Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

Model Checking Support

Deliverable Number: D33.1

Version: 0.4

Date: September 2013

Public Document

http://www.compass-research.eu

Contributors:

Adalberto C. Farias, UFPE
Alexandre C. Mota, UFPE
André Didier, UFPE
Jim Woodcock, UK

Editors:

Alexandre C. Mota, UFPE

Reviewers:

Luis Couto, AU
Richard Payne, NCL
Ken Pierce, NCL
Adrian Larkham, Atego

Document History

Ver	Date	Author	Description
0.1	13-06-2013	ACF	Initial document version
0.11	21-06-2013	ACF	Added the entire structure and the content of Introduction and User Guide sections
0.12	05-07-2013	ACF	Added some content in the Research Report section
0.13	07-08-2013	ACF	Research section was divided into two new sections. Added content to the CSP embedding section
0.14	13-08-2013	ACM	Added content to the sections introduction, lessons learned and CML embedding sections
0.15	14-08-2013	ACM,ACF	Added content to the related work section
0.2	20-08-2013	ACF	Content of User Guide section updated. Intermediated version.
0.21	26-08-2013	ACF,ACM	Changes in the CML embedding section
0.22	07-09-2013	ACF,ACM	Added content to the conclusions section
0.3	09-09-2013	ACF,ACM	Version for internal review
0.4	22-09-2013	ACF,ACM	Editing based on internal review

Contents

1 Introduction

Model checking [CGP99] is an automatic technique aiming to verify whether the relation $M \models f$[1] holds, where M is a model (in general some kind of Labelled Transition System, like a Kripke structure) of some formal language L and f is a temporal logic formula. The process algebra CSP [Ros10] has introduced another way of performing model checking, named refinement checking. The idea is to verify that the refinement relation $M_f \sqsubseteq M$ (M refines M_f) holds, where both M and M_f are models of a same language and M_f is the most non-deterministic model known to satisfy f.

Traditionally, a model checker is a tool that implements search procedures derived from the relation $M \models f$ (or from a refinement theory). These search procedures and representations of M and f are very specialized algorithms and data structures aiming at achieving the best space and time complexities possible. Because of this, it is not common to find model checkers for rich-state space languages (that use elaborate data structures). The best performing model checkers use very primitive data structures like natural numbers and arrays, and avoid sets, sequences, functions, etc.

This way of developing model checkers creates a gap between theory and practice, particularly for rich state languages that are more appropriate to model and reason about systems of systems. One of the problems is related to the guarantee of creating the right model M from the semantics (usually the Structured Operational Semantics, or simply SOS) of the language L. Another is whether the search procedure to check $M \models f$ (or $M_f \sqsubseteq M$) is correct. Finally, this restricts the kinds of formal languages that can have their own model checkers.

CML is the COMPASS Modelling Language, the first language specifically designed for modelling and analysing Systems of Systems (SoS). It is based on the following baseline languages: VDM [ABH+95], CSP [Ros10], and Circus [WC02]. It is fully described in deliverables D23.2 (syntax) and D23.3 (semantics). CML has a rich and heterogeneous semantics in the sense of combining several different paradigms with a rich state space. Developing a correct model checker for such a language is daunting. Thus instead of focusing on the best space and time complexities when creating such a model checker, we need first to focus on the most abstract and elegant implementation infra-structure to create a correct model checker for CML. This is the main goal of this deliverable.

The very recent technology developed by Microsoft Research, known as FORMULA [JSD+09] (Formal Modelling Using Logic Programming and Analysis),

[1] The model M (Design ou implementation) satisfies the property f (Specification).

seems to be an appropriate candidate to provide the right abstraction and elegance to implement a model checker able to handle the heterogeneity and rich-state features exhibited by CML (see a detailed discussion about this in Section 6). It is based on the Constraint Programming Paradigm [RvBW06] and Satisfiability Modulo Theory (SMT) solving provided by Z3 [DMB08].

The purpose of this deliverable is to present how a model checker for CML that conforms to its SOS was created, and how a feasibility study was performed to test the ability of FORMULA to capture and analyse CML specifications using the COMPASS CML tool.

As our model checker is provided through the COMPASS CML tool, we start this deliverable in Section 2 by presenting a user guide towards this tool (reusing some basic context of the COMPASS CML tool [CMLC13]). We cover installation procedures and requirements, usage of the tool and some illustrative examples. It is worth pointing out that the current implementation is platform dependent as FORMULA is available only for windows platforms. However, the plugin architecture (detailed in Section 5) allows extensions to invoke FORMULA remotely.

After the practical aspect of our contribution, we present the more theoretical contribution. As CML can be seen as a combination between a behavioural (CSP language) and state-based (VDM language) parts, we consider the effort to create a CSP model checker based on the FORMULA technology in Section 3. In this section we give a brief introduction to FORMULA, present the SOS of CSP (this is just to show how close the description in FORMULA is from its pure theoretical SOS counterpart), present details about CSP refinement checking and finally the model checker script written in FORMULA.

A CML model checker is the logical following step and it is considered in Section 4. We show how to incorporate the state aspect of VDM in the previously considered behavioural aspect of CSP. To this end, we present and discuss about the types supported by FORMULA and how the VDM Mathematical toolkit is supported. Some state-aspects are directly supported while others are interpreted. For those that are interpreted, we provide a FORMULA solution to a subset of them. The CML model checker has been implemented as an Eclipse plugin whose architecture and implementation are detailed in Section 5.

In Section 6 we discuss the advantages and disadvantages of creating a model checker for CML using the FORMULA framework and other alternatives.

This deliverable ends by presenting some related work in Section 7, and conclusions and future work in Section 8.

Complementary material, concerning the formal semantics of FORMULA and the relationship between first-order logic formulations of deadlock, livelock, nondeterminism and traces refinement analyses and FORMULA rules and queries, can be found in Appendices A and B, respectively. In Appendix C we present some key examples and the quantitative part of our feasibility study.

2 User Guide

This section provides essential information for the users of the CML model checker. Before using the model checker we suggest reading the main documentation about the entire COMPASS IDE tool [CMLC13]. This is useful to understand the resources provided by the IDE as well as to understand basic activities like creating CML projects, editing files, compilation errors and type checking errors, for example, as they have to be performed prior to the model checking itself.

2.1 Installation

The CML model checker is developed over the Microsoft FORMULA tool and GraphViz. The first is used as the main engine to analyse CML specifications whereas the second is used to show the counterexample found by the analysis.

The steps to install the CML model checker to work are listed as follows:

1. Download and install the Microsoft FORMULA tool. It is available at `http://research.microsoft.com/en-us/um/redmond/projects/formula/`. Although the tool is free, it requires Microsoft Visual Studio[2] is installed. This makes the current version of the CML model checker platform dependent as the underlying framework is from Microsoft.

2. Download and install the GraphViz software. Graphviz is open source graph visualization software. It allows several kinds of graphs to be written (in a text file) and graphical output generated in several formats to be presented. GraphViz is available at `http://www.graphviz.org/` and can be installed in several platforms. The CML model checker uses specifically the `dot.exe` program, which provides compilation from a textual description to several formats. We use the SVG format that is vectorial and accepted by most of Web browsers.

3. Download and install the COMPASS IDE tool. The COMPASS tool containing all features is available at `http://build.compass-research.eu/builds/compass-devel/`. We recommend to use the COMPASS-0.1.9-SNAPSHOT version.

[2] `http://www.microsoft.com/visualstudio`.

2.2 Using the CML model checker

This section introduces the CML model checker. We show how to invoke its functionalities and which components are available to the user.

The model checker functionalities are available through the CML Model Checker perspective (see Figure 1), or MC perspective, which is composed by the CML Explorer (1), the CML Editor (2), the Outline view (3), the internal Web browser (to show the counterexample when invoked) and two further specific views: the CML MC List view (4) to show the overall result of the analysis and the MC Progress view (5) to show the execution progress of the analysis.

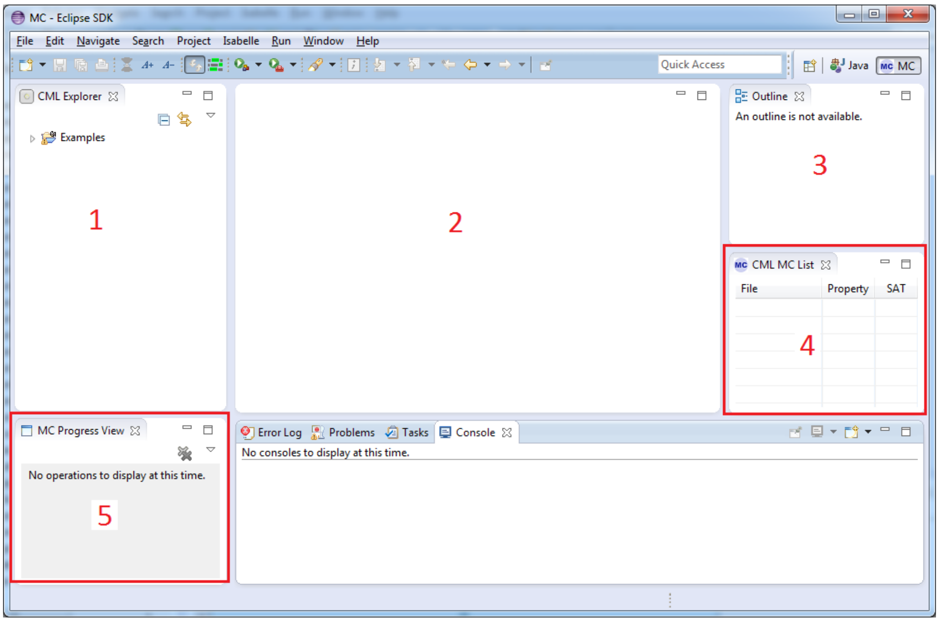

Figure 1: CML Model Checker Perspective

At startup, the CML model checker plugin checks (by using the PATH environment variable of your system) if the installation of FORMULA and GraphViz are working properly. For each problem found at startup, the COMPASS tool shows a warning as illustrated in Figure 2.

The analysis of a CML file is invoked through the context menu when the CML or the MC perspective are active (see Figure 3).

Select the CML file to be analysed. Then, `Right click -> Model check -> Property to be checked`. The analysis is performed and the information is shown in different views. The MC list view shows a ✓ or an **X** as result

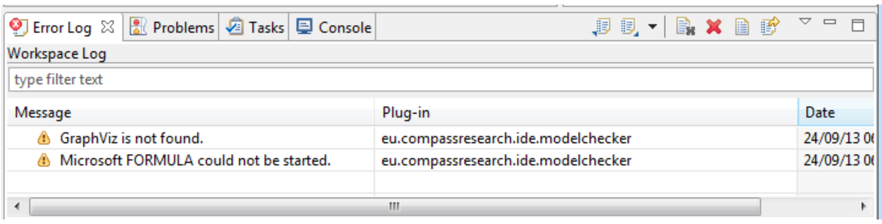

Figure 2: Warning about auxiliary software at startup

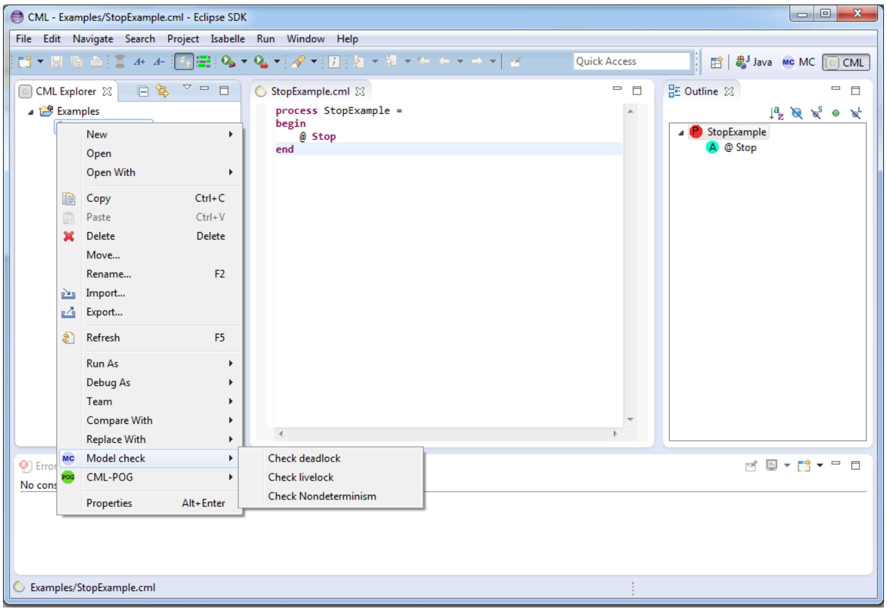

Figure 3: The Model Checker Context Menu

of the overall analysis (meaning satisfiable or unsatisfiable, respectively). Moreover, if the model is satisfiable, the trace validating the property can be viewed by double clicking the item of the MC list view.

It is worth noting that if FORMULA (or GraphViz) is not available and the user requires its use, the COMPASS tool shows appropriate messages like in Figure 4.

The model checker analysis uses an auxiliary folder (`generated\modelchecker`) to generate the FORMULA file (with extension `.4ml`). This file is loaded in the FORMULA tool to be analysed. Based on the result, the model checker plugin generates a GraphViz file (with extension `.gv`), compiles it (using `dot.exe`) to a graph file (with extension `.svg`) and shows it in the internal browser of Eclipse. All these steps are performed automatically.

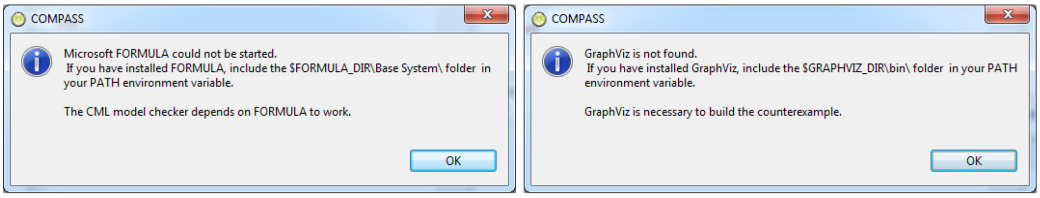

Figure 4: Messages when auxiliary software are not installed but are invoked

The initial state of the graph is two circles; intermediate states are simply circled; and the deadlocked state (or other special states related to properties verification) has a different colour (a red tone). Each state has a number and an information (hint) about the bindings (from variables to values), the name of the owner process, and the current context (process fragment). To see the internal information of each state just put the cursor over the state number.

Similarly, transitions are labelled with the corresponding event and also have a hint showing the source and the target states. This feature is useful to provide information about which rule (of the structured operational semantics) was applied.

The internal graph builder of the model checker considers the shortest path that makes the analysed file satisfiable. Thus, although there might be other counterexamples, it shows the shortest one.

2.2.1 Supported Constructs

This section gives an overview of the CML constructs that are implemented. We present the constructs using tables where the first column of each table gives the name of the operator, the second gives an informal syntax, and the last is a short description that gives the operator's status. If a construct is not supported entirely (no or partial implementation of the semantics), then the name of operator will be highlighted in red and a description of the issue will appear in the third column.

We also point out that type, values and operations definitions are implemented. The first two can involve only a single integer value.

The following tables describe all of the supported and partially supported actions. Where `A` and `B` are actions, `e` is an expression, `P(x)` is a predicate expression with `x` free, `c` is a channel name, `cs` is a channel set expression, `ns` is a nameset expression.

<table>
<tr><th>Operator
Syntax</th><th>Comments</th></tr>
<tr><td>Termination
<code>Skip</code></td><td>terminate immediately</td></tr>
<tr><td>Deadlock
<code>Stop</code></td><td>It yields a state with no outgoing transition</td></tr>
<tr><td>Chaos
<code>Chaos</code></td><td>Accepted but its analysis does not make sense as it can do anything (communicate or reject any event).</td></tr>
<tr><td>Divergence
<code>Div</code></td><td>It yields a livelock</td></tr>
<tr><td>Delay
<code>Wait e</code></td><td>Not implemented</td></tr>
<tr><td>Prefixing
<code>c!e?x:P(x)-> A</code></td><td>offers the environment a choice of events of the form <code>c.e.p</code>, where <code>p in set {x | x: T @ P(x)}</code>. Communication involving more than 1 data type are not represented uniformly in FORMULA, so only forms like <code>c.x</code> (where x is an integer) are supported.</td></tr>
<tr><td>Guarded action
<code>[e] & A</code></td><td>if <code>e</code> is true, behave like <code>A</code>, otherwise, behave like <code>Stop</code>.</td></tr>
<tr><td>Sequential composition
<code>A ; B</code></td><td>behave like <code>A</code> until <code>A</code> terminates, then behave like <code>B</code></td></tr>
<tr><td>External choice
<code>A [] B</code></td><td>offer the environment the choice between <code>A</code> and <code>B</code>.</td></tr>
<tr><td>Internal choice
<code>A |~| B</code></td><td>nondeterministically behave either like <code>A</code> or <code>B</code>.</td></tr>
<tr><td>Interrupt
<code>A /_\ B</code></td><td>Not implemented</td></tr>
<tr><td>Timed interrupt
<code>A /_ e _\ B</code></td><td>Not implemented</td></tr>
<tr><td>Untimed timeout
<code>A [_> B</code></td><td>Not implemented</td></tr>
<tr><td>Timeout
<code>A [_ e _> B</code></td><td>Not implemented</td></tr>
<tr><td>Abstraction
<code>A \\ cs</code></td><td>behave as <code>A</code> with the events in <code>cs</code> hidden. cs is a set of events involving communications with only one data type</td></tr>
<tr><td>Start deadline
<code>A startsby e</code></td><td>Not implemented</td></tr>
<tr><td>End deadline
<code>A endsby e</code></td><td>Not implemented</td></tr>
<tr><td>Channel renaming
<code>A[[c <- nc]]</code></td><td>Not implemented</td></tr>
<tr><td>Recursion
<code>mu X @ F(X)</code></td><td>explicit definition of a recursive action.</td></tr>
<tr><td>Mutual Recursion
<code>mu X,Y @ (F(X,Y), G(X,Y)</code></td><td>Not implemented</td></tr>
</table>

Operator Syntax	Comments
Interleaving `A [|| ns1	ns2 ||] B`	Not implemented
Interleaving (no state) `A ||| B`	execute `A` and `B` in parallel without synchronising. Neither `A` nor `B` change the state.
Synchronous parallelism `A [| ns1	ns2|] B`	Not implemented
Synchronous parallelism (no state) `A || B`	Not implemented
Alphabetised parallelism `A [ns1| cs1 || cs2 |ns2] B`	Not implemented
Alphabetised parallelism (no state) `A [cs1 || cs2] B`	Not implemented
Generalised parallelism `A [|ns1	cs	ns2|] B`	Not implemented
Generalised parallelism (no state) `A [| cs |] B`	execute `A` and `B` in parallel synchronising on the events in `cs`. Neither `A` nor `B` change the state.

Table 2: Parallel action constructors.

Operator Syntax	Comments
Replicated sequential composition `; i in seq e @ A(i)`	Not implemented
Replicated external choice `[] i in set e @ A(i)`	offer the environment the choice of all actions `A(i)` such that `i` is in the set `e`.
Replicated internal choice `|~| i in set e @ A(i)`	nondeterministically behave as `A(i)` for any `i` in the set `e`.
Replicated interleaving `||| i in set e @ [ns(i)] A(i)`	Not implemented
Replicated generalised parallelism `[|cs|] i in set e @ [ns(i)] A(i)`	execute all actions `A(i)` (for `i` in the set `e`) in parallel synchronising on the events in `cs`. Each action `A(i)` can only modify the state components in `ns(i)`.
Replicated alphabetised parallelism `|| i in set e @ [ns(i)| cs(i)] A(i)`	Not implemented
Replicated synchronous parallelism `|| i in set e @ [ns(i)] A(i)`	Not implemented

Table 3: Replicated action constructors.

Operator Syntax	Comments
Let `let p=e in a`	evaluate the action `a` in the environment where `p` is associated to `e`.
Block `(dcl v: T := e @ a)`	declare the local variable `v` of type `T` (optionally) initialised to `e` and evaluate action `a` in this context.
Assignment `v:=e`	assign e to v

Multiple assignment
`atomic (v1 := e1, ...,`
`vn := en)`
	Not implemented

Call
`(1)op(p)`
`(2)A(p)`
	execute operation `op` of the current or process (1) with the parameters `p`. (2) execute action `A` with parameters `p`.

Assignment call `v := op(p)`	Not implemented
Return `return e` or `return`	Not implemented

Specification
`[frame`
`wr v1: T1`
`rd v2: T2`
`pre P1(v1,v2)`
`post`
`P2(v1,v1~,v2,v2~)]`
	Not implemented

New `v := new C()`	Not implemented

Table 4: CML statements.

<table>
<tr><th>Operator
Syntax</th><th>Comments</th></tr>
<tr><td>Nondeterministic if statement
<pre>
if e1 -> a1
| e2 -> a2
| ...
end
</pre></td><td></td></tr>
<tr><td>If statement
<pre>
if e1 then a1
elseif e2 then a2
...
else an
</pre></td><td>Not implemented</td></tr>
<tr><td>Cases statement
<pre>
cases e:
p1 -> a1,
p2 -> a2,
...,
others -> an
end
</pre></td><td>Not implemented</td></tr>
<tr><td>Nondeterministic do statement
<pre>
do e1 -> a1
| e2 -> a2
| ...
end
</pre></td><td>Not implemented</td></tr>
<tr><td>Sequence for loop
<code>for e in s do a</code></td><td>Not implemented</td></tr>
<tr><td>Set for loop
<code>for all e in set S do a</code></td><td>Not implemented</td></tr>
<tr><td>Index for loop
<code>for i=e1 to e2 by e3 do a</code></td><td>Not implemented</td></tr>
<tr><td>While loop
<code>while e do a</code></td><td>Not implemented</td></tr>
</table>

Table 5: Control statements.

2.3 Examples

This section presents some examples of CML specifications and their analysis using the model checker. The examples are available in the COMPASS SVN repository. We recommend that you download and try them. The following figures are intuitive and show the analysis result for some examples.

Immediate Deadlock

The CML file `action-stop.cml` is the most simple deadlock process. Figure 5 shows the result of its analysis and the corresponding graph. The model checker list view shows the analysis result (satisfiable) for the file `action-stop.cml` considering the `Deadlock` property. Trivially, the process has only one initial state that is also a deadlock state. This can be seen by a double click in the model checker list view item. It is worth noting that the content of any state of the graph is available by putting the cursor over the state. Basically, the information of each state has the format `(vars,proc)`, where `vars` contains the manipulated variables (bindings) and `proc` is a process fragment. Furthermore, the generated files can be viewed by refreshing the project. The user can see the content of all files (`.4ml`,`.gv` and `.svg`) as they are text files.

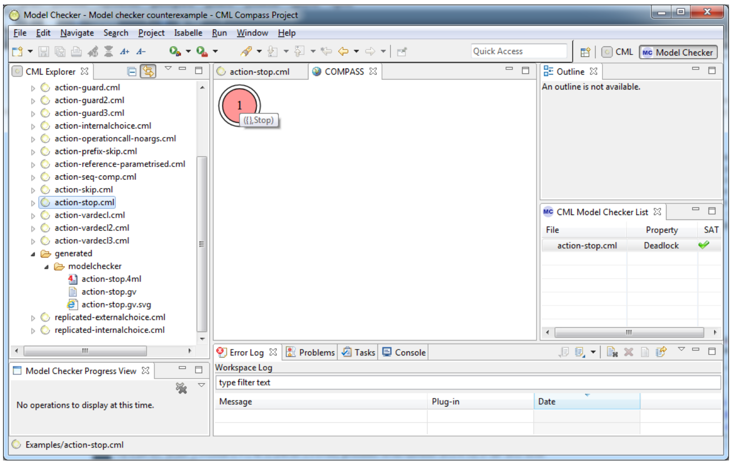

Figure 5: An immediate deadlock example

When the analysed file is unsatisfiable, and the user tries to see the graph, the model checker plugin returns a message indicating that the graph is available only for satisfiable models (Figure 6).

