrel{\epsilon}{\longrightarrow} (c_1 \mid s_1 \models A_1) \\ (c_1 \mid s_1 \models A1) \stackrel{st}{\Longrightarrow} (c_2 \mid s_2 \models A_2)\end{array}}{Prog\ (st, c_2)}$$

$$\frac{\begin{array}{c}\forall\, st\ c_2 \bullet \exists\, e\ s_2 ... \bullet (true \mid s_0 \models sync \rightarrow Skip) \stackrel{e}{\longrightarrow} (c_1 \mid s_1 \models A_1) \\ (c_1 \mid s_1 \models A1) \stackrel{st}{\Longrightarrow} (c_2 \mid s_2 \models A_2)\end{array}}{Prog\ (e\#st, c_2)}$$

The first case gives the empty trace, the last two cases introduces a predicate with the transition relation of the operational semantics. The application of the elimination rules of the operational semantics on the second rule will attempt to find an applicable transition rule with a label ϵ. The system will fail to find any applicable rule and the subgoal is removed (false premise). For the third subgoal, the system will find a rule that matches the premise and instantiate the unknown variables by their corresponding values. The result of this iteration contains two subgoals:

$$\frac{}{Prog\ ([], true)}$$

$$\frac{\begin{array}{c}\forall\, st\ c_2 \bullet (true \mid s_0 \models sync \rightarrow Skip) \stackrel{sync}{\longrightarrow} (true \mid s_0 \models Skip) \\ (true \mid s_0 \models Skip) \stackrel{st}{\Longrightarrow} (c_2 \mid s_2 \models A_2)\end{array}}{Prog\ ((sync)\#st, c_2)}$$

The second iteration concerns only the second case, after applying the first elimination rules, this case is replaced by the three possible cases. The application of the operational semantics elimination rule will fail to find any rule for *Skip*.

The two last cases will be removed and the final result will contain only two cases:

$$\overline{Prog\,([], true)}$$

$$\overline{Prog\,([sync], true)}$$

This proof state represents all the *cstraces* this system can perform.

6 Generating *csinitials* and $\overline{csinitials}$

The *csinitials* set is the set of all constrained symbolic events the system may perform after a given trace. The definition of *csinitials* is based on the *cstraces*:

$$csinitials(P, (st, c)) = \\ \{se, c_1 \mid (st \frown \langle se \rangle, c_1) \in cstraces(P) \wedge (c \wedge c_1) \bullet (se, c \wedge c_1)\}$$

given $(st, c) \in cstraces(P)$.

The generation of *csinitials* is done using the same tactic as for *cstraces*. The resulting subgoal gives a list of possible *csinitials* after a given *cstrace*. In order to generate tests for traces refinement relation, we need to introduce another set : $\overline{csinitials}$ [3,**?**]. This set contains all the constrained symbolic events the system should refuse to perform after a given trace. This set is defined using the *cstraces* and *csinitials*:

$$\overline{csinitials}(P, (st, c)) = \\ \{e, c_1 \mid (c_1 = c \wedge \neg \bigvee\{c_2 \mid (e, c_2) \in csinitials(P, (st, c))\}) \bullet (e, c_1)\}$$

provided $(st, c) \in cstraces(P)$

The generation of $\overline{csinitials}$ raises an important problem. The constraints of the elements are built from a set of *csinitials*, which is not easy to obtain since our tactic generates an enumeration of cases and not a set of elements. To solve this problem two directions were explored.

The source of our problem is the case splitting that results from our elimination rules. In order to obtain a set of elements, we need to avoid this splitting by merging the cases that can result to multiple subgoals (e.g. internal choice rules). As a consequence, many modifications should be done on the rules to correspond to the new representation. This solution is very complicated and may make the rules inconsistent. The automation of elimination and simplification rules becomes very hard with this solution.

The second direction we explored do not affect the rules and the generation tactic. A post-processing is applied to the results of the generation phase. We can gather in one subgoal a set of subgoals by applying a meta-tactic. The result of this meta-tactic is a conjunction of subgoals. For example, if we have two subgoals : "$x \in Prog$" and "$y \in Prog$", these subgoals are merged in on

subgoal :"$x \in Prog \wedge y \in Prog$". This subgoal can be simplified to give the set form: "$\{x, y\} \subseteq Prog$". This solution is very promising, it allows us to do some processing over the resulting set also. The $\overline{csinitials}$ will be generated by applying a transformation function over the set of resulting *csinitials*.