Figure 6: Message when the graph is not available

An External Choice Example

The CML file `action-externalchoice-nostate2.cml` is an example involving the use of auxiliary actions and the external choice operator. Figure 7 shows its analysis and counterexample to find a deadlock. The content of all states are also depicted just to illustrate the progress of the process as described by the rules of the operational semantics [BCC+13]. The external choice `[]` is translated (via a τ-transition) in the two first transitions (using left association). In state 3, the process expands (via a τ-transition) the action call `C`, which leads to an state (4) from where the transition labelled with `c` leads to a deadlock state (5).

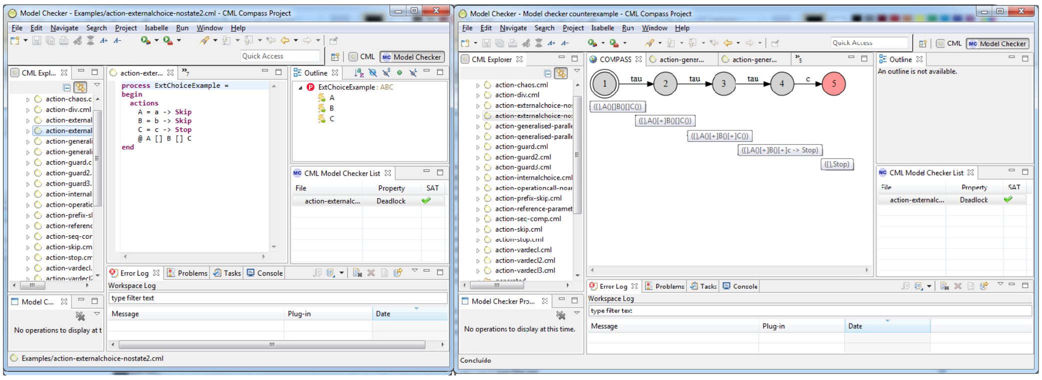

Figure 7: An external choice example

3 CSP embedding in FORMULA

This section provides information about the underlying infrastructure used by the CML model checker. We first introduce the FORMULA tool and its language to clarify such a framework and its constructs. Then we present the encoding of CML in the language of FORMULA. We start by showing the embedding for CSP and then evolve such a representation by including data manipulation to contemplate VDM constructs.

3.1 FORMULA Framework

The Microsoft FORMULA (Formal Modelling Using Logic Programming and Analysis) tool encompasses several facets to provide a framework to (abstractly) reason about models and analysis:

1. A modern formal specification language that follows the principles of model-based development (MBD). The language of FORMULA is based on algebraic data types (ADTs) and strongly-typed constraint logic programming (CLP). It supports concise specifications of abstractions (in a Prolog-like style) and model transformations.
2. Use of SMT (Satisfiability Modulo Theories) solving. The automatic integration with the Z3 SMT solver is useful to make automatic analysis and instantiations inside FORMULA. This brings the advantage of providing model finding and design space exploration facilities, in which FORMULA can be used to construct system models satisfying complex domain constraints.

The main elements of a FORMULA specification are:

- *Domains*: used to create abstractions of real-world problems in a way very similar to Prolog (with facts, rules, and queries);
- *Facts*: n-ary relations or constructors ($n \geq 1$), completely instantiated. They can be primitive or not. Only primitive facts can be used within (partial) models (given as initial facts). On the other hand, primitive facts cannot be used as head of rules because they cannot be derived from other facts;
- *Rules*: they have the same role as in Prolog, except that rules cannot leave unbounded the elements used in the head. A FORMULA rule has the format LHS :- RHS, where the left-hand side (LHS) is the head and the right-hand side (RHS) is the body of the rule (a list of facts used to derive the LHS). For

every element X used in the LHS, we must have some constructor Cons(X) in the RHS to constrain the possible values of X; FORMULA can only build the head from the elements of the body (bottom-up approach);

- *Queries*: quantifier-free formulae in terms of constructors of the language. The special query `conforms` combines other queries using logical operators and is used as the main goal to validate a model in a domain. When a (partial) model is inspected in FORMULA, the `conforms` clause is the starting point of the searching procedure. If it is not possible to find an instance that satisfies this special query, the (partial) model is said to be Unsatisfiable;
- *(Partial) models*: these are possible instances of domains. The main distinction between models and partial models are that models are closed instances and partial models are open (to be closed/instantiated by the solver) instances.

Although domains have similar elements like Prolog programs, they work differently. Prolog uses rules as starting points of the searching procedure and stops at facts (a top-down approach), whereas FORMULA uses (primitive) facts as starting points to create new facts (a bottom-up approach). Figure Figure 8 illustrates the work performed by FORMULA in an analysis. It takes the main goal (conforms clause) and the facts given in a (partial) model as starting point. From the (initial) base of facts and the RHS of domain rules, FORMULA tries to generate other facts (according to the LHS of domain rules). If the new base of facts satisfies the main goal, the model is SAT (satisfiable). Otherwise, FORMULA keeps generating new facts again until the base of facts stops increasing (a bottom-up fixed point based search). At the end of this iterative generation, if the goal cannot be satisfied, the model is UNSAT (unsatisfiable). Furthermore, if any SMT-solving activity (instantiation, evaluation, etc.) is required, FORMULA invokes Z3 automatically.

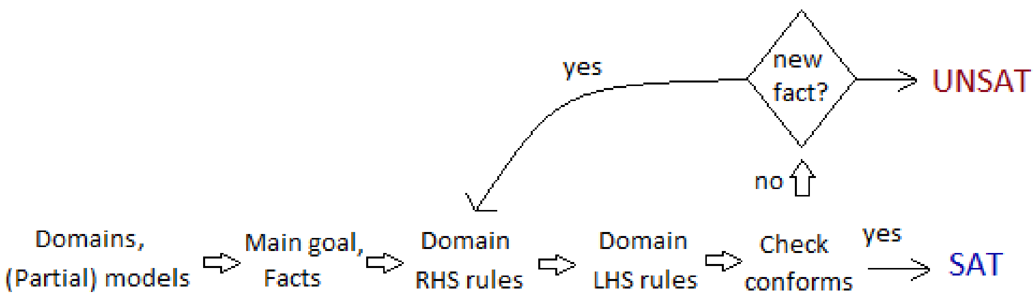

Figure 8: Iterative analysis of FORMULA

When open facts are used in partial models, they activate the symbolic execution algorithm inside FORMULA that creates symbolic derived facts (head of rules). If a rule (head) is bound only by previous derived facts, this can create an infinite loop in the symbolic execution algorithm of FORMULA and making the search diverges. Therefore it is advised to have at least one primitive fact in the body of a rule to avoid infinite application of such a rule. This creates a bounded analysis similar to what is done in bounded model checking [BCCZ99, AMP09]. Therefore our CML model checker can have infinite predicates and communications but not infinite states. That is, we aim at creating finite symbolic labelled transition systems as we will see later in Section 3.4.

3.1.1 A simple example

We illustrate the work of FORMULA using an example that captures the essence of a basic digraph (see Figure 9).

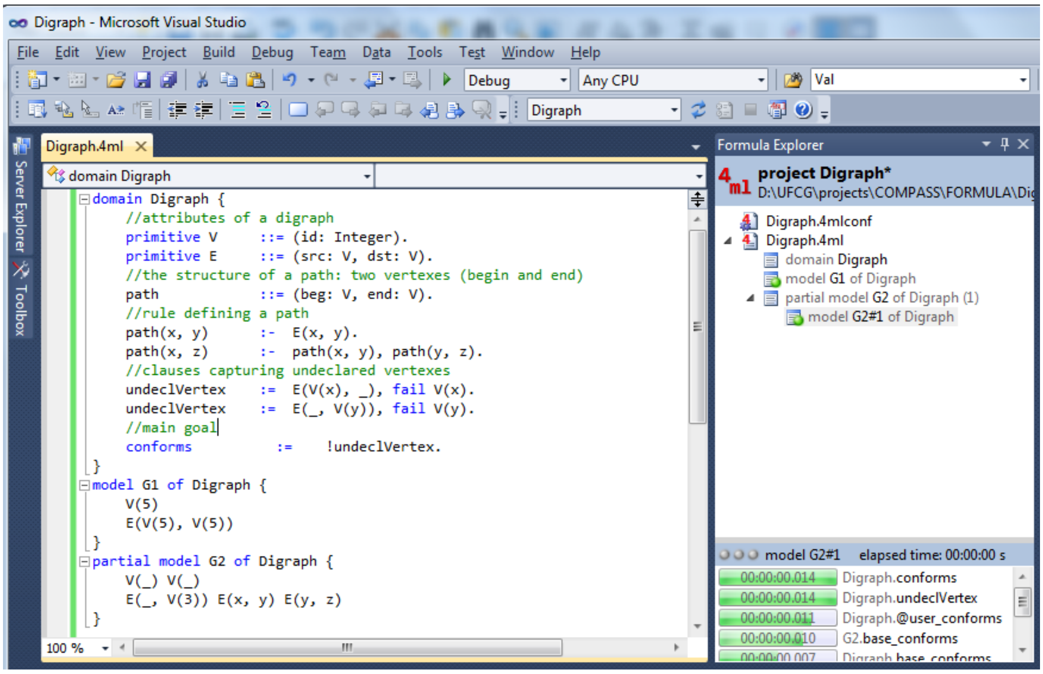

Figure 9: FORMULA snapshot model analysis

A digraph is modelled as a domain containing a set of vertexes (V) and a set of edges (E). The qualifier primitive indicates that vertexes and edges cannot be generated during the analysis (however their values can be instantiated). The rule path links vertexes where there is a single edge or several edges. By using the definition of path, FORMULA is able to find a path between two vertexes (if it exists) by building paths between intermediate vertexes. The query `undeclVertex` establishes constraints upon the domain; it captures undeclared vertexes by checking if the first (`E(V(x),_)`) or the second (`E(_,V(y))`)components of edges have not been declared as vertexes (`fail(V(x)` and `fail(V(y)`, respectively). Finally, the conforms query defines the main goal: a valid graph cannot have undeclared vertexes.

We use two models to check instances of the domain Digraph. The model `G1` defines a digraph with one vertex (`V(5)`) and a self-edge. As it has no undeclared vertexes, FORMULA detects its conformance with the Digraph domain (satisfiable). Concerning the partial model `G2`, there are three edges and two vertexes (some are left undetermined). These elements play the role of parameters to be instantiated by FORMULA to make G2 satisfiable. In this case, FORMULA found the instances `V(3)` and `V(-103701)` and used `V(3)` to validate the edge with the first vertex undetermined (`E(V(3),V(3))`). The value `-103701` is arbitrary and was generated only because there are two given vertexes in `G2`. If we remove one vertex, only `V(3)` is used. In this sense, FORMULA works as a symbolic executor, expanding its base of facts as much as necessary. This fits well the purposes of LTS generation.

3.2 Structured Operational Semantics of CSP

Although there are three formal semantics for CSP (algebraic, operational and denotational), we focus on the operational semantics [Ros10] as it is closer to the purpose of automatic verification via model checking: it defines the behaviour of a process as a labelled transition system (LTS). Formally, an LTS is a tuple $(S, S_0, T, \Sigma^{\checkmark,\tau})$, where S is a set of states, S_0 is an initial state ($S_0 \in S$), T is a transition relation over $S \times \Sigma^{\checkmark,\tau} \times S$, and $\Sigma^{\checkmark,\tau}$ is the set of all possible events; visible events are represented by Σ and the special events $\checkmark$ and τ are used to semantically represent successful termination and internal actions, respectively. The representation $\Sigma^{\checkmark,\tau}$ stands for $\Sigma \cup \{\checkmark, \tau\}$.

According to [Plo81], the structured (or structural) operational semantics (SOS) of a language is an operational method of specifying semantics based on syntactic transformations and simple operations on discrete data. The occurrence of such operations is associated to elementary steps (firing rules) and recorded as transitions (or moves). This means that the LTS corresponding to a specification (or program) P written in a language L can be generated by applying the firing rules of L on each syntactic fragments (BNF) of P. A firing rule has the format:

$$\frac{premises}{conclusion}, (conditions)$$

In the above format, the *conclusion* is mandatory. The rest is optional and depends on the language constructors involved as well as the kind of semantics the designer is intending to give. When premises are absent, the rule is said to be an *axiom*. A generic example of a firing rule is given as follows.

$$\frac{p_1^1 \longrightarrow p_1^{'}, \ldots, p_n \longrightarrow p_n^{'}}{Op(p_1, \ldots, p_n) \longrightarrow Op(p_1^{'}, \ldots p_n^{'})}, C(p_1, \ldots, p_n)$$

where $Op(p_1, \ldots, p_n)$ and $Op(p_1^{'}, \ldots, p_n^{'})$ are constructors of the language following its BNF, and $p_1, \ldots, p_n$ are its operands. The predicate $C(p_1, \ldots, p_n)$ states the conditions under which such a rule can be applied beyond the premises. That is, the premises act as a pattern condition and $C(p_1, \ldots, p_n)$ as a boolean and more general condition. Moreover, it can be the case of certain fragments of a language does not have an associated firing rule (as it is the case of CSP).

The embedding of CSP has been designed in such a way that tit directly follows its structured operational semantics (SOS). This leads to a very intuitive way of creating semantics-preserving model checkers. This is very important in our context because CML is intended to be a heterogeneous language integrating behaviour, state, time, mobility, probability, etc.

The language CSP is based on the notion of processes and (communication) events. A process is an independent self-contained entity with particular interfaces through which it interacts with its environment (the context outside the process). An event describes a particular kind of atomic and indivisible action that can be performed by the process. The set of all events a process can perform is known as the alphabet of the process. Our current embedding of CSP in FORMULA considers the most common constructs of CSP given by the following syntax:

$$
\begin{array}{lcll}
Proc & ::= & Stop & (Deadlock) \\
 & & \mid Skip & (Successfultermination) \\
 & & \mid a \rightarrow Proc & (Prefix) \\
 & & \mid Proc \sqcap Proc & (Internalchoice) \\
 & & \mid Proc \, \Box \, Proc & (External\, choice) \\
 & & \mid Proc \not\leqslant g \not\geqslant Proc & (Conditionalchoice) \\
 & & \mid g \,\&\, Proc & (Boolean\, guard) \\
 & & \mid Proc \underset{X}{\parallel} Proc & (Generalisedparallelism) \\
 & & \mid Proc \setminus X & (Hiding) \\
 & & \mid Proc \,|||\, Proc & (Interleaving) \\
 & & \mid Proc; Proc & (Sequentialcomposition) \\
 & & \mid \mu Y \bullet F(Y) & (Recursion) \\
 & & \mid ProcCall & (Process\, call)
\end{array}
$$

The primitive processes $Stop$ and $Skip$ denote, respectively, immediate deadlock (as a broken system) and successful termination (it does nothing besides terminating); while $Stop$ communicates no event, whereas $Skip$ communicates a special event ✓(tick) before terminating. The *prefix* process $a \rightarrow P$ offers the event a to its environment, and after its occurrence, it behaves as P. When values may be exchanged between processes, we use the constructs $c!exp$ (to send the value corresponding to expression exp) and $c?x$ (to receive a value and store it in the variable x) in place of the event a. The *internal choice* $P \sqcap Q$ behaves as P or Q, but the choice is arbitrary (an internal and nondeterministic decision). The *external choice* $P \,\Box\, Q$ behaves as P or Q where the choice is made by the environment (that is, the context outside P and Q decides which of P or Q should evolve). The *conditional choice* $P \not\leqslant g \not\geqslant Q$ denotes a process that behaves as P is the condition guard g is true, or as Q otherwise. The *guarded choice* $g \,\&\, P$ is equivalent to $P \not\leqslant g \not\geqslant Stop$. The process $P \underset{X}{\parallel} Q$ stands for the *generalised parallel composition* of the processes P and Q with synchronisation set X. This states that the processes P and Q must progress together for events that belong to X (that is, they must engage on the same events). On the other hand, for events outside X, if these events are different each process can evolve independently; otherwise, just one of them evolves (after a nondeterministic choice). The process $P \,|||\, Q$ establishes an *interleaved* execution where P and Q are executed independently; this construct is similar to a parallelism with empty synchronisation set (that is, $P \underset{\{\}}{||} Q$). The sequential composition $P; Q$ represents a process that behaves as P until P terminates successfully, and then the composition behaves as Q. The process $\mu Y \bullet F(Y)$ represents a recursive process where F is any CSP term involving Y). When Y occurs we replace it by $\mu Y.F(Y)$ again (unfold). Finally, the $ProcCall$ construct denotes any process call possibly involving parameters.

The semantic rules of CSP follow the Plotkin's style [Plo81] and are presented by the firing rules of Figure 10. The special state Ω is a semantic element that represents a state with no outgoing transition (a final state). It corresponds to the behaviour of $Stop$ (a deadlock process performs no action at all and, therefore, has no associated transition). The process $Skip$ can perform a single action $\checkmark$, after which it does nothing more; this corresponds to a $\checkmark$-transition leading to Ω (rule *termination*).

The transition rule for the prefix operator is represented by rule *prefix*; it states that an initially accepted event a (where $a \in A$) is performed and the following behaviour is determined by replacing the occurrences of x by a in the process P, where the event a possibly involves data communication. The internal choice can originate two transitions where the process decides (by an internal action) to behave as one of its parts. The special event τ represents such a decision.

The external choice operator has two main situations that originate several transitions. If there is a possible internal progress in any constituent process (the premises of the external choice τ rule), the external choice also evolves by performing an internal action. Otherwise, the external choice evolves by behaving as one of its constituent parts (stated in the premises of the external choice Σ rule). The conditional and the guarded choices have no explicit transition rules because they are rewritten to one of the other cases. The behaviour of $P \not\leftarrow g \not\rightarrow Q$ is defined by P or Q; this decision is determined by evaluating the boolean guard g and, hence, does not originate a specific transition. The behaviour of $g \,\&\, P$ is similar to $P \not\leftarrow g \not\rightarrow Stop$ and has no transition for evaluating the guard g.

The generalised parallelism has different situations for originating transitions. If any constituent process can evolve by an internal action, the parallelism also does so accordingly (the *parallelism* τ rule). When the constituent parts want to perform different events that do not belong to the synchronisation set, they evolve asynchronously and the parallelism can progress by evolving both of its constituent parts independently (the *async-parallelism* rule). On the other hand, if both constituent processes offer the same event (from the synchronisation set), then there is a synchronous progress (the *sync-parallelism* rule). Finally, if one process in the parallelism terminates the entire combination waits for the other process terminate as well. And only if both terminate, the entire combination performs a $\checkmark$-action and leads to the final state Ω (the *dist-term-parallelism* rule). Note that a deadlock might occur if one process terminates and the other wants to synchronise with it.

$$\frac{}{Skip \xrightarrow{\checkmark} \Omega} \qquad \textit{(termination)}$$

$$\frac{}{x{:}A \to P(x) \xrightarrow{a} P[a/x]}\ (a \in A) \qquad \textit{(prefix)}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \quad \frac{}{P \sqcap Q \xrightarrow{\tau} Q} \qquad \textit{(internal choice)}$$

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \quad \frac{Q \xrightarrow{\tau} Q'}{P \,\Box\, Q \xrightarrow{\tau} P \,\Box\, Q'} \qquad \textit{(external choice } \tau\textit{)}$$

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'}(a \neq \tau) \quad \frac{Q \xrightarrow{a} Q'}{P \,\Box\, Q \xrightarrow{a} Q'}(a \neq \tau) \qquad \textit{(external choice } \Sigma\textit{)}$$

$$\frac{P \xrightarrow{\tau} P'}{P \underset{X}{\|} Q \xrightarrow{\tau} P' \underset{X}{\|} Q} \quad \frac{Q \xrightarrow{\tau} Q'}{P \underset{X}{\|} Q \xrightarrow{\tau} P \underset{X}{\|} Q'} \qquad \textit{(parallelism } \tau\textit{)}$$

$$\frac{P \xrightarrow{a} P'}{P \underset{X}{\|} Q \xrightarrow{a} P' \underset{X}{\|} Q}(a \in \Sigma \setminus X) \quad \frac{Q \xrightarrow{a} Q'}{P \underset{X}{\|} Q \xrightarrow{a} P \underset{X}{\|} Q'}(a \in \Sigma \setminus X) \qquad \textit{(async-parallelism)}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{X}{\|} Q \xrightarrow{a} P' \underset{X}{\|} Q'}(a \in X) \qquad \textit{(sync-parallelism)}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \underset{X}{\|} Q \xrightarrow{\tau} \Omega \underset{X}{\|} Q} \quad \frac{Q \xrightarrow{\checkmark} Q'}{P \underset{X}{\|} Q \xrightarrow{\tau} P \underset{X}{\|} \Omega} \quad \frac{}{\Omega \underset{X}{\|} \Omega \xrightarrow{\checkmark} \Omega} \qquad \textit{(dist-term-parallelism)}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}(a \in A) \quad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A}\ (a \notin A) \qquad \textit{(hiding)}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \setminus A \xrightarrow{\checkmark} \Omega} \qquad \textit{(hiding } \checkmark\textit{)}$$

$$\frac{P \xrightarrow{a} P'}{P;Q \xrightarrow{a} P';Q}(a \neq \checkmark) \quad \frac{P \xrightarrow{\checkmark} P'}{P;Q \xrightarrow{\tau} Q} \qquad \textit{(sequential composition)}$$

$$\frac{}{\mu Y \bullet F(Y) \xrightarrow{\tau} F[(\mu Y \bullet F(Y))/Y]} \qquad \textit{(recursion)}$$

Figure 10: Firing rules for CSP

The transitions of the hiding $P \backslash A$ are the same transitions of P with a subtle change: for all events from A, the event is hidden and originates a τ-transition (the *hiding* rule). Furthermore, independently of the performed event, the set of events to be hidden is propagated to the following behaviour. In the case where the process terminates the hiding also does so (the *hiding* ✓ rule). The interleave operator has no firing rule because it is equivalent to a parallelism with an empty synchronisation set.

The *sequential composition* rule states that if the first process terminates successfully, the composition behaves as the second process; otherwise, only the first process evolves in the composition. The usual unfold of a recursion is represented by a τ-transition where the bounded variable is replaced in the original expression by the entire recursive definition again. The transition for a process call is not necessary as it simply corresponds to the execution of any (already) defined rule.

3.3 CSP Refinement Checking

Model checking is an automatic technique to investigate whether a property f is valid in a given model M, or simply $M \models f$. In general, the model M describes the behavior of some concurrent language L and the property f is written using some fragment of temporal logic (TL). It is normally implemented as a black box containing very optimised algorithms (in terms of space and time) that traverse the model M (a graph). Nevertheless, this satisfaction relation can also be checked via refinement, which is the focus of this work. That is, one can use another model M_f (the model M_f is known—or built in such a way—to satisfy the property f in the most nondeterministic possible way) as a way of checking that the model M satisfy a property f; in this case, it is formally represented as $M_f \sqsubseteq M$. This is the strategy used in CSP, where both models (M_f and M) can be compared with respect to three main models: traces ($\mathcal{T}$), failures ($\mathcal{F}$) or failures-divergences ($\mathcal{FD}$). These models are defined by the denotational semantics of CSP. However, due to the congruence between the operational and the denotational semantics [3], we can also check CSP refinements by using the operational semantics.

We focus on traces refinement to simplify our presentation and because it is the simplest denotational model that allows us to check the properties we implemented in this work. The extension of our model checker to deal with the standard failures-divergences requires a more elaborate embedding of properties

[3]This congruence for CSP models is stated in [Ros10]. However, the work reported in [HJ98] shows how one can obtain such a congruence in general.

(FORMULA queries) to capture failures and divergences of the generated LTS, but it is feasible and achievable from the infra-structure we create for traces analysis. In particular, we will see later that with the current infra-structure we already perform deadlock (this requires the stable failures semantics of CSP in the FDR model checker) and livelock (this requires the stable failures-divergences semantics in FDR) analyses because all complementary information can be inferred from the traces.

Concerning the traces model, for each fragment of the CSP language it is defined the traces it can produce by using the function $traces$: $Process \rightarrow (\Sigma^{\checkmark,\tau})$ from the denotational semantics of CSP. Some examples of traces calculation are listed as follows.

- $traces(STOP) = \{\langle\rangle\}$;
- $traces(SKIP) = \{\langle\rangle, \langle\checkmark\rangle\}$;
- $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid a \in A \wedge s \in traces(P)\}$;
- $traces(a?x \rightarrow P) = \{\langle\rangle\} \cup \{\langle a.v\rangle \frown s \mid v \in T_a \wedge s \in traces(P[v/x])\}$;
- $traces(P \sqcap Q) = traces(P) \cup traces(Q)$;
- $traces(P \,\Box\, Q) = traces(P) \cup traces(Q)$;
- $traces(P \backslash A) = \{s \backslash A | s \in traces(P)\}$;

The function $traces$ provides the set of histories (or performed actions) a process can exhibit. Based on it, the refinement relation for the traces model ($\sqsubseteq_{\mathcal{T}}$) is easily defined:

$$P \sqsubseteq_{\mathcal{T}} Q \equiv traces(Q) \subseteq traces(P)$$

According to the definition of traces refinement, a process Q traces refines another process P whether Q produces at least the same traces as P. This can also be captured by comparing the executions of P and Q, according to their operational semantics. We use this strategy in our work. That is, instead of creating sets of sequences of events as in the traces function, we walk through the event-annotated transitions in a somewhat similar way like the model checker FDR.

3.4 Capturing CSP SOS in FORMULA

Work on model checking assumes that M is given and focuses on formally describing what $M \models f$ means, or how to check f by traversing M. For languages whose syntax are closer to an LTS, such as LTSA [MK99] or Petri Nets [Mur89], the model M is easily achievable and (usually) is correct. Nevertheless, for languages such as CSP [Ros10], PROMELA [GM99] and Circus [WCF05], creating a model checker by a direct programming approach can be too error-prone, as the model M can be wrong and the tool concentrates on analysing it assuming the SOS of L. In particular our first effort towards creating a Circus model checker aimed at using Perfect Developer [Cro03]. however, the results were restrict and very difficult to maintain. This was expected because this approach is still programmatic, although Perfect Developer has formal development support.

In practice, most model checkers create M from L using some black-box implementation susceptible to programming errors. However, if the model M is systematically created from the SOS of L (that is, $M \in SOS\{L\}$) the model checker becomes a semantics-preserving model checker for L relative to the semantics of the framework used to encode the SOS of L.

In this section we show how to systematically capture the firing rules of the CSP SOS in FORMULA so that the LTS is directly derived from a conceptual (and formal) model similar to Leuschel [Leu01] and Verdejo [VMO02]. This systematic capture can be automated. The representation proposed by Corradini [CHM00] is an abstract and intuitive description for structured operational semantics. As long as it works as a Domain Specific Language (DSL), our strategy can be adapted to derive a FORMULA script (the model checker) from a SOS description.

The semantics of a complex language might have several aspects, such as, for example, data aspects and control (or concurrency). The ideal situation to guarantee full correctness about a possible encoding of a language semantics into a programming framework is a one-to-one mapping from each syntactic fragment of the language to its meaning (or interpretation) using the constructors of the programming framework. This is called a deep semantics embedding.

Sometimes, however, the semantics of part of the source language is close to the one available in the framework. This is frequently the case of data aspects, where the framework already provides the means to deal with arithmetic expressions, known data types (natural, integer, real numbers and strings), sets, relations, sequences, and so on. When a language semantics is captured in this way, we say that such a semantics was a shallow embedding in the programming framework.

3.4.1 Basic Shallow Embedding in FORMULA

We propose a way to capture the structured operational semantics of CSP using a so-called hybrid semantics embedding in which behavioural aspects are captured in a deep embedding way and data aspects are not interpreted as much as possible (For those that we cannot find a direct mapping we follow a deep embedding as well. This is used for supporting for sets, sequences, and mappings of VDM in FORMULA). They are simply mapped to the available elements, yielding a shallow embedding. Although FORMULA provides basic data types (Integer, Natural, Real, String), more complex types (like sets, relations, functions, sequences, bags, etc.) are absent and the mapping is not so direct. The domain `ShallowEmbedding` shows the mappings for basic types and for sequence type.

```
domain ShallowEmbedding {
 // Types
 primitive UNDEF   ::= {undef}.     //a default value for all types
 primitive Int     ::= (Integer).   //integers
 primitive Nat     ::= (Natural).   //naturals
 primitive Str     ::= (String).    //strings
 primitive IR      ::= (Real).      //reals
 primitive Seq     ::= (SeqDef).    //sequence
 EmptySeq          ::= {empty}.
 primitive SeqCont ::= (head:Types,tail:SeqDef).
 SeqDef            ::= EmptySeq + SeqCont.
 Types             ::= Int + Nat + IR + Str + Seq.
 aSeq              ::= (SeqRest).

 // Some relational operators
 primitive EQ    ::= (x:Types,y:Types).  //equal
 primitive NEQ   ::= (x:Types,y:Types).  //not equal
 primitive LT    ::= (x:Types,y:Types).  //less than
 primitive GT    ::= (x:Types,y:Types).  //greater than
 bExps           ::= EQ + NEQ + LT + GT.
}
```

The most basic types—integer (`Int`), natural (`Nat`), real (`IR`) and string (`Str`)—are directly mapped to their corresponding types in FORMULA. The type `UNDEF` is defined to provide a default value for variables declared but not initialised with a specific value of its type. The sequence type (`Seq`) is defined by another constructor (`SeqDef`) that is the union of two types (inductively defining a sequence). The constructor `EmptySeq` defines an empty sequence and `SeqCont` defines a non-empty sequence as a tuple containing a head and a tail. Finally, all types are defined by the union of the basic types and `Seq`. The (derived) constructor `aSeq` is used just to provide a way of generating sequences during the analysis.

Each relational operation (`EQ`, `NEQ`, `LT`, `GT`) is intuitively modelled as a pair containing the operands. At the end, the constructor `bExps` uses union of types to capture all possible relational expressions to be used in our encoding.

User Defined Types in FORMULA

The support of FORMULA for type union allows one to extend type definitions in a quite flexible way. For example, the following CSP `datatype` definition

$$datatype\ ANSWER = OK|ERROR$$
$$nametype\ POINT = Int.Int$$

is captured in FORMULA as follows

```
domain ShallowEmbedding {
 // Types
 primitive Int      ::= (Integer).  //integers
 primitive Nat      ::= (Natural).  //naturals
 primitive Str      ::= (String).   //strings
 primitive IR       ::= (Real).     //reals
 primitive Seq      ::= (SeqDef).   //sequence
 EmptySeq           ::= {empty}.
 primitive SeqCont  ::= (head:Types,tail:SeqDef).
 SeqDef             ::= EmptySeq + SeqCont.

 //Defining the new types
 primitive ANSWER ::= {OK,ERROR}
 primitive POINT ::=  (Integer,Integer).

 //Extending the pre-defined types
 Types              ::= Int + Nat + IR + Str + Seq + ANSWER + POINT.
}
```

Both types are represented by primitive constructs (as all pre-defined types also are). However, types defined by using explicit values are captured by sets of values in FORMULA, whereas types defined by combining existing types with the CSP "." type operator[4] are represented as tuples.

After representing each new type individually, the union of all types is adjusted to include the new types. This makes them available in the general scope of the FORMULA script.

3.4.2 CSP Syntax in FORMULA

The CSP SOS is captured as in the real scenario: the syntax and semantics are described in two separated domains: syntax and semantics. The former defines the structures (building blocks) necessary to represent CSP constructs for events and processes, according to its BNF grammar given in Section 3.2.

```
domain CSP_Syntax includes ShallowEmbedding {
 SpecialEvents       ::= {tick,tau}.
 primitive BasicEv   ::= (name:String).
```

[4]This operator means cartesian product or tuple construction.

```
primitive CommEv    ::= (name:String,data:Types).
Sigma               ::= BasicEv + CommEv.
SigmaTickTau        ::= Sigma + SpecialEvents.
BasicProcess        ::= {Stop,Skip}.
primitive Prefix    ::= (ev:Sigma,proc:CSPProcess).
primitive iChoice   ::= (lProc:CSPProcess,rProc:CSPProcess).
primitive eChoice   ::= (lProc:CSPProcess,rProc:CSPProcess).
primitive bChoice   ::= (cond:bExps,lProc:CSPProcess,rProc:CSPProcess).
primitive seqC      ::= (lProc:CSPProcess,rProc:CSPProcess).
primitive hide      ::= (proc:CSPProcess,hideS:String).
primitive par       ::= (lProc:CSPProcess,SyncS:String,rProc:CSPProcess).
NoPar               ::= {nopar}.
SPar                ::= (Types).
DPar                ::= (p1:Types,p2: Types).
Param               ::= NoPar + SPar + DPar.
primitive proc      ::= (name : String, p: Param).
CSPProcess          ::= BasicProcess + Prefix + iChoice + eChoice +
                        bChoice + seqC + hide + par + proc.
}
```

Events are represented by different constructs. The special events $\checkmark$ and τ are represented by the element `SpecialEvents`. The visible events (from Σ) are classified as basic events (`BasicEv` does not have communication values) and communication events (`CommEv` involves communication values); the union of these types defines the entire set `Sigma`. The element representing $\Sigma^{\checkmark,\tau}$ (`SigmaTickTau`) is obtained by the union of `Sigma` and `SpecialEvents`.

The representation of processes starts by the primitive processes $Stop$ and $Skip$. They are captured by the element `BasicProcess`. The `Prefix` is represented as a pair of an event (from `Sigma`) and a next behaviour (a process). Internal and external choices are respectively represented by the constructors `iChoice` and `eChoice`; each of them is composed by a left and a right processes. The conditional choice constructor (`bChoice`), on the other hand, has three components: a boolean condition, a process defining the behaviour if the condition is valid and another process defining the behaviour if the condition is invalid. The constructor for sequential composition (`seqC`) is defined as a pair containing the first and the second processes. The hiding (`hide`) is represented by a constructor containing a process and a set of events to be hidden (represented as a string). This is a design decision used to avoid interpretation of set operations in FORMULA; the necessary information over sets (membership, inclusion, etc.) is given as initial facts to improve the performance of FORMULA. We discuss more about this in Section 3.4.3.

The parameters are defined by a construct representing no parameters (`NoPar` contains only the element `nopar`), one parameter (`SPar` can be of any type previously defined) or two parameters (`DPar` is a pair). The type `Param` is just a union of those types. A process call is represented by a constructor (`proc`) that contains a process name and its parameters. Finally, the constructor `CSPProcess` defines (syntactically) all possible processes.

3.4.3 Deep Embedding of CSP SOS in FORMULA

Concerning the deep embedding, where the behavioural aspects are completely interpreted in FORMULA, we use an approach similar to those in the literature [Leu01, VMO02]: one-to-one mapping for each firing rule. Before showing these mappings we start by addressing the underlying LTS structure: states, events and transitions.

```
domain CSP_Syntax includes ShallowEmbedding {
 State ::= (p:CSPProcess).
 trans ::= (source:State,event:SigmaTickTau,target:State).
}
```

The constructor `State` captures any possible state (or context) of a CSP process during its execution directly from the syntax domain. A transition is intuitively captured by the constructor `trans` as a triple containing a `source` state, an `event` (captured as presented in the previous section) as label and a `target` state. Note that these constructors are derived because these elements will be generated during the LTS construction. This LTS construction is the main bottleneck of our CML model checker. As FORMULA is interpreted and to build the LTS we have to interpret several rules iteratively, this takes a considerable amount of time. We point our this as a future extension of this work by deriving an optimised implementation from the FORMULA script using Python or Haskell, for example.

Now we start by showing the representation of each firing rule for CSP in terms of FORMULA transitions and states.

No Transition In some languages, there are terminal symbols in the sense of their definitions do not involve the creation of transitions, but just states with no outgoing transitions. In CSP, for instance, Ω and $STOP$ represent a same state meaning deadlock, from where there is no progress. We represent this state in FORMULA by `State(Stop)`

Dynamic Creation of States The existence of a transition between states requires the existence of the source state and causes the (dynamic) creation of the target state (the initial state of a new transition). This is achieved by the general rule `State(nS) :- trans(State(iS), ev, State(nS))`. This rule is important to provide a way of creating an entire path (sequence of transitions).

Skip Recall from Figure 10 the firing rule for $Skip$:

$$\frac{}{Skip \xrightarrow{\checkmark} \Omega}$$

Its translation into FORMULA is quite intuitive as $Skip$ performs ✓ event and leads the system to Ω. We just replace the source and the target states with their respective representations to obtain a ✓-transition as follows.

```
trans(State(Skip),tick,State(Stop)) :- State(Skip).
```

Prefix The prefix has the following firing rule:

$$\frac{}{x : A \to P(x) \ \xrightarrow{a}\ P[a/x]} (a \in A)$$

Its representation in FORMULA is also intuitive as it simply creates a transition labelled with an event to the next behaviour (state):

```
trans(State(Prefix(a,P)),a,State(P)) :- State(Prefix(a,P)).
```

Internal choice The firing rules for the internal choice originate two transitions

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

Their translation originates the following elements in FORMULA:

```
//It creates following states
State(P)   :- State(iChoice(P,Q)).
State(Q)   :- State(iChoice(P,Q))

//It creates the corresponding transitions
trans(State(iChoice(P,Q)),tau,State(P)) :- State(iChoice(P,Q)).
trans(State(iChoice(P,Q)),tau,State(Q)) :- State(iChoice(P,Q)).
```

The existence of an internal choice needs to create the following states and their corresponding transitions.

External choice The external choice rules with internal progress are given by:

$$\frac{P \xrightarrow{\tau} P'}{P \Box Q \xrightarrow{\tau} P' \Box Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \Box Q \xrightarrow{\tau} P \Box Q'}$$

Their representations in FORMULA are given as follows:

```
//It creates states for the constituent parts
State(P)  :- State(eChoice(P,Q)).
State(Q)  :- State(eChoice(P,Q))

trans(State(eChoice(P,Q)),tau,State(eChoice(P_,Q))) :-
        State(eChoice(P, Q)),trans(State(P),tau,State(P_)).
trans(State(eChoice(P, Q)),tau,State(eChoice(P, Q_))) :-
        State(eChoice(P,Q)),trans(State(Q),tau,State(Q_)).
```

Similarly to the previous operator, we also need to create the constituent states for external choice. Note that the premises are added to the right-hand side of the corresponding FORMULA code as they are necessary to generate the transition. The firing rules with antecedent and conditions over the events are given by

$$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'} (a \neq \tau) \qquad \frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'} (a \neq \tau)$$

Their translations produce rules containing the premises and the conditions in the right-hand side.

```
trans(State(eChoice(P,Q)),ev,State(P_)) :-
  State(eChoice(P,Q)),trans(State(P),ev,State(P_)),ev!=tau.
trans(State(eChoice(P,Q)),ev,State(Q_)) :-
  State(eChoice(P,Q)),trans(State(Q),ev,State(Q_)),ev!=tau.
```

Parallelism The firing rules for parallelism with internal progress are given by:

$$\frac{P \xrightarrow{\tau} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P' \underset{X}{\parallel} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} Q'}$$

They are translated into

```
// Required by the premises.
State(P) :- State(par(P, X, Q)).
State(Q) :- State(par(P, X, Q)).

trans(State(par(P,X,Q)),tau,State(par(P_,X,Q))):-
   trans(State(P),tau,State(P_)),State(par(P,X,Q)).
trans(State(par(P,X,Q)),tau,State(par(P,X,Q_))):-
   trans(State(Q),tau,State(Q_)),State(par(P,X,Q)).
```

The rules of asynchronous parallelism are given by

$$\frac{P \xrightarrow{a} P'}{P \underset{X}{\parallel} Q \xrightarrow{a} P' \underset{X}{\parallel} Q}(a \in \Sigma \setminus X) \qquad \frac{Q \xrightarrow{a} Q'}{P \underset{X}{\parallel} Q \xrightarrow{a} P \underset{X}{\parallel} Q'}(a \in \Sigma \setminus X)$$

Note that both have a membership condition to activate the rule. In FORMULA, we avoid this membership interpretation and use FORMULA's base of facts itself as a set. Thus we define a special constructor `lieIn(..., ...)` that characterises when some element `a` lies in a set `X` by simply existing the fact `lieIn(a,X)`. Otherwise, we have `fail lieIn(a,X)`. The definition of `lieIn` and the translation of the asynchronous parallelism are presented as follows.

```
lieIn ::= (ev:Sigma, set:String).

trans(State(par(P,X,Q)),a,State(par(P_,X,Q))) :-
  State(par(P,X,Q)),a!=tau,a!=tick,
  trans(State(P),a,State(P_)),fail lieIn(a, X).
trans(State(par(P,X,Q)),a,State(par(P,X,Q_))) :-
  State(par(P,X,Q)),a!=tau,a!=tick,
  trans(State(Q),a,State(Q_)),fail lieIn(a, X).
```

The firing rule for synchronous parallelism is simpler

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{X}{\parallel} Q \xrightarrow{a} P' \underset{X}{\parallel} Q'}(a \in X)$$

Two processes evolve together only if they agree in the same event that lies in the synchronisation set. The translation produces:

```
trans(State(par(P,X,Q)),ev,State(par(P_,X,Q_))) :-
  State(par(P,X,Q)),ev!=tau,ev!=tick,lieIn(ev,X),
  trans(State(P),ev,State(P_)),trans(State(Q),ev,State(Q_)).
```

Recall the firing rules for parallelism that deal with distributed termination.

$$\frac{P \xrightarrow{\checkmark} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} \Omega \underset{X}{\parallel} Q'} \quad \frac{Q \xrightarrow{\checkmark} Q'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} \Omega} \quad \frac{}{\Omega \underset{X}{\parallel} \Omega \xrightarrow{a} \Omega}$$

They exist only to force both processes terminate together. In our embedding, we just need a rule for the distributed termination.

```
trans(s,tick,State(Stop)) :- s is State(par(Skip,X,Skip)).
```

Hiding The firing rules for hiding are given by

$$\frac{P \xrightarrow{a} P'}{P \backslash A \xrightarrow{\tau} P' \backslash A}(a \in A) \quad \frac{P \xrightarrow{a} P'}{P \backslash A \xrightarrow{a} P' \backslash A}(a \notin A) \quad \frac{P \xrightarrow{\checkmark} P'}{P \backslash A \xrightarrow{\checkmark} \Omega}$$

They are translated into

```
//Required by the premises
State(P) :- State(hide(P, X)).

trans(State(hide(P,X)),tau,State(hide(P_,X))) :-
  State(hide(P,X)),ev!=tick,lieIn(ev, X),
  trans(State(P),ev,State(P_)).
trans(State(hide(P,X)),ev,State(hide(P_,X))) :-
  State(hide(P,X)),ev!=tick,fail lieIn(ev, X),
  trans(State(P),ev,State(P_)).
trans(State(hide(P,X)),tick,State(Stop)) :-
  State(hide(P,X)),trans(State(P),tick,State(P_)).
```

Sequential composition The following firing rules describe the behaviour of the sequential composition operator

$$\frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{\tau} P'; Q}(a \neq \checkmark) \quad \frac{P \xrightarrow{\checkmark} P'}{P; Q \xrightarrow{\tau} Q}$$

The translation to FORMULA produces

```
//Required by the premises
State(P) :- State(seqC(P, Q)).

trans(State(seqC(P,Q)),ev,State(seqC(P_,Q))) :- ev!=tick,
 State(seqC(P,Q)),trans(State(P),ev,State(P_)).
trans(State(seqC(P,Q)),tau,State(Q)) :- State(seqC(P, Q)),
 trans(State(P),tick,State(P_)).
```

Recursion The firing rule for recursion is given by

$$\frac{}{\mu Y \bullet F(Y) \xrightarrow{\tau} F[\mu Y \bullet F(Y)/Y]}$$

The μ construct is just a way to call the process again. This is expressed in FORMULA as a process call in the body of the process. Furthermore, we need more constructors to deal with this. The following code is the translation for recursive processes.

```
ProcDef ::= (name:String,params:Param,proc:CSPProcess).
trans(State(proc(P,pP)),tau,State(PBody)) :-
  State(proc(P,pP)),ProcDef(P,pP,PBody),State(PBody).
State(PBody) :- State(proc(P,pP)),ProcDef(P,pP,PBody).
```

The constructor `ProcDef`(meaning Process Definition and a way of encoding CSP equations as $P(X) = PBody$) is a way of describing in FORMULA all processes that are defined in a CSP specification. It contains a name (of type `String`), a parameter (of type `Param`) and the process body itself (of type `CSPProcess`). The initial state of the firing rule is $\mu Y \bullet F(Y)$. This is captured in FORMULA by `State(proc(P,pP))`, where `pP` are the possible actual parameters of `P`. However the new state $P[\mu p.P/p]$ needs two FORMULA facts to work accordingly: `ProcDef(P,pP,PBody)` (the creation of the new process body substituting all arguments with the values provided by the actual parameters `pP`) and `State(PBody)` (the state building block corresponding to this new process body). As the new state is used in the right-hand side of the previous rule, it must be created beforehand. That is the reason we need the rule `State(PBody):-...` Note that its right-hand side is almost the same as the transition rule, except that here we are creating the state to be used there (a creation only when the actual parameter is available).

3.4.4 Capturing CSP Channels in FORMULA

In CSP channels are useful to define events (or set of events). In FORMULA channels have a similar purpose. For events without data communication, the existence of an event (`BasicEv("ch")`, for example) makes implicit the existence of a channel `ch`. This means that the CSP channel declaration

```
channel ev
```

has no corresponding FORMULA code.

Nevertheless, as FORMULA uses SMT solving to instantiate values, events involving data communication have a different purpose: providing basic facts (probably with uninstantiated values) so that FORMULA can instantiate values to be used in communications. This approach provides a powerful abstraction mechanism for data values. For example, the following channel declaration

```
channel in : Int
```

is represented in FORMULA by

```
primitive Channel ::= (chName:String,chType:Types).
```

The representation is intuitive and contains channel's name and the supported communication type. The `primitive` qualifier establishes that a channel in FORMULA must be given in the partial model. Moreover, all constructors involving the communicated type depend on the corresponding FORMULA channel. For example, the following CSP code shows a process that uses a communication event involving values from an infinite domain

$channel\ in : Int$
$P = in?x \rightarrow Skip$

Its translation to FORMULA produces the following code

```

//Inside the semantic domain of the problem
trans(So,CommEv("in",Int(x)),State(Skip)) :- So is State(CommEv("in",Int(x))),
                                             Channel("in",Int(x)).

//Inside the partial model of the problem
Channel("in",Int(_))
```

It is worth noting that the transition for the prefix construct has already been defined in the semantic domain. However, such a definition works only for basic events (without communication values) or events involving an already known (constant) communication values. When communication values need to be instantiated the channel declaration is necessary as premise to create a transition (or any other element) that depends on it. Furthermore, these codes are placed in different parts of the FORMULA script: the semantic domain contains the rule to generate a new transition, and the partial model contains the fact corresponding to the channel declaration. This separation occurs because the semantic domain manipulates dynamic information whereas the partial model provides all the necessary static information to make FORMULA work . This is also discussed in Section 3.4.7.

3.4.5 Classical Properties in FORMULA

Recall from Section 3.3 that model checking is basically stated as a possible walk-through (breath-first, etc.) in a given LTS. For CSP, such a check includes some classical properties like deadlock, livelock, and nondeterminism. Other properties are checked via explicit refinement[5].

For each domain instance (or model) to be analysed, we must be able to inform which process will be analysed. This is achieved by adding a new constructor with this purpose.

```
domain CSP_Semantics extends CSP_Syntax{
  ...
  // to allow informing the process to be analysed
  primitive GivenProc ::= (name:String).
  State(body) :- GivenProc(name),ProcDef(name,params,body).
}
```

The constructor `GivenProc` allows one to inform which process (actually only its name) will be analysed. Based on that information and on the corresponding process definition, we are able to create the first state and then start the creation of the entire LTS (dynamic states and transitions).

Once the LTS has been created, we can define queries (partial model) capturing properties over the LTS. It is worth noting however that FORMULA only presents a successful analysis when a query is satisfiable; this exactly corresponds to the counterexample provided by model checkers. Therefore, if one wants to find a counterexample in FORMULA, the properties must be stated in such a way that they aim at finding the conterexample. That is, instead of checking for deadlock-freedom we are interested in finding a possible deadlock; the same idea is used to the other classical properties. The encoding of each classical property in FORMULA is almost direct and based on its definition. As introduced later on in this section, the following formal descriptions assume a relation $Reachable(s)$ that holds only whether there is a path from the process equation (definition) to a semantic state s.