7 Conclusion

In this report we have described a representation of the *Circus* operational semantics in Isabelle/HOL. This representation is a continuation of our work on the formalisation of *Circus* and UTP [5]. We presented some problems related to the representation of configurations (states, actions, ...) and proposed well-founded solutions for them. The most important problem was the variable introduction in the state. As a solution, we defined a complete stack mechanism for the state variables, dealing with variable introduction, removal and nested scopes.

This report contains also the representation of the main part of the operational semantics rules. Those rules are slightly changed to fit the state representation. Another set of rules is introduced to allow the test generation automation. This set contains the elimination rules that match the introduction rules of the operational semantics.

In the last part of the report, we have explained with a simple example, our test generation tactics. We also provided some definitions used for this purpose.

In order to obtain a complete and consistent testing framework, our priority is to complete the remaining part of the operational (and denotational) semantics to cover the set of all the *Circus* constructs. The remaining rules concerns essentially those dealing with the parallel composition operator. Another important and urgent task is to complete the proof of some theorems, essentially those that allow us to use the fix-point operator in a safe way.

We can benefit from our formal representation of the *Circus* denotational semantics to complete the testing framework by proving the soundness of the elimination rules w.r.t the operational semantics introduction rules.

At this stage, we introduce tactics to generate symbolic tests for trace refinement in a consistent way (w.r.t the semantics of *Circus*). The continuation of this work will be to introduce new tactics for the other conformance relation based on *acceptances* defined in [4].

We usually obtain an infinite set of symbolic tests, some selection hypotheses should be defined to extract a finite subset of symbolic tests. This corresponds to the test criteria defined in [6].

Moreover, Symbolic tests generally corresponds to an infinity of concrete tests, where the symbolic values are instantiated according to the constraints. This will be done via the use of o constraint solver as done for instance in [1]. The choice of one ore more instantiations is another step in the testing strategy.

Acknowledgements

I would like to express my thanks to my supervisor Marie-Claude Gaudel for the valuable help in my work and in writing this report. I am also very grateful to Burkhart Wolff and Ana Cavalcanti for discussions which helped me develop the ideas put forward here.

References

1. Bentakouk, L., Poizat, P., Zaidi, F.: Checking the behavioral conformance of web services with symbolic testing and an smt solver. In: Springer (ed.) Tests & Proofs, TAP. LNCS, vol. 6706, pp. 33–50 (July 2011)
2. Brucker, A.D., Wolff, B.: Test-sequence generation with HOL-TestGen – with an application to firewall testing. In: Meyer, B., Gurevich, Y. (eds.) TAP 2007: Tests And Proofs, pp. 149–168. No. 4454 in Lecture Notes in Computer Science, Springer-Verlag, Zurich (2007)
3. Cavalcanti, A., Gaudel, M.C.: Testing for refinement in CSP. In: Formal Methods and Software Engineering, ICFEM 2007. Lecture Notes in Computer Science, vol. 4789, pp. 151–170. Springer Verlag (2007)
4. Cavalcanti, A., Gaudel, M.C.: Testing for refinement in Circus. Acta Informatica 48(2), 97–147 (2011)
5. Feliachi, A., Gaudel, M.C., Wolff, B.: Unifying theories in Isabelle/HOL. In: Unifying Theories of Programming 2010. Lecture Notes in Computer Science, vol. 6445, pp. 188–206. Springer Verlag, Shanghai, China (November 2010)
6. Gaudel, M.C.: Testing can be formal, too. In: TAPSOFT'95, International Joint Conference, Theory And Practice of Software Development. Lecture Notes in Computer Science, vol. 915, pp. 82–96. Springer Verlag, Aarhus, Denmark (1995)
7. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer-Verlag (2002)
9. Oliveira, M., Cavalcanti, A., Woodcock, J.: A denotational semantics for Circus. Electron. Notes Theor. Comput. Sci. 187, 107–123 (2007)
10. Woodcock, J.C.P., Cavalcanti, A.L.C.: The semantics of *circus*. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002: Formal Specification and Development in Z and B. Lecture Notes in Computer Science, vol. 2272, pp. 184—203. Springer-Verlag (2002)