- Deadlock - a process is deadlocked if it reaches some state from which it goes nowhere. Furthermore, such a state is not reached by a $\checkmark$-transition (successful termination). This is formally stated by,

  $$\exists s : State \bullet \neg \exists t : Transition \bullet Reachable(s) \wedge t = (s, ev, s'),$$

[5] Deadlock, livelock and nondeterminism are also checked via refinement. However, the process exhibiting the desirable property is internally defined in FDR and compared with the process given by the user.

where $Reachable$ captures all reachable states of the analysed system;

- Livelock - a process has a livelock if it can perform a τ-loop (a loop of internal or τ-transitions). This is formalised by

  $$\neg\exists p : TauPath \bullet Reachable(s) \land p = (s, s),$$

  where $TauPath$ represents a sequence of one or more invisible transitions between two states.

- Nondeterminism - a process is nondeterministic if it it decides to accept or reject the same event. This is similar to say that there are two transitions with the same event from the same state leading to states (with different initial acceptances). A formalisation of nondeterminism is given by

  $$\exists\, t_1, t_2 : Transition \bullet t_1 = (s, ev, s_1) \land t_2 = (s, ev, s_2) \land s_1 \neq s_2 \land$$
  $$Reachable(s_1) \land Reachable(s_2);$$

It is worth pointing out that the above properties are checked by FDR using refinement. That is, processes are analysed against some standard processes that exhibit the desirable properties. Furthermore, FDR checks for deadlock-freedom, livelock-freedom and determinism, whereas we check the existence of deadlock, livelock and nondeterminism. this is due to the purpose of FORMULA queries and because it is easy to find the counterexample.

Before performing the real check of a refinement, FDR generates the LTSs of both processes[6] and applies a *normalisation* to the specification, in the sense that the structure of the LTS is changed for optimization. Afterwards, it compares its LTS of the specification with that of the implementation. The chosen model ($\mathcal{T}$,$\mathcal{F}$ or $\mathcal{FD}$) determines which kind of information the LTS structure contains.

Concerning determinism checking FDR works differently. As deterministic processes are the maximal ones under refinement, and the nondeterministic choice of all deterministic processes is Chaos, one cannot check the determinism of a process P by refinement checking it against some specification in any model X: $Spec \sqsubseteq_X P$. Instead, FDR uses an algorithm that analyses the internal structure of P (it indeed extracts specific transitions of P). This cannot be reproduced using the refinement function of FDR. A detailed description of this algorithm can be found in [Ros10].

Our approach, on the other hand, works directly on the LTS an uses the high level

[6]For processes (specification and implementation) involving parallelism, there is a previous *compilation* stage, where FDR identifies the parallel components and compiles these to explicit state machines to make the comparison easier.

support of FORMULA queries. This makes the check of properties much more close to their definition, as the FORMULA language corresponds to first-order logic. This also shows how powerful is FORMULA to abstract away programmatic details.

The most natural way to capture properties is using clauses establishing constraints over the LTS (built according to the semantic domain rules). To avoid polluting the semantic domain, we use domain extension and define auxiliary definitions and the corresponding clause to each property.

```
domain CSP_Properties extends CSP_Semantics {
  //Determining a reachable state
  reachable                ::= (fS:State).
  reachable(State(PBody)) :- GivenProc(P),ProcDef(P,pPar,PBody).
  reachable(Q)   :- GivenProc(P),ProcDef(P,pPar,PBody),trans(State(PBody),_,Q).
  reachable(Q)   :- reachable(R), trans(R,_,Q).

  //A path of tau-transitions between two states
  tauPath      ::= (iS:State,fS:State).
  tauPath(P,Q) :- trans (P,tau,Q).
  tauPath(P,Q) :- tauPath(P,S),tauPath(S,Q).

  //The acceptances of a process in a given state
  accepts        ::= (iS:State,ev:SigmaTickTau).
  accepts(P, ev) :- trans(P,ev,_), ev != tau.
  accepts(P, ev) :- trans(P,tau,R),accepts(R,ev).

  //Capturing deadlock
  Deadlock := trans(_,_,L),fail trans(L,_,_),fail trans(_,tick,L),reachable(L).

  //Capturing livelock
  Livelock     := reachable(L),tauPath(L,L).

  //Capturing nondeterminism
  Nondeterminism := trans(L,ev,S1),trans(L,ev,S2),S1!=S2,ev1!= tau,
    accepts(S1,ev1), fail accepts(S2,ev1),reachable(S1),reachable(S2).
```

The rule `reachable` captures any state that is reachable by the analysed process. Based on the main process (`GivenProc(P)`) and on its definition (that is, `ProcDef(P,pPar,PBody)`), we calculate all reachable states (starting at `State(PBody)`) by using reflexive-transitive closure: the main process itself is reachable, all main state's neighbour are reachable, and all neighbour of a reachable state is also reachable.

The rule `tauPath` is a FORMULA description to represent a sequence (possibly unitary) of τ-transitions between two states. It is defined in terms of transitive closure.

The rule `accepts` captures the initial acceptances (only visible events) of a process in a given state/context. Thus, `accepts(P,ev)` means the analysed process accepts the visible event `ev` in a state `P` (possibly performing τ-transitions before `ev`).

Each property is almost a direct transcription from its definition considering the structure of the generated LTS and some auxiliary rule. A deadlock, for example, is found if there is an arbitrary (and reachable) state `L`, reached by a transition (`trans(_,_,L)`) that does not mean successful termination (`fail trans(_,tick,L)`) and from where there is no outgoing transition (`fail trans(L,_,_)`).

On the other hand, a livelock is intuitively defined by the existence of a `tauPath` from a reachable state to itself (`reachable(L),tauPath(L,L)`). This is a very simple way to capture the notion of a τ-loop (a cycle containing only τ-transitions).

The nondeterminism is captured by checking the existence of two transitions with a same event (possibly τ-transitions) from the same state `L` (`trans(L,ev,S1)` and `trans(L,ev,S2)`) leading to different states (`S1!=S2`) in which the process can accept (`accepts(S1,ev1)`) or reject (`fail accepts(S1,ev1)`) the same visible event (`ev1!= tau`). The remaining facts `reachable(S1)` and `reachable(S1)` are necessary to guarantee that `S1` and `S2` are reachable by the analysed process. Actually, when finding states (wider contexts) that depend on simpler states (narrower contexts), FORMULA generates auxiliary transitions for narrower contexts as premises for the transitions for the wider contexts. Nevertheless, the query must consider only the wider contexts as they actually represent the LTS of the analysed process. For example, in the analysis of the process $(a \rightarrow SKIP)\backslash a$, FORMULA generates the transitions

```
trans(State(Prefix(BasicEv("a"),Skip)),BasicEv("a"),State(Skip)),
trans(State(Skip),tick,State(Stop)),
trans(State(hide(Prefix(BasicEv("a"),Skip),"{a}"))),tau,State(hide(Skip,"{a}"))),
trans(State(hide(Skip,"{a}")),tick,State(Stop)).
```

However, the first and the second transitions are just premises for the third and fourth transitions (the real transitions of the original process), respectively. Hence, they must be discarded by the nondeterminism property query.

3.4.6 Refinement Checking in FORMULA

Recall from Section 3.3 that traces refinement ($\sqsubseteq_T$) is defined as

$$P \sqsubseteq_T Q \equiv traces(Q) \subseteq traces(P)$$

where the function $traces$ comes from the denotational semantics.

In terms of LTS analysis, traces calculation uses the transitive closure on the transitions from the initial state of the process being analysed to a certain end state. In FORMULA, the transitive close is easily obtained by connecting the final state of a transition with the initial state of another transition. However, we have to consider this approach for two processes (P and Q) that will be compared and use an on demand LTS creation and comparison. Therefore, from the negation of $traces(Q) \subseteq traces(P)$, it suffices we can find some end state of the process Q that cannot be achieved by process P or that it is achieved by different events. In other words, we evolve both processes together on demand (recording the sequence of performed events) and check if the refinement is invalid. In this case, we finish the LTSs construction (because we have found a counterexample).

It is worth pointing out that in the traces model, invisible events do not make sense. Moreover, tools like FDR and PAT optimize the LTS walkthrough process by applying a normalisation step in the specification. Thus, the resulting LTS is changed for optimization. In FORMULA we do that simultaneously through the comparison between specification and implementation.

In our refinement checking implementation in FORMULA we extend the properties domain[7] by defining two objects of it: specification (`Spec`) and implementation (`Impl`). This is a resource of FORMULA that allows both usual domain extension features as well as using renamed instances of a same domain. The constructor `CEPath` has the purpose of detecting if a given event has been performed by the specification just before a given final state (`Q`); it also discards previousτ-transitions. As we analyse simultaneously two processes, the structure of the counterexample (`C_EX`) should contain the initial states of both specification and implementation, an event, and the final states of both specification and implementation. The rules for building the counterexample will be explained by cases later. And the clause `counterexample` clause defines a valid counterexample (representing a witness of an invalid refinement). Provided that the main process (`GivenProc`) and its corresponding definition (`ProcDef`) are defined for `both` specification and implementation, the counter example must have in its first transition calls to those processes (`proc(P,Ppar)`) and (`proc(Q,Qpar)`) as initial states. And the final states of a valid counter example must have `State(Stop)` as final state for both specification and implementation. The inclusion of such final states in the counterexample happens when a situation violating the refinement is found during its construction.

```
domain TrRefinement extends  CSP_Properties as Spec,CSP_Properties as Impl{

 CEPath ::= (iS:Spec.State,event:Spec.SigmaTickTau,fS:Spec.State).
```

[7]This is due to the reuse of the `tauPath` constructor. We could also extend the semantic domain and (re) define a constructor to capture τ-path between two states.

```
CEPath(P,ev,Q)     :- Spec.trans(P,ev,Q),ev!=tau.
CEPath(P,ev2,Q)    :- Spec.tauPath(P,S),Spec.trans(S,ev2,Q),ev2!=tau.

//The conterexample  structure
C_Ex::=(spec:Spec.State,impl:Impl.State,event:Impl.SigmaTickTau,
        specNext:Spec.State,implNext:Impl.State).

//Rules for counterexample construction
// Counterexample definition
counterExample :=
  Spec.GivenProc(P),Spec.ProcDef(P,Ppar,PBody),
  Impl.GivenProc(Q),Impl.ProcDef(Q,Qpar,QBody),
  C_Ex(Spec.State(proc(P,Ppar)),Impl.State(proc(Q,Qpar)),_,_,_),
  C_Ex(_,_,_,Spec.State(Stop),Impl.State(Stop)).

//The main goal
conforms       := counterExample.
}
```

Now we explain the construction of the counterexample by detailing all situations we must capture on demand. In the first situation we consider the creation of the first transition. It is a simple case, as we just use process calls as initial states and an internal action to recover the bodies (states) of each process. This is possible as long as the main processes and their definitions are available in the base of facts.

```
//Building the first transition
 C_Ex(Spec.State(proc(P,pP)),Impl.State(proc(Q,pQ)),tau,
      Spec.State(PBody),Impl.State(QBody)) :-
       Spec.GivenProc(P),Spec.ProcDef(P,pP,PBody),
       Impl.GivenProc(Q),Impl.ProcDef(Q,pQ,QBody).
```

The second situation is the simplest situation and discards internal actions performed by the implementation. As long as there is a previous record in the counterexample structure leading the implementation to a state `S1Q`, and from that state there is a τ-transition, we simply discard it and evolve only the implementation in the counterexample structure.

```
//Tau transitions in the implementation are discarded.
C_Ex(S0P,S0Q,ev,S1P,S2Q) :- C_Ex(S0P,S0Q,ev,S1P,S1Q),Impl.trans(S1Q,tau,S2Q).
```

The third situation handles different sizes in traces of both specification and implementation. Actually, we need to detect if the implementation has a lengthier trace than the specification; that is, the implementation performs a visible event and the specification does not perform any visible event (in the future).

```
//Implementation has a lengthier trace
C_Ex(S0P,S0Q,evI,Spec.State(Stop),Impl.State(Stop)) :-
  Impl.trans(S0Q,evI,_),evI!=tau,evI!=tick,C_Ex(_,_,_,S0P,S0Q),
  fail CEPath(S0P,_,_).
```

The fourth situation deals with the case where both specification and implementation want to perform visible but different events. This represents an invalid refinement situation and we record the event performed by the implementation and stop the counterexample construction. As long as there is a record on the counterexample leading to `S0P` and `S0Q`, from which the implementation evolves via a specific visible event that is not the same event used by the specification to evolve, we record the event performed by the implementation as the final event of the counterexample. Note that these rules are the same, except for the event. This is necessary because FORMULA does not allow direct comparison between events of the specification and of the implementation. The last case concerns successful termination only in the implementation: if the implementation is ready to terminate successfully and the specification does not terminate (in the future), we record the ✓ event as the last event in the counterexample.

```
//Different events originate the final transition.
C_Ex(S0P,S0Q,evI,Spec.State(Stop),Impl.State(Stop)) :-
  Impl.trans(S0Q,evI,_),C_Ex(_,_,_,S0P,S0Q),
  evI=Impl.BasicEv(name),
  fail CEPath(S0P,Spec.BasicEv(name),_).

C_Ex(S0P,S0Q,evI,Spec.State(Stop),Impl.State(Stop)) :-
  Impl.trans(S0Q,evI,_),C_Ex(_,_,_,S0P,S0Q),
  evI=Impl.CommEv(name,data),
  fail CEPath(S0P,Spec.CommEv(name,data),_).

//For the case where impl performs tick
C_Ex(S0P,S0Q,tick,Spec.State(Stop),Impl.State(Stop)) :-
 Impl.trans(S0Q,tick,_),C_Ex(_,_,_,S0P,S0Q),
 fail CEPath(S0P,tick,_).
```

Finally, the last situation captures equal events performed by specification and implementation and records it in the counterexample structure. It is worth noting that we use the constructor `CEPath` to check if the specification performs the event because we have to discard τ-transitions in such a check. Due to the impossibility of comparing events of the specification and of the implementation directly the rule is duplicated for different events.

```
// Equal events were performed. Just record it.
C_Ex(S0P,S0Q,evI,S1P,S1Q) :- CEPath(S0P,evS,S1P),
  Impl.trans(S0Q,evI,S1Q),evI=Impl.BasicEv(name),
  evS=Spec.BasicEv(name),C_Ex(_,_,_,S0P,S0Q),evI!=tau.

C_Ex(S0P,S0Q,evI,S1P,S1Q) :- CEPath(S0P,evS,S1P),
  Impl.trans(S0Q,evI,S1Q),evI=Impl.CommEv(name,data),
  evS=Spec.CommEv(name,data),C_Ex(_,_,_,S0P,S0Q),evI!=tau.
```

3.4.7 Using the model checker directly in Visual Studio

The framework FORMULA allows two execution modes: inside Microsoft Visual Studio and command line based. In both modes one has to provide the entire encoding of CSP semantics as well as the encoding of the process to be analysed. The latter consists of extending (using simple inclusion) the properties domain (if one wants to check classical properties) or the refinement domain (if one wants to check traces refinement), and determining the main process and all necessary facts in a partial model. In case of refinement checking, the domain and the partial model contain necessary information for both specification and implementation. For example, let us consider the analysis of a simple process P given by $a \rightarrow Skip \Box b \rightarrow Stop$ and the refinement check between P and another process Q given by $a \rightarrow Skip \sqcap b \rightarrow Stop$. We perform deadlock check for both of them and the refinement $P \sqsubseteq_{\mathcal{T}} Q$. The encoding for each process is presented as follows.

```
//Domain and partial model defining P
domain PDomain includes CSP_Properties {
 ProcDef("P",nopar,eChoice(Prefix(BasicEv("a"),Skip),
                           Prefix(BasicEv("b"),Stop))).
 conforms := CSP_Properties.Deadlock.
}
partial model P of PDomain{
 GivenProc("P")
}

//Domain and partial model defining Q
domain QDomain includes CSP_Properties {
 ProcDef("Q",nopar,iChoice(Prefix(BasicEv("a"),Skip),
                           Prefix(BasicEv("b"),Stop))).
 conforms := CSP_Properties.Deadlock.
}
partial model Q of PDomain{
 GivenProc("Q")
}
```

For the process P we have a domain (`PDomain`) and a corresponding partial model (`P`). The domain contains a process definition representing a CSP definition for the process P as well as the deadlock check as the main goal. On the other hand, the partial model contains only a fact to establish P as the process to be analysed. The process Q is encoded similarly. We point out that this encoding allows the definition of auxiliary processes in the domain as the mais process is only informed in the partial model. This is an important resource to follow a modular description of a CSP specification.

Concerning the refinement, the encoding in FORMULA is given as follows.

```
domain PRefQDomain includes TrRefinement {
```

```
 Spec.ProcDef("P",nopar,eChoice(Prefix(BasicEv("a"),Skip),
                              Prefix(BasicEv("b"),Stop))).
 Impl.ProcDef("Q",nopar,iChoice(Prefix(BasicEv("a"),Skip),
                              Prefix(BasicEv("b"),Stop))).
 conforms := TrRefinement.conforms.
}
partial model PRefQ of PRefQDomain{
 Spec.GivenProc("P")
 Impl.GivenProc("Q")
}
```

The refinement is also represented by a domain and a corresponding partial model. The domain contains two process definitions establishing the specification and the implementation. Moreover, the conforms clause is defined as the main goal of the `TrRefinement` domain, which checks for the existante of a valid conterexample. The partial model just defined the main processes of the specification and of the implementation. Similarly to the previous encoding, this also allows the use of auxiliary processes for the specification and the implementation.

4 CML embedding in FORMULA

The embedding of CSP in FORMULA, detailed in Section 3 and shows how to build a model checker for CSP based on the its operational semantics. Such an embedding is important mainly because it can be reused in the context of CML. That is, the CML embedding is reuses the CSP embedding with some adjustments and extensions to include more behavioural aspects and data aspects as well. This allows one to create model checkers in a gradual approach.

We present the CML embedding as an extension of the embedding presented in Section 3. We start by showing how to deal with some data aspects and then present the new behavioural constructs and adjustment of those reused from the CSP embedding. We also point out that our embedding follows the structured operational semantics of CML of the deliverable D23.3 [BCC+13].

4.1 State and variables in FORMULA

In CML, states and local variables are similar to Circus, where they become available for manipulation in a specific scope. The most common FORMULA structure to represent a set of components (state and variables) together is a tuple. However, as they vary from specification to specification we have used a recursive structure to represent them: *bindings* (that is, mappings from variables to values). The immediate consequence of such a modelling is the existence of a specific value to be used in components that have been declared but not initialised yet. Such a value (`undef`) is defined in the types definition section of the FORMULA embedding.

```
domain AuxiliaryDefinitions {
  //Types
  UNDEF           ::= {undef}.    //it works like a bottom value for all types
  primitive Int   ::= (v:Integer).
  ...
  Types           ::= UNDEF + Int + ...
  ...
}
```

The representation of bindings is introduced as follows.

```
  // Bindings
  NullBind              ::= { nBind }.
  primitive SingleBind ::= (name: String, val: Types).
  primitive BBinding   ::= (b: SingleBind, rest: Binding).
  Binding              ::= NullBind + BBinding.

  // Operations over bindings.
```

```
//fetches the single bind containing the variable var
fetch ::= (var: String, bind: Binding, b: SingleBind).

//updates the old binding by replacing (or adding) a new single binding to it
upd   ::= (old: Binding, b: SingleBind, new: Binding).

//removes the single binding associated to var from the old binding
del   ::= (old: Binding, var: String, new: Binding).
```

The value `nBind` denotes the null (or empty) binding (base case). The constructor `SingleBind` represents a tuple (var, val) maintaining the association of a value (val) to a variable (var). The constructor `BBinding` represents the inductive case of bindings. Its structure is similar to a list definition: a single bind is the head and another binding is the rest (tail) of the structure. Both empty and non-empty bindings are represented together as the type `Binding`. With this representation, specifications containing (without initialising) none, one (variable $x : int$) or two variables ($x : int$ and $y : int$) have bindings, respectively given by

```
nBind,
BBinding(SingleBind("x",undef),nBind),
BBinding(SingleBind("x",undef),
         BBinding(SingleBind("y",undef),nBind))
```

Concerning binding manipulation, we use representations for the operations of updating, deleting and fetching. Each operation is represented as a relation whose facts are created on demand to activate semantic rules that depend on it.

Consider the following CML specification

```
channels
choose,out : int

process P =
begin
  state v : int := 2
  actions
   TEST = (dcl x : int @ (x := 4;
                 choose.x -> out.(x+v) -> Skip))
  @ TEST
end
```

Its initial binding must contain the bindings `(v,2)` and `(x,undef)` as the variable `x` is initiated only when the assignment `x := 4` is executed. This changes dynamically the binding structure.

```
BBinding(SingelBind("v",2),
          BBinding(SingelBind("x",undef),nBind))
```

It is worth pointing out that the notion of bindings must be carried out along the LTS. To make this possible, we extend the `State` constructor to include such an information as follows:

```
domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // Including binding information into State
  State ::= (b: Binding, procName: String, p: CMLProcess).
  ...
}
```

4.2 User Defined Types in FORMULA

The representation for type definitions in FORMULA is quite intuitive. For example, consider the following type declaration in CML

```
types
 Index = nat
   inv i == i in set {0,1}
 Money = nat
   inv m == m in set {0..5}
```

It introduces the new types `Index` and `Money` whose invariants limit their values to the sets `0,1` and `0..5`, respectively. The most natural way to represent these types in FORMULA is by extending the existing types and using constraints (clauses) over them (in the domain of the analysed process) to be considered in all models of the specification domain. Thus, the resulting embedding is given as follows

```
domain AuxiliaryDefinitions {
  //Types
  UNDEF           ::= {undef}.    //it works like a bottom value for all types
  primitive Int   ::= (v:Integer).
  ...
  primitive Index ::= (Natural).
  primitive Money ::= (Natural).

  Types           ::= UNDEF + Int + ... + Index + Money.
  ...
}

domain CML_SyntaxSpec includes AuxiliaryDefinitions {
 ...
}
```

```

domain CML_SemanticsSpec extends CML_SyntaxSpec {
 ...
}

domain CML_PropertiesSpec extends CML_SemanticsSpec {
 ...
}

domain DependentDomain includes CML_PropertiesSpec {
 ...
 //capturing the constraints defined by invariants over types
 badIndex  := Index(i), i != 0.
 badIndex  := Index(i), i != 1.
 badMoney  := Money(m), m > 5.

 conforms := !badIndex & !badMoney & ...

}
```

It is worth noting that our FORMULA constraints are indeed negation of the invariants. This is more suitable to FORMULA and simplifies the embedding.

4.3 User Defined Values in FORMULA

The representation for user defined values in FORMULA is simpler than user defined types. As they are intended to establish global values (constants), they are represented as `primitive` constructors, whose real values are given in the partial model. For example, consider the following CML code

```
values
 N : nat = 10
 V : nat = 20
```

Its conversion originates the following FORMULA code

```
partial model StartProcModel of DependentDomain {
  ...
  N(10)
  V(20)
  ...
}
```

The translation of each CML element that uses `N` and `V` must use these facts in some way.

4.4 CML Specific Processes Fragments

Although CML reuses some constructs of CSP, some of them are adjusted and new constructs are available only in CML. This section can be viewed as an extension of Section 3.4. The translation follows the structured operational semantics rules of CML presented in [BCC+13]. Furthermore, in Section 3.4 we did not consider state (and variables) information. This is handled in CML by using the notion of *bindings* (mappings from variables to values) – an extra information inserted in the state of the generated LTS. This design has been important to create the CML model checker from the CSP one. Hence, the constructs presented in Section 3.4 implicitly manipulate empty bindings.

4.4.1 Div and Chaos

The Div process originates a transition to itself to represent an auto-loop of invisible transitions. Its translation to FORMULA extends the syntax of basic processes to include `Div` and the semantics domain to include the corresponding transition. Concerning `Chaos` it is only represented as a basic process with no corresponding transition.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  BasicProcess  ::= {Stop, Skip, Chaos, Div}.
  ...
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  //Div
    trans(iS,tau,iS) :- iS is State(st,pN,Div).
  ...
}
```

4.4.2 Input and Output

Inputs and outputs are handled uniformly by using a generic representation for communication involving values. In the syntax domain, `IOComm` is a constructor that handles the real value to be communicated. `IOCommDef` is a constructor to make the corresponding changes in the bindings and `CommEv` is the event to be present in transitions.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  primitive IOComm ::= (id: Natural, chName: String, chExp:String, val: Types).
  primitive CommEv ::= (chName: String, chExp:String, val: Types).
```

```
  IOCommDef        ::= (id: Natural, exp: Types, st: Binding, st_: Binding).
  Sigma            ::= BasicEv + CommEv + IOComm.
  ...
}
```

Concerning the firing rules, we have different rules. For events without communication values, we create a transition whose event is `BasicEv`. When values are involved, we need to obtain values from a channel (using the constructor `Channel`) or from the bindings (using `fetch`). The link between `IOComm` and `IOCommDef` is essential for separating values from the process body. `IOCommDef` is responsible to handle the value and give it to `IOComm`. After that a new `CommEv` is created as the label of the transition.

```
domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // communications
  State(st,pN,P),
  trans(State(st,pN,Prefix(BasicEv(a),P)),BasicEv(a),State(st,pN,P)) :-
      State(st,pN,Prefix(BasicEv(a),P)).

  State(st_,pN,P),
  trans(ini, CommEv(chName,chExp, chType),State(st_,pN,P)) :-
      ini is State(st,pN,Prefix(IOComm(id,chName,chExp,chType),P)),
              Channel(chName,chType), IOCommDef(id,chType,st,st_).

  State(st_,pN,P),
  trans(ini,CommEv(chName,chExp,v), State(st_,pN,P)):-
      ini is State(st,pN,Prefix(IOComm(id,chName,chExp,chType),P)),
      Channel(chName,chType1), chType1 != v, IOCommDef(id,chType,st1,st_),
        fetch(chExp,_,v).
  ...
}
```

4.4.3 Variable Block

Variable block is also implemented in CML by extending the syntactical domain to include a new process fragment and by translating the corresponding operational semantic rules as follows.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  primitive var  ::= (name: String, tName: String, p: CMLProcess).
  primitive let  ::= (name: String, p: CMLProcess).
  ...
  CMLProcess := ... + var + let.
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // variable block begin
  trans(iS,tau,State(st,pName,let(nx,pBody))) :-
      iS is State(st, pName, var(nx, xT, pBody)).
```

```

  // variable block visible
  State(st, pName, P) :- State(st,pName,let(x, P)).
  trans(iS, ev, State(st_, pName,let(x,P_))) :-
      iS is State(st,pName, let(x,P)),
      trans(State(st,pName, P), ev, State(st_,pName, P_)).

  // variable block end
  trans(iS, tau, State(st_,pName,Skip)) :-
      iS is State(l,st,pName,let(x,Skip)), del(_,vName,st_).
  ...
}
```

4.4.4 Sequence

CML sequence has almost the same meaning as sequential composition in CSP. In terms of FORMULA, this correspondence is also true. Thus, there is a constructor in the syntax domain and the rules in the semantic domain.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
 ...
 //Similar to CSP but it uses CMLProcesses
 primitive seqC ::= (lProc : CMLProcess, rProc : CMLProcess).
  ...
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // sequence progress
  State(st,pN,P) :- State(st,pN,seqC(P,Q)), P != Skip.
  State(st_,pN,seqC(P_,Q)),
  trans(iS,ev,State(st_,pN,seqC(P_,Q))) :- iS is State(st,pN,seqC(P,Q)),
      trans(State(st,pN,P),ev,State(st_,_,P_)).

  //sequence end
  State(st,pN,Q),
  trans(iS,tau,State(st,pN,Q)) :- iS is State(st,pN,seqC(Skip,Q)).
  ...
}
```

4.4.5 Nondeterministic Choice

The CML nondeterministic choice has almost the same meaning as the internal choice in CSP. In the FORMULA script we have a constructor in the syntax domain and the rules in the semantic domain.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
 ...
 //Similar to CSP but it uses CMLProcesses
 primitive iChoice       ::= (lProc : CMLProcess, rProc : CMLProcess).
  ...
```

```
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // nondeterministic choice left
  State(st,pN, P),
  trans(State(st,pN,iChoice(P,Q)),tau,State(st,pN,P)) :-
    State(st,pN,iChoice(P,Q)).

  // nondeterministic choice right
  State(st,pN,Q),
  trans(State(st,pN,iChoice(P,Q)),tau,State(st,pN,Q)) :- State(st,pN,iChoice(P,Q)
      ).
  ...
}
```

4.4.6 Guard

The translation of the CML guard construct establishes a process fragment (syntax) to allow such a construct in FORMULA and semantic rules that depend on the evaluation of some boolean expression. The evaluation of boolean expression is performed by reusing the built-in FORMULA support, where relational and boolean expressions are directly converted in conditions that enable the creation of a valid (`guardDef`) or invalid (`guardNDef`) guard definition. Guard definitions are useful to provide a way to FORMULA to know which behaviour to follow based on the guard evaluation.

```
domain AuxiliaryDefinitions{
  ...
  //Guard evaluation to handle boolean expression evaluation
  guardDef      ::= (id: Natural, st: Binding).
  guardNDef     ::= (if: Natural, st: Binding).
}
```

The syntax domain is also adjusted to include CML guards as a process fragment. In this case, we define a general constructor for conditional choice to provide a uniform way to deal with guard and conditional choices in CML.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
 ...
 //Similar to CSP but it uses CMLProcesses
 primitive condChoice  ::= (id: Natural, procTrue: CMLProcess, procFalse:
     CMLProcess).
  ...
}
```

Concerning the semantic domain we define the following firing rules.

```
domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // conditional choice
  State(st,pN,p),
```

```
  trans(iS,tau,State(st,pN,p)) :- iS is State(st,pN,condChoice(condId,p,q)),
                    guardDef(condId,st).
  State(st,pN,q),
  trans(iS,tau,State(st,pN,q)) :- iS is State(st,pN,condChoice(condId,p,q)),
                    guardNDef(condId, st).
  ...
}
```

Note that the rule for conditional choice is replicated and it depends on the existence of a `guardDef` or `guardNDef`. These facts are created in the domain of the process being analysed according to the condition to be evaluated. For example, consider the following action

```
P = [2 > 1] & Skip
```

Its translation to FORMULA originates a process definition whose conditional choice can behave as Skip or Stop. The expression to be evaluated is translated directly to FORMULA and is a premise to create a `guardDef` fact that will trigger the correct conditional choice rule.

```
domain DependentDomain extends CML_PropertiesSpec {
  ...
  ProcDef("P",nopar,condChoice(1,Skip,Stop))).
  guardDef(1,nBind) :- 2 > 1.
  ...
}
```

4.4.7 External Choice

CMl contains two operators for external choice: `[]` and `[+]`. They are respectively represented in formula by `eChoice` and `extraChoice`. The firing rule for `[]` establishes a transition in which the associated binding is copied to each constituent process and the operator changes to `[+]`. The firing rules for `[+]` define the real behaviour of the external choice. The definition of `extraChoice` and the rules of external choice are described as follows:

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  extraChoice    ::= (lSt: Binding, lProc: CMLProcess, rSt: Binding, rProc:
      CMLProcess).
  ...
  CMLProcess := ... + extraChoice.
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // P [] Q (external choice begin)
  State(st,name,P),
  State(st,name,Q),
  State(nBind,name,extraChoice(st,P,st,Q)),
```

```
  trans(iS,tau,State(nBind,name,extraChoice(st,P,st,Q))) :-
      iS is State(st,name,eChoice(P,Q)).

  //external choice skip
  State(st1,name,Skip),
  trans(iS,tau,State(st1,name,Skip)) :-
      iS is State(st,name,extraChoice(st1,Skip,st2,_)).
  State(st2,name,Skip),
  trans(iS,tau,State(st2,name,Skip)) :-
      iS is State(st,name,extraChoice(st1,_,st2,Skip)).

  //external choice silent
  State(st3,pName_,P_),
  State(st,pN,extraChoice(l,st3,P_,st2,Q)),
  trans(iS,tau,State(st,pN,extraChoice(l,st3,P_,st2,Q))) :-
      iS is State(st,pN,extraChoice(st1,P,st2,Q)),
      trans(State(st1,pName,P),tau,State(st3,pName_,P_)).

  State(st3,qName_,Q_),
  State(st,pN,extraChoice(st1,P,st3,Q_)),
  trans(iS,tau,State(st,pN,extraChoice(st1,P,st3,Q_))) :-
      iS is State(st,pN,extraChoice(st1,P,st2,Q)),
      trans(State(st2,qName,Q),tau,State(st3,qName_,Q_)).

  //external choice end
  State(st3,pName,P_),
  trans(iS,ev,State(st3,pN,P_)) :-
      iS is State(st,pN,extraChoice(st1,P,st2,Q)),
      trans(State(st1,pName,P),ev,State(st3,pName,P_)),ev != tau.
  State(st3,qName,Q_),
  trans(iS,ev,State(st3,pN,Q_)) :-
      iS is State(st,pN,extraChoice(st1,P,st2,Q)),
      trans(State(st2,qName,Q),ev,State(st3,qName,Q_)),ev != tau.
  ...
}
```

4.4.8 Parallel

In a similar way to the external choice, parallelism in CML is represented by two constructors: one for the begin and another for independent, synchronised and end (following the terminology introduced in the Deliverable D23.3). Thus, we use a syntactical (`parll`) and a semantic (`par`) operators.

We also had to provide implementation for merging bindings to be used by the parallel. The syntax domain contain these definitions.

```
domain AuxiliaryDefinitions{
 ...
 primitive Set      ::= (SetDef).    //sequence
 EmptySet           ::= {empty}.
 primitive SetCont  ::= (head:Types,tail:SetDef).
 SetDef             ::= EmptySet + SetCont.
 aSet               ::= (SetDef).
 ...
 //merge
```

```
 merge   ::= (st1: Binding, lVars: String, st2: Binding, rVars: String, stF:
     Binding).
 merge(bindL,setL,setR,bindR,bindRes) :-  filter(bindL, setL, bindRes1),
                  filter(bindR, setR, bindRes2),
                  unionB(bindRes1, bindRes2, bindRes).

 filter  ::= (b: Binding, vars: aSet, st2: Binding).
 filter(bind,set,bindR) :-  bind = BBinding(SingleBind(vN,vVal),restB),
              set = aSet(vN,empty),
              bindR = BBinding(SingleBind(vN,vVal),nBind).

filter(bind, set, nBind) :- bind = BBinding(SingleBind(vN, vVal), restB),
              set = aSet(vN_, empty), vN != vN_.

filter(bind, set, bindR) :- set = aSet(vN, restS),
              filter(bind, aSet(vN, empty), bind1),
              filter(bind, restS, bind2),
              unionB(bind1, bind2, bindR).

unionB  ::= (bindX:Binding, bindY: Binding, bindZ:Binding).
unionB(nBind,nBind,nBind).
unionB(nBind,Sy,Sy) :- Sy is Binding(_,_,_).
unionB(Sx,nBind,Sx) :- Sx is Binding(_,_,_).
unionB(BBinding(SingleBind(varX,valX),S),
       BBinding(SingleBind(varY,valY),nBind),
       BBinding(SingleBind(varX,valX),S_)) :-
        BBinding(SingleBind(varX,valX),S_)), varX != varY,
        union(S,BBinding(SingleBind(varY,valY),nBind),S_).
unionB(BBinding(SingleBind(varX,valX),S),
       BBinding(SingleBind(varY,valX),nBind),
       BBinding(SingleBind(varX,valX),S_)) :-
        BBinding(SingleBind(varX,valX),S_)), varX = varY
        S_=S.

}

domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  //the semantic parallelism
  primitive par   ::= (lSt: Binding, lProc: CMLProcess,
               SyncS : String, rSt: Binding, rProc: CMLProcess).
  //the syntactical parallelism
  primitive parll ::= (lProc : CMLProcess, lVars: String,
                       SyncS : String, rVars: String, rProc : CMLProcess).
  lStVars         ::= (refName: String, vName: String).
  rStVars         ::= (refName: String, vName: String).


  ...
}
```

Concerning the semantic rules, we have provided operations for merging bindings manipulated by the constituent processes of the parallelism. These are presented as follows.

```
domain CML_SemanticsSpec extends CML_SyntaxSpec {
 ...
 //the syntactical parallelism (par) originates the semantic one (parll)
 //and all necessary premises
 State(st,nP,P) :- State(st,nP,parll(P,lV,X,rV,Q)).
```

```
 State(st,nP,Q) :- State(st,nP,parll(P,lV,X,rV,Q)).

 //parallelism begin
 State(st,nP,par(st,P,X,st,Q)),
 trans(iS,tau,State(st,nP,par(st,P,X,st,Q))):- iS is State(st,nP,parll(P,lV,X,rV,
     Q)).

 //parallel independent
 trans(iS,ev, State(st,name,par(st_,P_,X,st_,Q))) :- iS is State(st,name,par(st,P
     ,X,st,Q)),
  trans(State(st,nP,P),ev,State(st_,nP,P_)),fail lieIn(ev, X).

 trans(iS,ev, State(st,name,par(P,st_,X,st_,Q_))) :- iS is State(st,name,par(st,P
     ,X,st,Q)),
  trans(State(st,nQ,Q),ev,State(st_,nQ,Q_)),fail lieIn(ev, X).

 //parallel synchronised
 trans(iS,ev,State(par(st_,P_,X,st_,Q_))) :- iS is State(par(st,P,X,st,Q)),
    trans(State(st,nP,P),ev,State(st_,nP,P_)),
    trans(State(st,nQ,Q),ev,State(st_,nQ,Q_)), lieIn(ev, X).

 //parallel end
 State(st,pN,Skip),
 trans(iS,tau,State(st,pN,Skip)) :- iS is State(st,pN,par(st,Skip,X,st,Skip)).
  ...
}
```

4.4.9 Hiding

The hiding is almost the same as in CSP, where the process depends on facts that say if an event belongs to a specific set (`lieIn` facts). The translation of hiding is illustrated as follows.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  primitive hide  ::= (proc : CMLProcess, hideS : String).
  ...
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  //hiding general to create the premise
  State(st,pN,p) :- State(st,pN,hide(p,X)).

  //hiding internal
  State(st_,pN,hide(P_,X)),
  trans(iS, tau, State(st_,pName,hide(P_, X))) :- iS is State(st,pN,hide(P,X)),
    trans(State(st,pName,P),ev,State(st_,pName,P_)), lieIn(ev, X).

  // hiding visible
  State(st_,pN,hide(P_, X)),
  trans(State(st,pN,hide(P,X)),ev, State(st_,pN,hide(P_, X))) :-
    State(st,pN,hide(P,X)),
    trans(State(st,pN,P),ev,State(st_,pN,P_)), fail lieIn(ev, X).
  ...
}
```

The events `lieIn` depend on the events used in the process body. For example, consider the following process

```
P = (a -> Skip)\{a}
```

Its translation to FORMULA results in a process and in a list of `lieIn` facts to provide all premises for the firing rules of hiding.

```
domain DependentDomain extends CML_PropertiesSpec {
  ...
  ProcDef("P",nopar,hide(Prefix(BasicEv("a"),Skip),"{a}")).
  lieIn(BasicEv("a"),{a}).
  ...
}
```

4.4.10 Recursion

Implementation of CML recursion in FORMULA is similar to that for CSP. The special constructor `proc` represents a process call that is replaced by the suitable process body when necessary (via an internal transition). The FORMULA code for recursion is presented as follows

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  primitive proc  ::= (name : String, p: Param).
  ...
}

domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // Call reusing state
  trans(n,tau,State(st,P,PBody)) :- n is State(st,P,proc(P,pP)),
    State(st,P,PBody), ProcDef(P,pP,PBody).

  //The body of a process is a call to another process
  State(st,name2,PBody),
  trans(n,tau,State(st,name2,PBody)) :- n is State(st,name1,proc(name2,pP)),
    ProcDef(name2,_,PBody).
  ...
}
```

Consider the following recursive process

```
P = a -> P
```

Its translation to FORMULA results in a process that calls itself.

```
domain DependentDomain extends CML_PropertiesSpec {
  ...
  ProcDef("P",nopar,Prefix(BasicEv("a"),proc("P",nopar))).
  ...
}
```

4.4.11 Assignment and Operations

Assignment are viewed as actions that change values of variables. In FORMULA, assignments are represented by two constructs: one identifying the CML assignment and another containing the variable change. The former is present in a process fragment whereas the latter manipulates bindings to make the necessary changes.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  primitive assign   ::= (id: Natural).
  assignDef          ::= (id: Natural, st: Binding, st_: Binding).
  ...
  CMLProcess := ... + assign.
}
```

The constructor `assign` represents a process fragment and has an identifier. The constructor `assignDef` is associated to its assignment fragment through the identifier and contains two bindings: one before the assignment (`st`) and another after the assignment (`st_`). Let's consider a simple CML process example, where its main action declares a local variable, assigns a value to it and behaves like Skip.

```
process P =
begin
    @(dcl x : int @(x := 4; Skip))
end
```

Its translation to FORMULA is given as follows

```
domain DependentDomain includes CML_PropertiesSpec {
  ProcDef("P",nopar,seqC(assign(1),Skip)).
  assignDef(1,BBinding(SingleBind("x",valX),noBind),BBinding(SingleBind("x",Int
      (4)),noBind)).
  ...
}
```

As the process manipulates only one variable (`x`), the bindings contain only one element. The process definition is a sequential composition whose first action is an assignment and the second action is Skip. The corresponding assignment definition has the same identifier and changes the old value (represented by `valX`) with the intended value (`Int(4)` in FORMULA).

The representation of the firing rule for assignments is given as follows:

```
domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  // Assignment
    trans(n,tau,State(st_,pN,Skip)) :- n is State(st,pN,assign(id)),assignDef(id,
        st,st_).
  ...
}
```

The existence of an assignment and its corresponding definition enables the creation of an invisible transition from the assignment fragment to a state whose binding contains the effect of the assignment and the action is Skip.

Operations are also represented by more than one constructor: one for syntactical purposes, one for establishing operation's effect and two others to represent the enabling condition when it is valid or not. This approach to represent precondition evaluation in two ways has been used to avoid interpretation of operation's precondition.

```
domain CML_SyntaxSpec includes AuxiliaryDefinitions {
  ...
  operation    ::= (name: String).
  operationDef ::= (name: String, st: Binding, st_: Binding).
  preOpOk      ::= (name: String, st: Binding).
  preOpNOk     ::= (name: String, st: Binding).
  ...
  CMLProcess := ... + assign + operation.
}
```

The representation of the firing rule for assignments is given as follows:

```
domain CML_SemanticsSpec extends CML_SyntaxSpec {
  ...
  State(st_, Q),
  trans(iS,tau,State(st_,Q)) :- preOpOk(name,st),iS is State(st, seqC(schema(name
      ),Q)),
                                schemaDef(schN, st, st_).
  State(st, Chaos),
  trans(iS,tau,State(st,Chaos)) :- iS is State(st,seqC(schema(name), Q)),
                                   preOpNOk(schN, st).
  ...
}
```

4.4.12 VDM Types and Collections

In this section we present the embedding of the VDM types and collections in FORMULA. We present a hybrid embedding of the VDM types and collections of a CML specification in terms of FORMULA. The hybrid embedding is because some basic VDM types and operators can be directly available in FORMULA

Type name	VDM type	FORMULA type
Integers	**int**	NegInteger, PosInteger, Integer
Naturals	**nat**	Natural
Characters	**char**	String
Strings	**seq of char**	String
Reals	**real**	Real
Booleans	**bool**	Boolean
Basic	?	Basic
Any	?	Any
Tuples	tuple	constructors
Records	record	constructors
Sets	**set of** T	Interpreted
Sequences	**seq of** T	Interpreted
Mapping	**map** A **to** B	Interpreted

Table 6: Correspondence between VDM and FORMULA types

whereas others need interpretation to become available. Table 6 shows a preliminary correspondence between VDM and FORMULA. The correspondence will be given in terms of the operators supported by the respective types. This is because FORMULA supports some types that are not supported by VDM and vice-versa, and even for directly corresponding types, FORMULA does not support some operators available in VDM.

From Table 6 we can see that several types have a direct correspondence between VDM and FORMULA. However, we need to further detail this correspondence in terms of the available and corresponding operators. Table 7 has the correspondence between VDM and FORMULA operators.

For those VDM operators that do not have a corresponding FORMULA counterpart, we have to provide an interpretation. Thus, for example, let's explain how the absolute value (abs), directly available in VDM, can be obtained in FORMULA. First, we have to recall that FORMULA works similarly to Prolog in the sense that everything is made of facts that are instances of relations. So, the VDM function

$$abs : real \rightarrow real$$

becomes the FORMULA relation (construtor)

```
abs       ::= (inp:Real, res:Real).
abs(x, x) :- x is Real, x >= 0.
abs(x, y) :- x is Real, x < 0, y = - x.
```

Operation name	VDM operator	FORMULA operator
The numeric types		
Unary minus	- x	- x
Sum	x + y	x + y
Difference	x - y	x - y
Product	x * y	x * y
Division	x / y	x / y
Less than	x < y	x < y
Greater than	x > y	x > y
Less or equal	x <= y	x <= y
Greater or equal	x >= y	x >= y
Equal	x = y	x = y
Not equal	x <> y	x != y
	Character	String
Equal	c1 = c2	c1 = c2
Not equal	c1 <> c2	c1 != c2
Record types		
Field select	r.i	r.i
Equality	r1 = r2	r1 = r2
Inequality	r1 <> r2	r1 != r2
Is	is_A(r1)	r1 = A(_)
Union/optional types		
Equality	t1 = t2	t1 = t2
Inequality	t1 <> t2	t1 != t2

Table 7: Correspondence between VDM and FORMULA basic operations

In the first line we introduce the construtor `abs` with two real numbers, one named `inp` (standing for input) and one named `res` (standing for result). The other two lines capture the definition of the absolute value, where $|x| = x$ (when $x \geq 0$) and $|x| = -x$ (when $x < 0$). To use the result of such a calculation, one has just to use the field select operator (.) suffixed by the name `res`. So, from an `abs(x, y)` one can use `y` directly or use `absN is abs(x, y)` and apply the field select operator to `absN` (or `absN.res`).

Similarly to the absolute value operator, we can encode the floor operation as

```
floor       ::= (inp:Real, res:Integer).
floor(x, y) :- x is Real, y is Integer, y <= x, x < y + 1.
```

The remainder operation is obtained directly from its mathematical definition.

```
rem       ::= (inp1:Integer, inp2: Integer, res:Integer).
rem(x, y, z) :- x is Integer, y is Integer, y > 0, z = x - y * (x / y).
```

Similarly to the remainder operation, the modulus operation is obtained directly from its mathematical definition.

```
mod       ::= (inp1:Integer, inp2: Integer, res:Integer).
mod(x, y, z) :- x is Integer, y is Integer, y > 0, f = floor(x/y, r), z = x - y
* r.
```

It is worth noting that `x rem y` and `x mod y` are the same if the signs of x and y are the same, otherwise they differ and rem takes the sign of x and mod takes the sign of y.

The Boolean type VDM supports booleans through its **bool** primitive data type with the traditional boolean operators. Let a and b be booleans: negation (**not** b), conjunction (a **and** b), disjunction (a **or** b), implication (a => b), biimplication (a <=> b), equality (a = b), and inequality (a <> b).

In FORMULA, booleans only support directly negation, equality, inequality, conjunction and disjunction. Booleans are treated differently in three distinct situations. The first is as the type (Boolean), one can only use the equality (=) and inequality (!=) operators. In rules (second situation), we have booleans as facts. As facts, we have conjunction (as a comma). For example: for a **and** b we have

```
Rule :- a, b.
```

That is, `Rule` only holds whenever the facts `a` and `b` are present (hold) in the database of facts. For disjunction (by splitting a rule). For example: for a **or** b we have

```
Rule :- a.
Rule :- b.
```

Boolean operator	VDM expression	FORMULA query
Negation	**not** b	Query := !b.
Conjunction	b1 **and** b2	Query := b1 and b2.
Disjunction	b1 **or** b2	Query := b1 or b2.

Table 8: Correspondence between VDM and FORMULA booleans

That is, `Rule` holds whenever the fact `a` is present (holds) in the database of facts. The same occurs in an independent statement concerning the fact `b`. This means disjunction in FORMULA based on facts. For negation (`fail` a). For example: for **not** a we have

```
Rule :- fail a.
```

It is worth observing that the `fail` construct requires some prerequisites to be used successfully. The most important of all is that the rule must be stratisfied [JSD+09]. In general terms this means that a fact $f(p_1, \ldots, p_k)$ can only be used with a `fail` construct whether none of its parameters $p_1, \ldots, p_k$ are found in the head of the rule nor the fact $f(.)$ itself cannot be created by another rule creating a cycle between these rules. The CML model checker satisfies this requirement easily, except for guards where we had to have a fact corresponding to the positive evaluation of a guard and a complementary fact related to the negative evaluation of the same guard.

Finally we can have booleans inside queries (third situation). Now we are able to use the FORMULA boolean operators `and` (conjunction), `or` (disjunction), and `!` (negation), and thus we have a direct correspondence with VDM as illustrated in Table 8.

To be able to represent the state part of CML specifications more flexibly we created a "super" type that is a disjoint union of all supported types (In what follows we simply illustrate this).

```
    primitive Int       ::= (v:Integer).
    primitive Nat       ::= (v:Natural).
    primitive Str       ::= (v:String).
    primitive IR        ::= (v:Real).
    Types               ::= Int + Nat + Str + IR.
```

Set types VDM supports sets of any primitive or user-defined data type. Thus we use **set of** T meaning "the set of elements of type T"[8].

[8] Sets of user created types can be obtained by extending the base type `Types`.

FORMULA does not support sets directly. Thus we need to give a deep embedding (interpretation) of sets into FORMULA. To this end FORMULA provides us with a recusive type that can be used to represent sets, sequences, and mapping. For sets we have:

```
NullSet       ::= { empty }.
Powerset      ::= NullSet + aSet.
primitive TS  ::= (t:Types).
aSet          ::= (anElement: TS, rSet: Powerset).
```

The constructor `NullSet` is a enumeration type that introduces the empty set (`empty`). This empty element is type independent and can be used for any set of a specific type T. A set is captured by the type `Powerset` that can contain two elements: an empty set or a set (`aSet`). The constructor `aSet` is the way we capture a single element (`anElement`) a given set of type `TS` (Recall that we have created a "super" type that can represent any FORMULA directly supported data type) and the rest of the set (by a recursive definition) is given by another `Powerset` element. It is worth observing that, from the "super" type we use inside a set, our sets can be heterogenous. That is, we can represent sets as: $\{1, “vdm'', true\}$. On the other hand, with such a definition we do not support sets of sets as well as set compreehensions (These can be done but have been left for future work).

Concerning set operations we present some encodings (An exhaustive encoding is straightforward and has been left for future work). The first set operation is membership that is available in VDM as "e **in set** s", where e is an element of type T and s is a set of type T.

In FORMULA membership becomes the following constructor and rules:

```
member               ::= (elem: TS, set: Powerset).
member(x, aSet(x, S)) :- aSet(x, S).
member(x, aSet(y, S)) :- aSet(y, S), x != y, member(x, S).
```

The constructor `member` can create facts relating single elements of type `TS` to set elements. The definition of `member` is given recursively by considering first the base case $x \in \{x, \ldots\}$ and then the recusive situation $x \in \{y, \ldots\} \equiv x \in \{\ldots\}$ (if $x \neq y$). Note that to create `member` facts we need facts in the right-hand sides of the rules. In the base case, we need to find a set `aSet(x, S)` as a fact. In the recusive case, we need to find a set (`aSet(y, S)`) and a membership relation (`member(x, S)`) as well as facts to create the new fact (`member(x, aSet(y, S))`). The relation `x != y` is trivially handled as long as the variables `x` and `y` are bound to facts.

The other important operation about sets is the union between two sets, possibly resulting in a new set. In VDM it is simply stated as "s1 **union** s2", for sets s1 and s2. In FORMULA, similarly to the previous case of the membership relation, we have to create a new construct and corresponding rules to interpret the union of sets correctly.

```
union                      ::= (setX: Powerset,
                                setY: Powerset,
                                setZ: Powerset).
union(empty, empty, empty).
union(empty, S, S)         :- S is aSet(_, _).
union(S, empty, S)         :- S is aSet(_, _).
union(Sx, Sy, X)           :- Sx is aSet(x, S), Sy is aSet(y, empty),
                              y = x, X = aSet(x, S).
union(Sx, Sy, X)           :- Sx is aSet(x, S), Sy is aSet(y, empty),
                              y.t.v < x.t.v, X = aSet(y, aSet(x, S)),
                              fail member(y, S).
union(Sx, Sy, X)           :- Sx is aSet(x, S), Sy is aSet(y, empty),
                              x.t.v < y.t.v, union(S, Sy, X_),
                              X = aSet(x, X_).
union(S, Sy, X_)           :- T != empty, Sy is aSet(x, T),
                              union(S, aSet(x, empty), X), union(X, T, X_).
```

The most trivial fact of all about set union is that $\emptyset \cup \emptyset = \emptyset$. A direct consequence of having a $\emptyset$ is that it represents the zero property of union. Thus, we have that $\emptyset \cup S = S \cup \emptyset = S$. If the input sets are not empty then we have a number of situations to consider in FORMULA. But before to go on, it is worth pointing out that—to minimize the number of facts created towards set operations—we consider sets as ordered collections (partial orderings). Because of such an ordering, we need to consider the union with singleton sets to put the element in the right slot in the recursive structure. We have three situations: (i) the elements are equal. We do not create a new set (`y = x, X = aSet(x, S)`); (ii) the element in the singleton set (`y`) is less than (`y.t.v < x.t.v`) the current element (`x`) of the set being considered. We put the `y` before `x` in the set resulting set `X = aSet(y, aSet(x, S))`; and (iii) the element in the singleton set (`y`) is greater than (`x.t.v < y.t.v`) the current element (`x`) of the set being considered. We make a recursive call that puts `y` is the right place. Finally the general situation is based on the associativity of set union: $S \cup (\{x\} \cup T) = (S \cup \{x\}) \cup T$.

Complementing set union, we consider set intersection. In VDM it is simply stated as “s1 **inter** s2”, for sets s1 and s2. In FORMULA, it is defined like union, requiring a new constructor and several rules.

```
inter                        ::= (setX: Powerset,
                                  setY: Powerset,
                                  setZ: Powerset).
inter(empty, empty, empty).
inter(empty, Sy, empty)      :- Sy is aSet(_, _).
inter(Sx, empty, empty)      :- Sx is aSet(_, _).
inter(Sx, Sy, Sy)            :- Sx is aSet(x, S), Sy is aSet(x, empty).
inter(Sx, Sy, empty)         :- Sx is aSet(x, empty), Sy is aSet(y, empty),
                                x != y.
```

```
inter(Sx, Sy, X)            :- S != empty, Sx is aSet(x, S),
                               Sy is aSet(y, empty),
                               inter(aSet(x, empty), aSet(y, empty), X1),
                               inter(S, aSet(y, empty), X2),
                               union(X1, X2, X).
inter(S, Sy, X)             :- S != empty, T != empty, Sy is aSet(x, T),
                               inter(S, aSet(x, empty), X1),
                               inter(S, T, X2), union(X1, X2, X).
```

Like set union, the intersection between empty sets is an empty set ($\emptyset \cap \emptyset = \emptyset$). Complementarily to set union, the intersection with an emptyset results in an empty set ($\emptyset \cap S = S \cap \emptyset = \emptyset$). As set intersection means possibly removing elements from the input sets (those that are different), we consider three situations: (i) the intersection between the set $\{x\} \cup S$ and the singleton set $\{x\}$ equals $\{x\} \cup S$ (ii) the intersection of singleton sets results in an empty set when the elements are different ($\{x\} \cap \{y\} = \emptyset$, if $x \neq y$); (iii) the intersection $(\{x\} \cup S) \cap \{y\}$ equals $(\{x\} \cap \{y\}) \cup (S \cap \{y\})$ by the distributivity of $\cap$ over $\cup$; and finally (iv) $S \cap (\{x\} \cup T) = (S \cap \{x\}) \cup (S \cap T)$ by distributivity of $\cap$ over $\cup$ again.

Another set operation we consider is set difference. In VDM it is simply stated as “s1 $\setminus$ s2”, for sets s1 and s2. In FORMULA, it is defined like the previous operations, requiring a new constructor and several rules.

```
diff                      ::= (setX: Powerset,
                               setY: Powerset,
                               setZ: Powerset).
diff(empty, empty, empty).
diff(empty, Sy, empty)    :- Sy is aSet(_, _).
diff(Sx, empty, Sx)       :- Sx is aSet(_, _).
diff(Sx, Sy, S)           :- Sx is aSet(x, S), Sy is aSet(x, empty).
diff(Sx, Sy, aSet(x, X))  :- Sx is aSet(x, S), Sy is aSet(y, empty),
                             x != y, diff(S, aSet(y, empty), X).
diff(S, Sy, X)            :- S != empty, T != empty, Sy is aSet(x, T),
                             diff(S, aSet(x, empty), X1), diff(S, T, X2),
                             inter(X1, X2, X).
```

The first rule is the trivial one: $\emptyset \setminus \emptyset = \emptyset$. The second rule comes from the fact set difference cannot remove elements from the empty set ($\emptyset \setminus S = \emptyset$). Complementarily, the empty set does not change the original set ($S \setminus \emptyset = S$). Before the last general rule, we have two rules that deal with singleton sets: (i) when the elements are equal and it is removed from the resulting set ($(\{x\} \cup S) \setminus \{x\} = S$); (ii) when the initial elements are different we recurse to consider the other elements as well ($(\{x\} \cup S) \setminus \{y\} = S \setminus \{y\}$, if $x \neq y$). The last rule states the general situation: $S \setminus (\{x\} \cup T) = (S \setminus \{x\}) \cap (S \setminus T)$.

Our last set based operation is the subset relation. In VDM it is written as “s1 **subset** s2”. In FORMULA we have:

```
subs                ::= (setX: Powerset, setY: Powerset).
subs(empty, empty).
subs(empty, Sy)     :- Sy is aSet(_, _).
```

```
    subs(Sx, Sy)          :- Sx is aSet(x, empty), Sy is aSet(x, S).
    subs(Sx, T)           :- Sx is aSet(x, S), subs(aSet(x, empty), T),
                             subs(S, T).
```

The subset relation requires less rules to be captured in FORMULA. Again the first one the most basic rule: $\emptyset \subseteq \emptyset$. A direct consequence is the next rule: $\emptyset \subseteq S$ (for any set S). The third rule concerns the case of the singleton set: $\{x\} \subseteq (\{x\} \cup S)$. And the last rule is the general case: $(\{x\} \cup S) \subseteq T = (\{x\} \subseteq T) \wedge (S \subseteq T)$.

Our last rules concerning sets are a bit curious because they are simply stated to decompose compound set in terms of its internal elements. This is necessary to allow the previous rules to work correctly.

```
    aSet(y, empty),
    aSet(x, S)                :- aSet(y, aSet(x, S)).
```

The previous rules simply state that from the set $\{y, x\} \cup S$ (as a fact) we may decompose it as the sets $\{y\}$ and $\{x\} \cup S$ as new facts. Obviously that the new created set $\{x\} \cup S$ can activate this rule again until the set S becomes empty.

Sequences type Sequences are interpreted in FORMULA like sets because we only have the recursive structure to capture these more elaborated data types. But sequences, differently from sets, are more easily captured because they can repeat internal elements and thus do not need a partial ordering to minimize its representation. Unfortunately, sequences can become infinite very easily, contrary to sets. In the case of a set `S`, the new element `Powerset of S` is only infinite if `S` is. But for sequences, it suffices that the base set be nonempty. Our solution is to consider a bound (`SBound`) in the number of elements that can constitute a sequence.

```
  primitive SBound  ::= (Natural).
```

The recursive sequence representation follows directly from the recursive set representation only differing the names of the constructors.

```
    NullSeq  ::= { empty }.
    Seq      ::= NullSeq + aSeq.
    aSeq     ::= (anElement: TS, rSeq: Seq).
```

In this document we describe four basic sequence operators: cardinality (**len** in VDM), head (**hd** in VDM), tail (**tl** in VDM), and concatenation (^ in VDM).

We start with cardinality. It is very easily captured and similar to functional programming.

```
card          ::= (seqX: Seq, c: Natural).
card(empty, 0).
card(Sx, n_)    :- Sx is aSeq(x, S), card(S, n), n_ = n + 1,
                   n_ <= L, SBound(L).
```

The first rule corresponds to $len\ [] = 0$. The second rule corresponds to the traditional recursive definition $len\ ([x]\hat{}S) = 1 + len\ S$, except for the bound.

The head of a sequence is trivially defined. As long as the sequence has at least one element we can obtain its head as the second element of the constructor `hd` ($hd([x]\hat{}S) = x$).

```
head       ::= (seqX: Seq, h: TS).
head(Sx, x) :- Sx is aSeq(x, S).
```

Similarly to the head of a sequence, its tail is easily obtained.

```
tail              ::= (seqX: Seq, seqT: Seq).
tail(empty, empty).
tail(Sx, S)         :- Sx is aSeq(x, S).
```

The first rule is the base case: $tl\ [] = []$. And the general situation ($tl([x]\hat{}S) = S$) is described by the last rule.

Sequence concatenation is more complex than the previous operations because it creates new sequences similarly to set union. The only exception is that we do not need to worry about element repetition. Another difference with regards to traditional sequence concatenation is that we need to consider the cardinality of the resulting sequence beause it is bound.

```
conc                 ::= (seqX: Seq, seqR: Seq, seqT: Seq, c: Natural).
conc(empty, empty, empty).
conc(empty, Sx, Sx, n) :- Sx is aSeq(x, S), card(aSeq(x, S), n), n <= L,
                          SBound(L).
conc(Sx, empty, Sx, n) :- Sx is aSeq(x, S), card(aSeq(x, S), n), n <= L,
                          SBound(L).
aSeq(x, X),
conc(Sx, Sy, X_, n_)   :- Sx is aSeq(x, S), Sy is aSeq(y, T), card(X, n),
                          conc(S, aSeq(y, T), X, n), X_ = aSeq(x, X),
                          n_ = n + 1, n <= L, SBound(L).
```

The first rule is the trivial situation $[]\hat{}[] = []$. The following two rules are a direct consequence of the previous rule: $S\hat{}[] = []\hat{}S = S$. The last rule is the general case $(\langle x\rangle\hat{}S)\hat{}([y]\hat{}T) = [x]\hat{}(S\hat{}([y]\hat{}T))$ which is based on sequence associativity.

Like sets, our last rules concern sequence decomposition. This is necessary to allow the previous rules to work correctly.

```
aSeq(y, empty),
aSeq(x, S)            :- aSeq(y, aSeq(x, S)).
```

4.4.13 VDM operations

Apart from the VDM Mathematical toolkit, the state part of CML also supports functions and operations. As FORMULA only supports relations, we need to show how to describe functions and operations as relations. This is somewhat straightforward because relations are more general than functions and operations.

Functions We start by considering functions. Let `f` be a VDM function from a generic K-parameterized type `Tp1 * ... * TpK`—corresponding to the function's input—to a resulting type `TR`, corresponding to the function's result. Its definition follows next and it is characterized by the token ==. Thus assuming the K input parameters `p1`, ..., `pK` and a certain function's body definition `body`, we have `f(p1, ..., pK) == body`. Finally we can have a precondition, stating when the function can be applied safely. Let's consider the predicate `Pred` as the precondition of the function `f`. Thus the VDM definition looks like.

```
f: Tp1 * ... * TpK -$>$ TR
f (p1, ..., pK) == body
pre Pred
```

The first thing to observe when transforming the previous VDM definition into FORMULA is that the name of the function has to be defined as a new constructor with the same name where all input parameters as well as the result of the function are declared as fields of the construtor. Thus in FORMULA we have the constructor `f ::= (p1: Tp1, ..., pK: TpK, r: TR).`. The function's body is transformed to a FORMULA expression (`T(body)`) that restricts the possible outputs by an equality operation. But as FORMULA requires expressions to be bound, we need to add bound restrictions for all input parameters of the form `pi is Tpi(_)` (for $i \in 1..K$). Thus we have in FORMULA the rule

```
f(p1, ..., pK, r) :- p1 is Tp1(_), ..., pK is TpK(_), r = T(body).
```

Finally, the precondition is simply appended to the right-hand side of the previous rule, obviously transformed (becoming `T(Pred)`) like the function's body[9]. Therefore, we get in FORMULA the whole definition as.

```
f ::= (p1: Tp1, ..., pK: TpK, r: TR).
f(p1, ..., pK, r) :- p1 is Tp1(_), ..., pK is TpK(_), r = T(body), T(Pred).
```

It is worth observing that, depending on the precondition `Pred`, the above rule can be split in several rules as we pointed out previously when considering the Boolean data type.

[9]Post-conditions are treated like preconditions.

Let's consider a concrete example to illustrate how a VDM function is transformed into a FORMULA construtor with defining rules.

```
id    : nat +> nat
id(n) == n
```

This becomes

```
id   ::= (p1: Nat, r: Nat).
id(n, n) :- n is Nat(_).
```

in FORMULA.

Another example with a precondition.

```
divide     : real * real +> real
divide(x, y) == x / y
pre y <> 0
```

This is transformed into FORMULA as.

```
divide       ::= (p1: Real, p2: Real, r: Real).
divide(x, y, r) :- x is Real(_), y is Real(_), r = x.v / y.v, y.v != 0.
```

Operations Following deliverable D31.2, CML has two ways of defining operations: an implicit (more abstract and basically described in terms of pre and postconditions) and an explicit (more concrete and described in terms of an action possibly requiring some precondition) form. In this deliverable we focus on the implicit formulation.

Operations are captured similarly to functions. The differences come from the fact that operations can change the system state. Thus we consider the initial and after state bindings as parameters of operations described in FORMULA (Indeed CML uses the `frame` keyword to explicitly indicate which state variables can be changed by an operation, following similar ideas of The Refinement Calculus of Morgan [Mor90]).

Let `Op` be a CML operation with input parameters `p1: Tp1, ..., pK: TpK`. Furthermore, consider its precondition given by the predicate `preC` and its postcondition by the predicate `postC`. Thus we have.

```
 Op (p1: Tp1, ..., pK: TpK)
  pre preC
  post postC;
```

To transform the previous definition into FORMULA, we consider the name of the operation, and the current (`st`) and after (`st_`) system state binddings as new parameters (note that both bindings have type `SS` standing for system state) of a generic constructor named `operation`. The rule that defines how the operation

Op may work is only given by the transformation of the postcondition postC with a FORMULA right-hand side rule (That is, a conjunction of facts and propositions). Let's consider that T(postC) is the respective conjunction of facts. As result the operation in FORMULA becomes.

```
operation(``Op'', p1, ..., pK, st, st_, r) :- p1 is Tp1(_), ..., pK is TpK(_),
                                              T(postC).
```

Finally we need to take care of the precondition. From the SOS rules of CML, we need to test whether the precondition holds and otherwise; in this last case the resulting transition yields a Chaos process (see Section 10). Unfortunately due to the stratification restriction we need to have the transformation of the precondition preC in both (positive and negative) forms. That is, we need the FORMULA right-hand sides T(prec) and T(**not** preC). Finally we have the FORMULA constructors and rules.

```
preOper(``Op'', p1: Tp1, ..., pK: TpK, st: SS) :- T(preC).
preNOper(``Op'', p1: Tp1, ..., pK: TpK, st: SS) :- T(not preC).
operation(``Op'', p1, ..., pK, st, st_, r) :- p1 is Tp1(_), ..., pK is TpK(_),
                                              T(postC).
```

5 COMPASS Tool Model Checker Plugin

The COMPASS tool platform was designed as a plugin-based architecture. The model checker functionality is added to the COMPASS IDE using such an architecture. The plugin connects to the COMPASS tool and the core functionality (CML parser and type checker) through the generated CML AST. This connection is defined through AST visitors. Further details are provided in [CML+13].

The model checker plugin (or MCP for short) consists of two main parts: the `core` – containing modules to converting the AST of a CML model into FORMULA (using the AST visitors), to invoke FORMULA as an external application and to build the counterexample, the `ide` – establishing the extension points for Eclipse such as views, perspective, commands, handlers, etc., and the `feature` part, which simply creates a feature to be included in the entire compilation (generating a COMPASS IDE tool with all plugins).

5.1 Architecture

The model checker plugin is a component in the COMPASS core analysis libraries, and is bundled in the
`eu.compassresearch.core.analysis.modelchecker.visitor`
package. The plugin core is based on a collection of classes extending the QuestionAnswerCMLAdaptor. The visitor generates a single FORMULA (with extension `.4ml`) file. To achieve this, it loads the basic embedding (also packaged as a resource in the model checker core part) and complements it by adding a new domain and a partial model corresponding to the processes to be analysed. As the basic embedding also allows extensions (for example, type extensions), the visitor also adds information to the basic content loaded.

The visitor traverses the AST and, for each node, it generates a corresponding FORMULA code. This task involves the use of context objects (CMLModelcheckerContext) to keep information used by other nodes in such a way that dependencies between nodes are resolved using the context object.

There are some utility classes (Utilities and FormulaIntegrationUtilities) that contain useful methods used by the core part of the model checker plugin.

5.2 Model Checker Plugin Behaviour

This section describes the usual flow of behaviour of the model checker plugin.

Plugin initialisation The `MCHandler` class (the event handler of the model checker) captures the AST of the selected unit (a CML file), the property to be checked and instantiates the visitor.

Generate FORMULA script The `generateFormulaScript` method of the `CMLModelcheckerVisitor` class takes a CML AST and the property to be checked. Then it initialises a new context object and calls the `apply` method for the top-level node. This method invokes the `apply` method in the children nodes and generates the corresponding FORMULA code (as a String object). This may also involve putting information on the context to be processed by other nodes.

Invoke FORMULA After receiving the script from the visitor, the handler instantiates a `FormulaIntegrator`, whose method `analyseFile` invokes the FORMULA as an external application and keeps the result (a text containing the base of facts produced FORMULA and other extra information).

Build the counterexample The FORMULA result is given to an instance of a `GraphBuilder` object that builds a graph description (written in DOT language with extension `.gv`) of the counterexample (only if the checked property is valid) and save it to a file. The counterexample construction naturally involves algorithms over graphs such as BFS, DFS, shortest paths and cycle detection.

Graph file generation and visualization The `GraphViz` class receives the path of the generated DOT file and compile it to another file in the Scalable Vector Graphics (SVG) format. After a double click in the MC List View component the plugin opens the generated SVG file using the internal Web browser of Eclispe.

6 Lessons Learned from the Model Checker Implementation

This section contains an overall evaluation of the experiences acquired in the development of the CML model checker.

- **Chosen framework**: When we started this project, we did an evaluation among certain implementation infrastructures to support the development. Several alternatives emerged from basic (object-oriented, functional or logic-based) programming languages, through contract-based languages (such as Perfect Developer [Cro03], which is able to create implementation code from contracts), reuse and extend other model checkers (FDR [For10] or PAT [SLDP09]), SMT solvers (such as Microsoft Research Z3 [DMB08] or Bremen SONOLAR [PVL11]), to abstract frameworks (like Microsoft Research FORMULA [JSD$^+$09]). As we needed to develop the CML model checker conforming to its semantics while the CML language itself (both syntactically and semantically) was being designed, we chose Perfect Developer as our first alternative because it has a minimum desired high-level descriptive powerful infrastructure that seemed to meet our needs. But reasonably soon, we figured out that Perfect Developer would not be the best option. We spent a lot of effort (several months—from November, 2011 to March, 2012) just to create a basic model checker infrastructure (based on [Fre05]) similar to the future CML needs, assuming that the CML semantics would be closer to the Circus language [WCF05] (one of its baseline languages). This effort was huge even considering the helpful support from Escher Technologies to resolve our doubts about Perfect. So this alternative seemed to be too risky particularly because it would take too much time after the right CML syntax and semantics would be available to finish the model checker following such artifacts. Thus we decided to abandon this initiative and try to use FORMULA, whose risk was related to the next lesson learned item (**Framework support**). Although our CML model checker has serious performance problems (see Appendix C), we still think that Microsoft FORMULA was the right framework under the context that it was developed (its one-to-one relation with the semantics was fundamental to build a correct CML model checker within the schedule).

- **Framework support**: Our first direct contact with Microsoft FORMULA occurred in the York University on March, 2012, during a COMPASS convergence meeting. At that time, we thought that FORMULA was like the logic programming language Prolog and thus very easy to learn and use, and full of available literature and users. However, after some initial experiments with FORMULA we realised that it did not behave like Prolog. Our only hint at that time was some powerpoint presentations and conference papers where the author (FORMULA project's leader) said: "Prolog works top-down and FORMULA is bottom-up". As we did not have any kind of support from Microsoft Research, except the presentations and papers, we tried to apply some work available in the literature close to our needs but described in Prolog. We found the work of Leuschel [Leu01]. Our current solution is quite similar to [Leu01] but uses the FORMULA behavioural difference from Prolog. This difference created a serious initial difficulty to create the model checker because we started learning and experimenting with FORMULA on April, 2012 and only on December, 2012 we get a stable version of a CSP model checker. But we were happy with that choice because the time was not spent developing the model checker but mainly on learning how to use FORMULA to build the model checker. This was clear when we extended the CSP model checker to a preliminary Circus model checker in a few working hours. Therefore FORMULA was the right option to create a correct model checker for a formal language that was being developed simultaneously.

- **Orthogonal development**: CML is a language that combines features from the process algebra CSP and the model-based language VDM, with some constructs from the language of Dijkstra [Dij76], in a similar way to Circus. Assuming the orthogonality of these aspects, we decided to create the model checker incrementally from CSP, through VDM until the full CML. This was a very successful decision in the sense that these aspects were really independent of each other (confirming one of the benefits of the Unifying Theories of Programming [HJ98] that was used to create CML). The current version of the model checker links the constituent aspects by pattern-matching, where the CSP constructs guide the activation of the SOS trigger rules. When a VDM syntactic element is found in the body of a process (like an operation call), it creates a CML state just mentioning such a call and containing certain holes to be filled by the interpretation of the VDM part. The syntactic operation call matches with a respective VDM semantic rule that defines the VDM operation itself. Upon activation of such an operation rule, the full CML state becomes available in the labelled transition system. This was also evident when we extended the CSP model checker version to a preliminary Circus version in a few hours.

- **Semantics conformance**: Probably the easiest way to create a CML model checker would be to reuse FDR as we have done in [MS01, FMS04]. However, as the CML semantics has some subtle differences to the CSP semantics (possibly correctly implemented in FDR), we would have to resolve two main difficulties to show that we could create a correct CML model checker by reusing FDR. First, we would need to show which subset of CML could be represented by CSP elements and prove the respective required proof obligations. Such an effort was accomplished in another COMPASS task (but using Circus instead of CML) and reported in this work [OSA+13]. Unfortunately, the model checker for a subset of Circus based on FDR following [OSA+13] also exhibited a poor performance, particularly because of the required CSP encoding to handle the semantics related to external choice and parallelism. This bottleneck is also present in the FORMULA model checker and thus indicates that the performance degradation is not purely associated to the use of the FORMULA technology. Second, and probably the most difficult aspect, CML is intended to support heterogeneous aspects such as time, probability, mobility, etc., that creates a big gap to existing model checkers, prohibiting possible reuses. Therefore we needed to create a model checker that followed the formal SOS semantics of CML independent of combined aspects. Once again FORMULA satisfied such a requirement (For instance, PAT is another model checker for CSP but it does not conform completely to the CSP semantics as FDR does, although in several situations it is faster than FDR due particularly to its on-the-fly model checking algorithm that is not based on a refinement theory [SLS+12]).

- **Building versus searching in a model**: As we presented in the introduction of this deliverable as well as in other sections, most model checkers focus on the search part of the problem, abstracting almost completely the part concerned with building a model from the semantics of a formal language (This discussion is related to the previous item **Semantics conformance**). The FORMULA model checker performs both efforts because we are aiming at correctness about the whole model checking process. Thus it takes a time T_M—for building a model—and a time T_S—for searching for a certain problem in the model built. In our experiments we get that $T_M > T_S$ in general, particularly because the model construction is solely based on the successive application of FORMULA rules that are interpreted against the search procedure that is fully performed by the SMT solver Z3. Obviously if we create the model using another solution (or get a Kripke structure for free), like a programming language (Java, Python, Haskell, etc.), our FORMULA model checker becomes faster. We performed some experiments where we executed FORMULA to build an LTS of a problem as a collection of facts. Then we took this collection of facts as an input to an extremely simpler FORMULA abstraction (basically containing search related queries and nothing about LTS creation) and executed FORMULA again. While FORMULA took minutes to build the LTS, it took seconds to solve the query. But we go back to the original problem of guaranteeing correctness. While a FORMULA abstraction is close to the SOS semantics of a language, a programming code in general is far distant.

- **On-the-fly model checking**: After we have created the CSP, Circus and CML model checkers using a combination between FORMULA rules (to build the LTS) and queries (to search for the desired properties), we tried another possibility: instead of creating the LTS, lets try to find only the counter-example trace if one exists. This alternative is a kind of combinatorial problem: given some open (not initially instantiated) transitions and the set of states containing all fragments of a process's body (the process that is being analysed), use FORMULA to try to fill the transitions using the given states in such a way that it cannot create invalid transitions and finds the counter-example. This is indeed possible and get such an alternative working. To do that we had to calculate the complement of every SOS trigger rule of a formal language (for instance, CSP). This is because FORMULA queries always answer existential questions and SOS triger rules are stated using universal quantifiers. Thus, we used the logical equivalence $\neg\neg\forall x : T \bullet P(x) \equiv \neg\exists x : T \bullet \neg P(x)$ and encoded the problem in this new way: (i) SOS rules are stated in its complementary form (the $\exists x : T \bullet \neg P(x)$ part) as a query `SOSComplRule`, and (ii) the goal becomes the negation of such a query (the ¬(i)) part) or `conforms := !SOSComplRule`.

- **Elaborate data types**: CML is not a simple language in this respect. By inheriting the power of VDM (its Mathematical toolkit), CML supports abstract and elaborate data types from sets to mappings. This is one of the reasons why we decided to opt for Microsoft Research FORMULA instead of using PAT, Microsoft Research Z3 or Bremen SONOLAR. It is well-known from the model checking literature that most model checkers have very restricted data types. An exception to this rule is FDR. FDR provides sets, tuples and enumerated data types, which can easily be used to create a VDM toolkit (as we have done for the Z toolkit [MS01]). By comparing PAT to Z3 or SONOLAR, we agree that one could create a model checker very easily [DSL13] as long as such a model checker does not demand elaborated data types. With respect to data types, PAT is similar to SONOLAR and Z3 would be a better choice. Z3 provides richer data types than the others. Finally, although FORMULA is based on Z3, it has a much more elegant language with recursive data types that allows one to create a VDM Mathematical toolkit as presented in Section 4.4.12. Unfortunately as we have to define all operations related to sets, sequences, and mappings, this creates a huge facts database that worsens the CML model checker performance.

- **FORMULA monotonicity**: One of the most difficult and worst aspects of FORMULA is its facts database monotonicity. With FORMULA, you do not have temporal facts. After creation a fact will persist until the end of the computation. Concerning the CML model checker this creates a problem with respect to two main things: (i) several SOS trigger rules use auxiliary facts that are not used in the final LTS structure but they are necessary to create the facts that will belong to such a structure; (ii) all interpreted operations (for instance, the VDM toolkit) are defined by rules that create facts. Similarly to the auxiliary transitions necessary to build the final LTS, if a set is used then this set is a fact as well as all its subsets must become facts to allow set operations to be available, which by themselves become facts as well. Therefore, another difference between FORMULA and Prolog is that FORMULA does not have backtracking. All intermediate facts are never garbage collected.

- **FORMULA symbolic executor**: As we presented in Section 3, FORMULA uses a combination between a symbolic execution algorithm and the SMT solver Z3. In December, 2012 we thought that Z3 was called during the interpretation of each FORMULA rule. However, during the encoding the mini-mondex problem[10] we realised that Z3 is only called after the symbolic algorithm finishes its job. And this creates a problem when using symbolic data (or an open primitive fact). If the CML process has a recursive call and before such a call, a VDM operation can change the system state, the symbolic algorithm does not stop creating symbolic variables and FORMULA diverges. In such a situation, as we cannot change FORMULA's internal implementation, we have to use a bound to control how many recursive calls a CML process can make. It is really curious because even though the CML process has a finite state space, the use of a symbolic input data creates an infinite symbolic state expansion. By using a bound we find another problem: which bound is appropriate for each problem? This is a similar problem that occurs to several bounded model checkers. An easy solution is to use the abstraction by counter-example approach reported in [CES86].

- **Concrete vs symbolic data**: This topic is related to the previous one. It is very important because although a model checker created by FORMULA exhibits a poor performance in general, it can beat the best model checker when heavy data types are used. In simple comparison tests between FDR

[10] Mini-mondex is a simpler and abstract CML specification version of the Mondex electronic purse specification.

and our CSP model checker created in FORMULA, our model checker found problems in a CSP specification in less time than FDR when FDR had to expand the LTS in a huge structure due to sets of reasonable cardinality used in channel declarations. This is because the time required to interpret a FORMULA specification to build a symbolic LTS and find a suitable instance (by Z3) offset the effort of creating a fully interpreted LTS (as is done by FDR). Finding the right amount of data to exercise a model is an intrinsic model checking problem [CGP99]. The best solutions comes from abstract interpretation [CC92] and SMT solving [BMR12]. As FORMULA is based on SMT solving, we are already using an state-of-the-art solution to the problem. On the other hand, referring to FORMULA symbolic executor, one has to find how to control the symbolic execution algorithm used by FORMULA to avoid symbolic state space explosion.

7 Related Work

Robby [RDH03], Dong [DSL13], and Duret-Lutz [DLP04] provide model checker frameworks whose idea is that we can create any model checker by simply extending the facilities these frameworks offer. From these, the most generic seems to be Bogor [RDH03] because it gives the power of a functional language to define new data types. However, none of them have facilities to guarantee that the SOS semantics of a given language is correctly implemented by the new model checker extension. Kázmierczak et al [KPg12] shows how to create a CTL model checker for Normative systems using Haskell. He uses Kripke structures and concentrates his implementation in an adaptation of traditional model checking algorithms (search only). The Kripke structure (the model of a given problem) is given directly without SOS rules. Data structures are trivial (integers). Still concerning the reuse of language and tools but concentrating on the model based language, we have the work reported in [DNS11] where the authors show how to encode a subset of the Z language into the SAL toolset (it includes a model checker and a simulator). In several points such an encoding is similar to ours as described in Section 4.4.12. The main difference is that in Z2SAL the authors focus only in Z instead of an integration between Z and a behavioural language. The resulting model checker seems to be fast but it must be observed that the checks are related to discharging proof obligations instead of the analysis of a full LTS.

Banda [BG10] shows how to apply completely standard techniques for constructing abstract interpretations of a CTL semantic function, without restricting the kind of properties that can be verified. Furthermore the author shows that this leads directly to implementation of abstract model checking algorithms for abstract domains based on constraints, making use of an SMT solver. Her work is done in Prolog.

Palikereva [POR12] proposes a prototype called SymFDR, which implements a bounded model checker for CSP based on SAT-solvers. The authors compare with the FDR tool to show that SymFDR can deal with problems beyond FDR, such as complex combinatorial problems. Moreover, they found that FDR outperforms SymFDR when a counter-example does not exist. In our work we extend the capability of SymFDR by using SMT-solving and not depending on FDR to create the LTS. In this way we can handle infinite state systems while SymFDR can only deal with systems that FDR can, that is, finite state systems.

Leuschel [Leu01] proposes an implementation of the CSP language based on SICStus Prolog (a variation of Prolog). His main goal is to provide a CSP interpreter and animator. According to Leuschel's work, with a little effort his solution could be combined with a CTL model checker (e.g. SPIN) and also provide verification of CTL properties. Part of the design of our model checker in FORMULA follows a similar declarative and logic representation as reported in [Leu01], but the main idea is to be able to reason about concurrent systems using a rich specification language like CSP. As our model checker can handle infinite state systems, we indeed concretise the future work of [Leu01] towards this subject.

Meseguer [BM12] works with Maude but this time showing differences between dealing with Kripke structures (state info is relevant) and LTS (behaviour is relevant). The paper presents the need to use a formalism to specify state and another to specify properties, and analyses the consequence in "cooking" both the system and the property in both state-based and action-based tandems as a lack of expressiveness in both cases. It then considers a semantics extension of a CTL model checker written in Maude to a TLR (Temporal Logic of Rewriting) model checker.

The idea of using an SMT-solver for model checking purpose is not new either. The advances of SMT solvers bring a new level of verification. Bjorner [BMR12] extend the SMT-LIB to describe rules and declare recursive predicates, which can be used by a symbolic model checking. The idea of property verification is similar to the reachability analysis. That is, the property verification can be rewrite as reachability questions [BMR12]. Alberti [ABG$^+$12] proposes an SMT-based specification language to improve the verification of safety properties. We are interested in providing an efficient model checking for the CSP specification language. Ghilardi [GR10] proposes a SMT model checker to check safety properties of infinite-state systems. Its capability of dealing with infiniteness is inherited from an SMT solver like in our case. This paper [GR10] shows the performance of the model checker for simple examples like ours and the result is quite similar. In our work we bring a new perspective for reasoning about infinite systems by using a high level specification language. Our work differs from them by using an SMT-solver to increase the expressiveness of the process algebra CSP to provide a powerful tool for verification and reasoning of programs.

8 Conclusions

In this deliverable we have shown how to build a semantics preserving model checker for a rich-state language in an iterative way, starting from a language as CSP and then dealing with the complexity of CML, where rich-state is described in VDM.

The main reason to work this way was that CML is a formal language based on (a combination of) other mature and formal languages such as VDM [ABH$^+$95], CSP [Ros10], and Circus [WC02], whose syntax (reported in deliverable D23.1) and semantics (reported in deliverable D23.3) were being developed concurrently to the model checker. Furthermore, it is also expected that CML will evolve in the future to accomodate still more complex aspects such as mobility, probability, etc.

Apart from developing the model checker gradually, we also had to use an implementation framework that was trustworthy as well as easy to keep in pace with the evolution of the CML syntax and semantics. After some investigation related to possible alternatives (see Section 6), we chose Microsoft FORMULA as the best alternative to follow the CML semantics as close as possible while delivering a model checker of reasonable performance. In Sections 3 and 4 we present the details of the construction of the CML model checker in terms of FORMULA syntax, but in the Appendices A and B we also provide more formal material towards why FORMULA is a good candidate as implementation infrastructure to build a trustworthy CML model checker.

Our feasibility study has shown that although the CML model checker works reasonably well for creating a prototype tool, a number of improvements can be done to evolve such a model checker to a competitive scenario. We list some of them in what follows as possible future work.

- We have used FORMULA for two main things:

  1. Create the labelled transition system of a CML specification: This requires FORMULA to process several rules corresponding to Structure Operational Semantics trigger rules;

  2. Search the FORMULA knowledge base to ensure the satisfaction of desired properties: FORMULA knowledge base is a database or set of logical facts. Facts can be given as input (primitive facts) or generated by processing rules.

  Step 1 takes time to execute while Step 2 is considerably fast. One solution to improve the performance of the CML model checker is to create the LTS using another implementation medium, such as a functional programming language, like Haskell, or an object-oriented or mixed language, like Java or Python (This is more an engineer's problem);

- Still related to Step 1, one can try to simply rewrite the FORMULA rules in a more optimised way, following the correspondence between the FORMULA semantics and that of the Datalog language. As Datalog is more mature than FORMULA, the literature has some material related to Datalog rules optimisation [CGT89] (This is more an engineer's problem as well);

- A third option, aiming at optimising the CML model checker, is to see the FORMULA framework as a prototype generation medium that serves to create a correct by construction tool (a kind of implementable specification), whose optimal implementation is derived from it. Again from the literature of Datalog, we can use work in the literature towards a derivation of an imperative implementation code from a FORMULA abstraction (rules and queries). Although this can be classified as an engineer's effort, this also needs some research effort as well. It seems that a good candidate to follow this direction is to use the integration between Python and Z3, named Z3Py [dM13];

- As several SOS rules, particularly those related to external choice and parallelism, need to anticipate facts (what we call in Sections 3 and 4 as auxiliary facts) and the FORMULA knowledge base is monotonic (that is, once a fact is created it cannot be removed from the knowledge base), this creates a huge and heavy knowledge base to deal with. As future work one can avoid expanding such rules and acting on demand, following a similar solution that is implemented in the model checker FDR [For10];

- Although we provide the material in Appendices A and B, linking FORMULA code to First-Order Logic, the ideal situation is to create a refinement calculus for FORMULA in such a way that one can derive, following a stepwise refinement approach, a correct FORMULA abstraction from a formal description of a problem. This is indeed a hot topic for future research, particularly whether one can provide the semantics of FORMULA as well as a refinement calculus using The Unifying Theories of Programming [HJ98];

- The current CML model checker works similarly to several model checkers in the sense that it cannot cope with some infinite-state systems. It can handle some infinite-state systems, where the source of infinity comes from channel data, but it cannot reason about systems that have infinite internal states. As future work one can create a FORMULA abstraction suitable for inductive proofs;

A FORMULA Semantics

Microsoft FORMULA is a combination between Constraint Logic Programming (CLP) and Satisfiability Modulo Theories (SMT) [JSD+09]. Executing a FORMULA abstraction means determining whether a logic program can be extended by a finite set of (primitive) facts so that a goal is satisfied. This requires searching through (infinitely) many possible extensions using the state-of-the-art SMT solver Z3 [DMB08]. Consequently, FORMULA abstractions can include variables ranging over infinite domains and rich data types. Nonetheless, the method is constructive. That is, the algorithm behind FORMULA returns extensions of the program witnessing goal satisfaction.

First, let's introduce the concept of an interpretation. Let U be a (possibly infinite) set called a universe. Let r be an n-ary relation symbol and r^I a (finite) interpretation of r; r^I is a (finite) subset of U^n. As shorthand, we use $r(\bar{t})$ meaning r applied to elements $t_1, \ldots, t_n$ of U.

Definition 1 (interpretation) *An interpretation is a triple $I = \langle D, \phi, \pi \rangle$, where*

- *D is the domain (a nonempty set). Elements of D are individuals,*
- *ϕ is a mapping that assigns to each constant an element of D. Constant c denotes individual $\phi(c)$,*
- *π is a mapping that assigns to each n-ary predicate symbol a relation: a function from D^n into booleans ($\{true,\ false\}$).*

followed by what means a truth in some interpretation.

Definition 2 (truth in an interpretation)

- *A constant c denotes in I the individual $\phi(c)$.*
- *Ground (variable-free) atom $p(t1, ..., tn)$ is*
  - *$true$ in interpretation I if $\pi(p)(t'_1, \ldots, t'_n)$, where t_i denotes t'_i in interpretation I and*
  - *$false$ in interpretation I if $\neg\pi(p)(t'_1, \ldots, t'_n)$.*
- *Ground clause $h \leftarrow b_1 \wedge \ldots \wedge b_m$ is*
  - *$false$ in interpretation I if h is $false$ in I and each b_i is $true$ in I, and*
  - *$true$ in interpretation I, otherwise.*
- *A knowledge base, KB (or a least Herbrand universe $lm(\Pi)$, for a program Π), is true in interpretation I if and only if every clause in KB is true in I*

And variable assignment means the following.

Definition 3 (variable assignment) *A variable assignment is a function from variables into the domain.*

FORMULA has the concept of a model.

Definition 4 (model) *A model of a set of clauses is an interpretation in which all the clauses are true.*

and logical consequence as in the following definition.

Definition 5 (logical consequence) *If KB is a set of clauses and g is a conjunction of atoms, g is a logical consequence of KB, written $KB \models g$ (or $lm(\Pi^*) \models \tilde{\exists} g$), if g is $true$ in every model of KB.*

Finally, we have the concept of CLP satisfiability.

Definition 6 (CLP Satisfiability). *Given:*

- *A program Π with relation symbols $R = \{r_1, \dots, r_n\}$,*
- *$R_p \subseteq R$ a subset of the program relations, called the primitive relations.*
- *A quantifer-free goal g over the program relations.*

Then find a finite interpretation R_p^I for primitive relations such that:

$$lm((\Pi \cup R_p^I)^*) \models \tilde{\exists} g$$

The program $\Pi \cup R_p^I$ is obtained by extending Π with a fact $r(\vec{t})$ whenever $R_p^I \models r(\vec{t})$.

The program can only be extended by primitive relations R_P. The contents of R_P^I are the facts that, when added to the program, cause the goal to be satisfied.

FORMULA rules have a direct correspondence with First-Order Logic formulas. For instance,

```
q(X,Y) :- p(X,Y).
q(X,Z) :- q(X,Y), q(Y,Z).
```

is equivalent to

$$\forall X, Y \bullet (p(X,Y) \implies q(X,Y)) \wedge$$
$$\forall X, Z \bullet \exists Y \bullet (q(X,Y) \wedge q(Y,Z) \implies q(X,Z))$$

To avoid repetition of the right-hand side of a rule, one can write a comma between heads. For example.

```
q(X) , r(X) :- p(X).
```

is equivalent to $\forall X \bullet p(X) \implies (q(X) \vee r(X))$. When the head is the same for different bodies, one can use semicolon as in the following example.

```
q(X) :- r(X); p(X).
```

is equivalent to $\forall X \bullet (r(X) \vee p(X)) \implies q(X)$.

FORMULA queries, unlike rules, are existentially quantified. Thus, for example

```
query1 := q(X,2), p(X,Y).
```

is equivalent to

$$\exists X, Y \bullet q(X, 2) \wedge p(X, Y)$$

and

```
query2 := q(X,_), fail p(X,Y).
```

is equivalent to

$$\exists X, Y, Z \bullet q(X, Z) \wedge \neg p(X, Y)$$

B Properties in FORMULA

In this section we show how the properties encoded in FORMULA were derived following the correspondence between first-order logic formulas and constraint-logic programs given by Clark completion, which is inherited in FORMULA [JSD+09].

B.1 Deadlock analysis

Formally, a deadlock occurs whenever the formula

$\exists s : \mathcal{T}(P) \bullet ref(P/s) = \Sigma \cup \{\surd\}$

holds. That is, if a process P evolves through a trace s and after that it cannot engage (it refuses) in any visible event, including $\surd$.

To find the equivalent FORMULA rules and queries that answer the above first-order logic formula, let us first rewrite that formula in some of its equivalent (simpler) logical formulas to become closer to a FORMULA corresponding logical solution.

$\exists s : \mathcal{T}(P) \bullet ref(P/s) = \Sigma \cup \{\surd\}$ (by Definition)

$\equiv \exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\tau, \surd\} \bullet e \in ref(P/s) \iff e \in \Sigma \cup \{\surd\}$(=-Def)

$\equiv \exists s : \mathcal{T}(P) \bullet \forall e : \{\tau\} \cup (\Sigma \cup \{\surd\}) \bullet e \in ref(P/s) \iff e \in \Sigma \cup \{\surd\}$ (Set Th)

$\equiv \exists s : \mathcal{T}(P) \bullet (\forall e : \{\tau\} \bullet e \in ref(P/s) \iff e \in \Sigma \cup \{\surd\}) \wedge (\forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s) \iff e \in \Sigma \cup \{\surd\})$ (Range split)

$\equiv \exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s) \iff e \in \Sigma \cup \{\surd\}$(By conj)

$\equiv \exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s)$ (By FOL)

To find the equivalent FORMULA query to the previous first-order logic formula we have to introduce some definitions. Ideally we would like to define a fact concerning the after (/) operator as follows. Let $\langle e_1, \ldots, e_k \rangle$ be a trace of P (that is, $\langle e_1, \ldots, e_k \rangle \in traces(P)$), such that P is a process equation. Then $P/\langle e_1, \ldots, e_k \rangle = \mathtt{S}_k$, given by the following hypothetical fact.

`Pafter(`$\langle e_1, \ldots, e_k \rangle$`, S`$_k$`)`

available in the FORMULA knowledge base as long as the following right-hand side of its rule

`trans(State(ProcDef(`P`,_,`P_{body}`)),tau,State(`P_{body}`)),`

`    trans(State(`P_{body}`), e`$_1$`, S`$_1$`), ..., trans(S`$_{k-1}$`, e`$_k$`, S`$_k$`).`

holds.

As FORMULA cannot have a variable-size rule body, we have to obtain it by transitivity (creating possibly several intermediary facts). Thus we have to define the previous general non-realisable rule by one or more new realisable rules in FORMULA.

First, it is worth noting that the trace s is not special. Any trace ($\exists\ s$) is acceptable. So let us focus our solution on a reachability analysis viewpoint. That is, let us create the state P/s for any s.

Definition 7 *Let s be a trace of P, such that P is the name of a process (that is, $P(pPar) = P_{body}$). If the fact `reachable(`Q`)`, given by the following rule.*

`reachable(Q) :-`

`GivenProc(`P`), ProcDef(`P`, `$pPar$`, `P_{body}`),`
`trans(State(`P_{body}`),_,Q);`

`reachable(R), trans(R,_,Q).`

becomes available in the FORMULA knowledge base, then `Q=` $\exists s : \mathcal{T}(P) \bullet P/s$.

Note that with Definition 7 we are computing P/s in FORMULA without recording the specific events of s.

Refusals are defined following their logical formulation reported in [RBH84] as.

$$ref(P) = \{X \mid X finite \wedge \exists Q.P \xrightarrow{\tau^*} Q \wedge X \cap initials(Q) = \emptyset\}$$

where the notation τ^* means zero or more internal events can occur (and τ^+ means that at least one internal event occurs).

Definition 8 *Let P and Q be states of an LTS. If $P \xrightarrow{\tau^+} Q$ then the* `tauPath(P, Q)` *is present in the FORMULA knowledge base, where* `tauPath` *is given by the following rules.*

```
tauPath(P,Q) :- trans(P,tau,Q).
tauPath(P,Q) :- tauPath(P,S),tauPath(S,Q).
```

Now we can define by what means an event e be refuted at state P (formally, $e \in ref(P)$).

Definition 9 *Let e be a visible event. Then*

$e \in ref(P) \triangleq$ ***fail*** *trans(P, e, _), e != tau; tauPath(P, Q),* ***fail*** *trans(Q, e, _), e != tau.*

proviso. P is a process expression.

To obtain the corresponding FORMULA script related to the formula $\exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s)$, we have to generalise the previous definition to any event. This is easy in FORMULA by using the "don't care" (_) operator, as shown in the following lemma.

Lemma 1 $\forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P) \equiv$***fail*** trans(State($P$), _, _).

Proof.

1. $\forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P)$
2. $e_1 \in ref(P) \wedge \ldots \wedge e_k \in ref(P)$ *($\forall$-ext)*

3. ***fail*** *trans(P, e_1, _), e_1 != tau; tauPath(P, Q),* ***fail*** *trans(Q, e_1, _), e_1 != tau, ...,* ***fail*** *trans(P, e_k, _), e_k != tau; tauPath(P, Q),* ***fail*** *trans(Q, e_k, _), e_k != tau.* *(By Def. 9)*

4. ***fail*** *trans(P, _, _); tauPath(P, Q),* ***fail*** *trans(Q, _, _).* *(_ - Def.)*

Now we have to show what happens to a refusal check when the location (current state of the labelled transition system) changes.

Theorem 1 *Let s be a trace of P. If $\exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s)$ then* `reachable(Q), fail trans(Q, _, _); reachable(Q), tauPath(Q, R), fail trans(R, _, _).`

Proof.

1. $\exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s)$ *(By hyp.)*

2. $Q = \exists s : \mathcal{T}(P) \bullet P/s \land \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(Q)$ *(By Pred. Calc.)*

3. `reachable(Q), fail trans(Q, _, _); reachable(Q), tauPath(Q, R), fail trans(R, _, _).` *(By Def. 7 and Lemma 1)*

Theorem 4 represents the corresponding FORMULA encoding (a query) of $\exists s : \mathcal{T}(P) \bullet \forall e : \Sigma \cup \{\surd\} \bullet e \in ref(P/s)$. That is, a deadlock was found by following some trace s. This encoding is enough for CML because CML does not use the special event $\surd$ for representing SKIP. However, for CSP we have to consider an extra clause because this encoding considers a CSP process ending with SKIP as a deadlock as well and this is not conceptually correct although the CSP model checker FDR works this way. Therefore, to check deadlock in FORMULA for CSP we have to add the condition $last(s) \neq \surd$. This is easily represented in FORMULA as `trans(_, tick, L)`. Therefore for CSP, the final query is

```
reachable(Q),  fail trans(_, tick, Q),
fail trans(Q, _, _); reachable(Q),
tauPath(Q, R),  fail trans(R, _, _).
```

and for CML it is

```
reachable(Q),  fail trans(Q, _, _); reachable(Q),
tauPath(Q, R),  fail trans(R, _, _).
```

B.2 Livelock analysis

Livelock analysis is similar to deadlock analysis in the sense of finding some initial trace, from which something happens. In the case of livelock, this means finding a loop of infinite invisible events.

In logical terms, livelock is characterised as

$\exists s : \mathcal{T}(P); t : \mathcal{T}(P/s) \mid \mathbf{ran}\, t = \{\tau\} \bullet P/(s \frown t) = P/s$

Similarly to deadlock, let us first rearrange the previous logical formula in a more independent (orthogonal) description.

$\exists s : \mathcal{T}(P); t : \mathcal{T}(P/s) \mid \mathbf{ran}\, t = \{\tau\} \bullet P/(s \frown t) = P/s$

$\equiv \exists s : \mathcal{T}(P); t : \mathcal{T}(P/s) \mid \mathbf{ran}\, t = \{\tau\} \bullet (P/s)/t = P/s$ (/ - Def.)

$\equiv \exists s : \mathcal{T}(P); Q; t : \mathcal{T}(Q) \mid Q = P/s \wedge \mathbf{ran}\, t = \{\tau\} \bullet Q/t = Q$ ($\exists$ - Def.)

$\equiv (Q = \exists s : \mathcal{T}(P) \bullet P/s) \wedge (\exists t : \mathcal{T}(Q) \mid \mathbf{ran}\, t = \{\tau\} \bullet Q/t = Q)$ (By FOL)

From the previous formula, we already have the first part. That is, from Definition 7 we know that $Q = \exists s : \mathcal{T}(P) \bullet P/s$ corresponds to

`reachable(Q)`

The other formula ($\exists t : \mathcal{T}(Q) \mid \mathbf{ran}\, t = \{\tau\} \bullet Q/t = Q$) is similar to reachability in terms of transitivity and it was already introduced. We have only to find the fact `tauPath(Q, Q)` in the FORMULA knowledge base to conclude that process Q has a infinite loop of invisible actions.

Thus livelock analysis is simply the conjunction of the previous FORMULA encodings, or

Theorem 2 *If* $(Q = \exists s : \mathcal{T}(P) \bullet P/s) \wedge (\exists t : \mathcal{T}(Q) \mid \mathbf{ran}\, t = \{\tau\} \bullet Q/t = Q)$ *then* `reachable(Q), tauPath(Q,Q).`

Proof.

1. $(Q = \exists s : \mathcal{T}(P) \bullet P/s) \wedge (\exists t : \mathcal{T}(Q) \mid \mathbf{ran}\, t = \{\tau\} \bullet Q/t = Q)$ *(By hyp.)*
2. `reachable(Q), tauPath(Q, Q).` *(By Defs. 7 and 8)*

B.3 Nondeterminism analysis

Roscoe [Ros10] defines determinism for a process P as

$s \frown \langle a \rangle \in \mathcal{T}(P) \implies (s, \{a\}) \notin \mathcal{F}(P).$

In order to find a counter-example, we have to negate the previous definition. Thus we get

$s \frown \langle a \rangle \in \mathcal{T}(P) \wedge (s, \{a\}) \in \mathcal{F}(P).$

By a simple rewrite we obtain.

$a \in initials(P/s) \wedge a \in ref(P/s).$

The term $a \in initials(P)$ is trivially defined in FORMULA as follows.

Definition 10 *Let P be a state of an LTS. If $e \in initials(P)$ then the fact* `trans(P, e, \_)` *is present in the FORMULA knowledge base.*

proviso *P is a process expression.*

Finally we can obtain nondeterminism in FORMULA as a result of the follwing theorem.

Theorem 3 *Let P be a CML process. If $a \in initials(P/s) \wedge a \in ref(P/s)$ then the query*

```
reachable(Q), trans(Q, a, _),
tauPath(Q, R), fail trans(R, a, _)
```

holds in the FORMULA knowledge base.

Proof.

1. $a \in initials(P/s) \wedge a \in ref(P/s)$ *(By hyp.)*
2. `reachable(Q), trans(Q, a, _),`
   `tauPath(Q, R), fail trans(R, a, _)` *(By Defs. 10 and 9)*

B.4 Traces refinement

Our last property of interest in this deliverable is traces refinement. As said previously, in FORMULA we indeed look for a violation of such a property. Thus in this section we show how to detect a counter-example in a traces refinement following its mathematical definition.

The traces of a process (already in LTS form) are given by

$$\mathcal{T}(P) = \{s \mid P \xrightarrow{s} Q\}$$

Traces refinement ($\sqsubseteq_{\mathcal{T}}$) is defined as follows.

$$P \ \sqsubseteq_{\mathcal{T}} \ Q \equiv \mathcal{T}(Q) \subseteq \mathcal{T}(P)$$

which means (by FOL) that

$$\forall s \bullet s \in \mathcal{T}(Q) \implies s \in \mathcal{T}(P)$$

As before, we have to negate the previous formula to get a counter-example (if one exists). Hence

$$\begin{gathered} \neg \forall s \bullet s \in \mathcal{T}(Q) \implies s \in \mathcal{T}(P) \\ \equiv \exists s \bullet s \in \mathcal{T}(Q) \land s \notin \mathcal{T}(P) \end{gathered}$$

As the traces semantics is prefix closed and $\langle\rangle \in \mathcal{T}(P)$ for any process P, we can work with the above formula by a case analysis (induction).

Suppose $s = \langle e \rangle$. Thus the formula

$$\langle e \rangle \in \mathcal{T}(Q) \land \langle e \rangle \notin \mathcal{T}(P)$$

can be rewritten as

$$e \in initials(Q) \land e \notin initials(P)$$

The other case is similar to this one, but more general. Consider now that $s = t \frown \langle e \rangle$. Thus

$$t \frown \langle e \rangle \in \mathcal{T}(Q) \land t \frown \langle e \rangle \notin \mathcal{T}(P)$$

can be rewritten to

$$\langle e \rangle \in \mathcal{T}(Q/t) \land \langle e \rangle \notin \mathcal{T}(P/t)$$

that is equivalent to

$$e \in initials(Q/t) \land e \notin initials(P/t)$$

As result, we just have to find an after state for both processes P and Q for a prefixed trace t (which can be empty—the base case) and check the possibility and impossibility of a same event occurring in these processes.

$$\exists\, t \bullet e \in initials(Q/t) \wedge e \notin initials(P/t)$$

As FORMULA cannot handle traces of variable-size we had to capture the above logical formula by walking both LTSs simultaneously. With respect to the previous formula to be computable in FORMULA we need this rewritten.

$$\begin{aligned}
&\exists\, t \mid t = \langle e_0, \ldots, e_k \rangle \bullet e_0 \in initials(Q) \wedge e_0 \in initials(P) \wedge \\
&\qquad\qquad\qquad\qquad \ddots \\
&\qquad\quad e_k \in initials(Q/\langle e_0, \ldots, e_k \rangle) \wedge e_k \in initials(P/\langle e_0, \ldots, e_k \rangle) \wedge \\
&\qquad\qquad\quad e \in initials(Q/t) \wedge e \notin initials(P/t)
\end{aligned}$$

We consider a relation `C_Ex` from states (of the specification and implementation) to states (of the specification and implementation) via an event from the implementation, given by a case analysis (or step-law).

Theorem 4 *Let P and Q be CML processes. If $\exists\, t \bullet e \in initials(Q/t) \wedge e \notin initials(P/t)$ then the fact `C_Ex(_, _, _,` Ω`,` Ω`)` holds in the FORMULA knowledge base.*

Proof.

For the very first transition (process definitions) we have.

$C\_Ex(P_0,\ Q_0,\ tau,\ P_{body},\ Q_{body})$:-
    `Spec.GivenProc(P), Impl.GivenProc(Q),`
    `ProcDef(P,pP,PBody),ProcDef(Q,pQ,PBody).`

where

- P_0 =`Spec.State(proc(P,pP)),`
- Q_0 =`Impl.State(proc(Q,pQ)).`
- P_{body} =`Spec.State(PBody),`
- Q_{body} =`Impl.State(PBody).`

It is worth pointing out that this fact will be present in the FORMULA knowledge base even if a counter-example cannot be found because, in logical terms, the fact $C\_Ex(P_0,\ Q_0,\ tau,\ P_{body},\ Q_{body})$ holds merely by the presence of the intent to check a trace refinement. We need it just to create a first case that satisfies the general rules to capture a traces refinement violation. Finally the prefixes `Spec.` *and* `Impl.` *are needed by FORMULA due to a design decision (reuse of the domains related to syntax and semantics). In what follows we present it in a more mathematical fashion for easy reading.*

As $\tau \notin initials(P)$ for any process P, the following rule jumps to another possible visible transition of the implementation LTS.

$$\texttt{C\_Ex}(P,\ Q,\ ev,\ P',\ Q'')\texttt{:-}$$
$$Q' \overset{\tau}{\to} Q'',\ \texttt{C\_Ex}(P,\ Q,\ ev,\ P',\ Q').$$

To capture the traces prefix-closedness property in FORMULA we use the following rule.

$$\texttt{C\_Ex}(P,\ Q,\ ev,\ P',\ Q')\ \texttt{:-}$$
$$\texttt{C\_Ex}(\_,\ \_,\ \_,\ P,\ Q),\ Q \overset{ev}{\to} Q',\ P \overset{ev}{\to^{\star}} P',\ ev\ \texttt{!=}\ \texttt{tau}.$$

where $P \overset{ev}{\to^{\star}} P'$ means that we can have several invisible actions before ev can occur. This special symbol is indeed equivalent to Definition 7 (Reachability), although we had to write a new rule in FORMULA to deal specifically with the specification process of a refinement relation. This rule is given by.

```
CEPath(P, ev, Q)  :-
    Spec.trans(P, ev, Q), ev!=tau ;
    Spec.tauPath(P, S), Spec.trans(S, ev, Q), ev!=tau.
```

Finally the logical formula

$$\exists\, t \bullet e \in initials(Q/t) \wedge e \notin initials(P/t)$$

is equivalent in FORMULA to the presence of the fact

$$\texttt{C\_Ex}(\_,\ \_,\ \_,\ \Omega,\ \Omega).$$

in the FORMULA knowledge base, which is only possible whether the follwing rule can be fired.

$$\texttt{C\_Ex}(P,\ Q,\ ev,\ \Omega,\ \Omega)\ \texttt{:-}$$
$$\texttt{C\_Ex}(\_,\ \_,\ \_,\ P,\ Q),\ Q \overset{ev}{\to} Q',\ ev\ \texttt{!=}\ \texttt{tau},\ P \overset{ev}{\not\to^{\star}} P'.$$

C Key Examples

This section contains key examples to emphasize strong and weak points of FORMULA and FDR. We have observed that although FORMULA is able to deal with infinite data types via SMT solving, its performance degrades with some issues:

- Size of the knowledge base: The more facts are in the knowledge base the more time the analysis takes to finish. The analysed examples show that this relation is exponential for some operators.
- Uninstantiated data: The more uninstantiated data is used, the more expensive is the analysis. This is because FORMULA calls Z3 to instantiate these data.
- Low coupling between rules: The generation of facts can be related in some way. The more precise is the specification of these relations, the more faster is the analysis. The language of FORMULA allows rules to be defined through constraints that can be sufficient (in the sense that they enable one rule but also enable others) or optimal (in the sense that they enable more than one rule).

On the other hand, the capability of instantiating values that satisfy the constraints of a model is a strong feature of FORMULA that makes it more useful than FDR[11]. We show these differences through two simple examples.

Replicated constructs

Replicated constructs are a common source of degrading performance as they might combine executions in different ways (synchronous, asynchronous, etc.). For example, the following process replicates an action that gets a value `x` (a parameter), communicates it on channel `choose` and terminates successfully.

```
channels
choose : int

process P =
begin
actions
TEST = val x : int @ choose.x -> Skip

@ [] i in set {1,2} @ TEST
end
```

We consider the indexing variable `i` varying through the sets {`1,2`}, {`1,2,3`}, {`1,2,3,4`} and {`1,2,3,4,5`}. The result is illustrated in Table 9.

[11]Under circumstances where automatic data abstraction is not available

Tool	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3,4\}$	$\{1,2,3,4,5\}$
FORMULA	3s and 44 facts	5s and 156 facts	21s and 556 facts	183s and 1946 facts
FDR	0.011s	0.012s	0.012s	0.013s

Table 9: FORMULA and FDR in replicated constructs

Infinite Types Involved in Communications and Predicates

Although the performance of FDR is superior to that of FORMULA, FDR cannot analyse specifications containing infinite data types in communications and in predicates. This is because FDR generates the set of events prior to the LTS construction. The following example shows a system that cannot be analysed by FDR whereas our model checker is able to handle it.

```
channels
choose : int

process P =
begin
@ choose?x -> choose?y -> [x = y] @ Skip
end
```

The events `choose?x` and `choose?y` are infinite as there are no constraint on the values of `x` and `y`. This is a typical situation not handled by FDR. However, our model checker is able to instantiate values suitable to falsify the guard and thus originate a deadlock.

References

[ABG$^+$12] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Reachability modulo theory library. In *SMT Workshop 2012 10th International Workshop on Satisfiability Modulo Theories SMT-COMP 2012*, page 66, 2012.

[ABH$^+$95] D.J. Andrews, H. Bruun, B.S. Hansen, P.G. Larsen, N. Plat, et al. *Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1: Base language*. ISO, 1995. Draft International Standard: 13817–1.

[AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[BCC$^+$13] Jeremy Bryans, Samuel Canham, Ana Cavalcanti, Andy Galloway, Thiago Santos, Augusto Sampaio, and Jim Woodcock. CML Definition 2. Technical report, COMPASS Deliverable, D23.3, March 2013. Available at http://www.compass-research.eu/.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[BG10] G. Banda and J. P. Gallagher. Constraint-based abstraction of a model checker for infinite state systems. In U. Geske and A. Wolf, editors, *Proceedings of the 23rd Workshop on (Constraint) Logic Programming (WLP 2009)*, pages 109–124, Potsdam, Germany, 2010. Potsdam Universitätsverlag.

[BM12] Kyungmin Bae and José Meseguer. A rewriting-based model checker for the linear temporal logic of rewriting. *Electron. Notes Theor. Comput. Sci.*, 290:19–36, December 2012.

[BMR12] Nikolaj Bjørner, Kenneth L McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT Workshop at IJCAR*, 2012.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.

[CHM00] Andrea Corradini, Reiko Heckel, and Ugo Montanari. Graphical operational semantics. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques*, 2000.

[CML+13] Joey W. Coleman, Anders Kaels Malmos, Rasmus Lauritsen, Luís D. Couto, and Richard Payne. Second release of the COMPASS tool — developer documentation. Technical report, COMPASS Deliverable, D31.2b, January 2013.

[CMLC13] Joey W. Coleman, Anders Kaels Malmos, Rasmus Lauritsen, and Luís D. Couto. Second release of the COMPASS tool — user manual. Technical report, COMPASS Deliverable, D31.2a, January 2013.

[Cro03] David Crocker. Perfect developer: A tool for object-oriented formal specification and refinement. tools exhibition notes at formal methods europe. In *In Tools Exhibition Notes at Formal Methods Europe*, page 2003, 2003.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ., 1976.

[DLP04] Alexandre Duret-Lutz and Denis Poitrenaud. Spot: An extensible model checking library using transition-based generalized büchi automata. In *MASCOTS*, pages 76–83. IEEE Computer Society, 2004.

[dM13] Leonardo de Moura. The z3 api for python. `http://rise4fun.com/z3py`, 2013. Accessed Sep 09 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and algorithms for the construction and analysis of systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DNS11] John Derrick, Siobhn North, and Anthony J. H. Simons. Z2sal: a translation-based model checker for z. *Formal Asp. Comput.*, 23(1):43–71, 2011.

[DSL13] Jin Song Dong, Jun Sun, and Yang Liu. Build your own model checker in one month. In *ICSE*, pages 1481–1483, 2013.

[FMS04] Adalberto Farias, Alexandre Mota, and Augusto Sampaio. Efficient csp-z data abstraction. In EerkeA. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 108–127. Springer Berlin Heidelberg, 2004.

[For10] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement. FDR2 User Manual.*, 2010.

[Fre05] Leonardo Freitas. *Model Checking Circus*. PhD thesis, University of York, 2005.

[GM99] Maria-del-Mar Gallardo and Pedro Merino. A framework for automatic construction of abstract promela models. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 184–199. Springer Berlin Heidelberg, 1999.

[GR10] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In Jrgen Giesl and Reiner Hhnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, 2010.

[HJ98] Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.

[JSD+09] Ethan K. Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjrner, and Wolfram Schulte. Specifying and composing non-functional requirements in model-based development. In Alexandre Bergel and Johan Fabry, editors, *Software Composition*, volume 5634 of *Lecture Notes in Computer Science*, pages 72–89. Springer Berlin Heidelberg, 2009.

[KPg12] Piotr Kazmierczak, Truls Pedersen, and Thomas gotnes. Normc: a norm compliance temporal logic model checker. In Kristian Kersting and Marc Toussaint, editors, *STAIRS*, volume 241 of *Frontiers in*

Artificial Intelligence and Applications, pages 168–179. IOS Press, 2012.

[Leu01] Michael Leuschel. Design and implementation of the high-level specification language csp(lp) in prolog. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 14–28. Springer Berlin Heidelberg, 2001.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[MS01] A. Mota and A. Sampaio. Model-Checking CSP-Z: Strategy, Tool Support and Industrial Application. *Science of Computer Programming*, 40:59–96, 2001.

[Mur89] Tadao Murata. Petri Nets: properties, Analysis and Applications. In *Proceedings of the IEEE*, pages 541–580, April 1989.

[OSA$^+$13] M. V. M. Oliveira, A. C. A. Sampaio, P. R. G. Antonino, R. T. Ramos, A. L. C. Cavancalti, and J. C. P Woodcock. Compositional Analysis and Design of CML Models. Technical report, COMPASS Deliverable, D24.1, 2013. Available at http://www.compass-research.eu/.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[POR12] Hristina Palikareva, Joël Ouaknine, and A. W. Roscoe. Sat-solving in csp trace refinement. *Sci. Comput. Program.*, 77(10-11):1178–1197, September 2012.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Nasa Formal Methods, Third International Symposium, NFM 2011*, pages 298–312, Pasadena, CA, USA, April 2011. NASA, Springer LNCS 6617.

[RBH84] A. W. Roscoe, S.D. Brookes, and C.A.R. Hoare. A theory of communicating sequential processes. *Journal of the ACM*, (3):560–599, July 1984.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, September 2003.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, London Dordrecht Heidelberg New York, 2010.

[RvBW06] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[SLDP09] Jun Sun, Yang Liu, JinSong Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer Berlin Heidelberg, 2009.

[SLS+12] Ling Shi, Yang Liu, Jun Sun, JinSong Dong, and Gustavo Carvalho. An analytical and experimental comparison of csp extensions and tools. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 381–397. Springer Berlin Heidelberg, 2012.

[VMO02] Alberto Verdejo and Narciso Mart-Oliet. Implementing ccs in maude 2. In *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002*, pages 239–257. Elsevier, 2002.

[WC02] Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag.

[WCF05] J. C. P. Woodcock, A. L. C. Cavalcanti, and L. Freitas. Operational Semantics for Model-Checking *Circus*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 237–252. Springer-Verlag, 2005.