DOI 10.1007/s00165-015-0342-2
BCS © 2015
Formal Aspects of Computing (2015) 27: 975–1001

Formal Aspects of Computing

Model checking CML: tool development and industrial applications

A. Mota[1], A. Farias[2], J. Woodcock[3] and P. G. Larsen[4]

[1] Centro de Informática, Federal University of Pernambuco, Av. Jornalista Anibal Fernandes, s/n-Cidade Universitária, Recife, PE, CEP 50.740-560, Brazil
[2] Federal University of Campina Grande, Campina Grande, Brazil
[3] University of York, York, UK
[4] Aarhus University, Aarhus, Denmark

Abstract. A model checker is an automatic tool that traverses a specific structure (normally a Kripke structure referred as the model M) to check the satisfaction of some (temporal) logical property f. This is formally stated as $M \models f$. For some formal notations, the model M of a specification S (written in a formal language L) can be described as a labelled transition system (LTS). Specifically, it is not clear in general how usual tools such as SPIN, FDR, PAT, etc., create the LTS representation from a given process. Although one expects the coherence of the LTS generation with the semantics of L, it is completely hidden inside the model checker itself. In this paper we show how to create a model checker for L, using a development approach based on its operational semantics. We use a systematic semantics embedding and the formal modeling using logic programming and analysis (FORMULA) framework to this end. We illustrate our strategy considering the formal language COMPASS modelling language (CML)—a new language that was based on CSP, VDM and the refinement calculus proposed for modelling and analysis of systems of systems. As FORMULA is based on satisfiability modulo theories solving, our model checker can handle communications and predicates involving data with infinite domains by building and manipulating a symbolic LTS. This goes beyond the capabilities of traditional CSP model checkers such as FDR and PAT. Moreover, we show how to reduce time and space complexities by simple semantic modifications in the embedding. This allows a more semantics-preserving tuning. Finally, we show a real implementation of our model checker in an integrated development platform for CML and its practical use on an industrial case study.

Keywords: CML, Model checker, Analysis, FORMULA, Operational semantics, SMT

1. Introduction

It is very common, and with certain frequency, to find news about the creation of a new model checker T for a certain formal specification (or programming) language L [CGL94]. There are two main reasons for this continuous creation:

1. The language L is new and specifically designed to deal with a certain kind of problem that justifies the existence of model checking support; or
2. The model checker T is better (for instance, it has a superior performance or can handle a larger class of problems) than another model checker T' for the same language L.

Correspondence and offprint requests to: A. Mota, E-mail: acm@cin.ufpe.br

Although a model checker should obey the structured operational semantics (SOS) of a formal language L [Plo04], it is created, in general, as any other software. That is, it can be error-prone itself. In this sense, a development process strongly associated to the SOS of L results in an immediate benefit: the correctness of the model checker by construction (with respect to the correctness of the underlying implementation infrastructure).

In this paper we propose a new model checker based on three main arguments:

1. the language CML [WCF+12] is new and there is no available tool support for it;
2. we cannot create a CML model checker by reusing previous model checkers in a semantic preserving way; and
3. our proposed model checker follows the SOS of CML clearly and directly, as opposed to the general practice for previous formal languages.

This work is one of several efforts in the context of the COMPASS project [FLW14]. It is implemented as a feature of the Symphony tool platform [CML+12].[1] The Symphony tool provides syntax and type checking, interpretation/debugging, proof obligation generation, theorem proving, model checking, test automation and a connection to the Artisan Studio SysML tool where static fault analysis additionally is supplied. Symphony is based on COMPASS modelling language (the CML), a semantic combination among CSP, VDM and refinement calculus, whose purpose is to provide support for modelling systems of systems (SoS).

The approach presented here has a low adaptation cost to become applicable to any language with CSP-like concurrency and communication and rich state expressed in B, VDM, or Z, or even an imperative programming language like C. So, the work is really applicable to CSP ‖ B [ST02], Circus [WCF05] and Jin Song Dong's CSP# [SLDC09], as well as languages that can be translated into these more basic languages, such as Wellings and Cavalcanti's Safety-Critical Java [ZCW11]. The semantic link between all these languages together is UTP. So, the choice of CML is really made for two reasons: (i) because we needed a model checker for COMPASS; and (ii) because it is a concrete exemplar of a whole class of languages with quite a diverse user base.

Concerning the class of problems we can deal with, consider the following simple CML process.

```
channels a: int

process Main =
begin
  actions
   P = a?x -> a?y ->
            if x <> y and x*x > 9
            then Stop
            else P
  @ P
end
```

The process `P` receives two integers `x` and `y` on channel `a`. After that, if these values are different (`x <> y`) and `x*x > 9` the system deadlocks; otherwise, it behaves as `P` again. Process `P` cannot be handled by traditional CSP model checkers, such as FDR [Ros98, Ros10] or PAT [LSD11], because it can accept infinite values (**`int`**) for `x` and `y`, as well as the predicate `x <> y` is completely unbounded.[2] To use traditional model checkers, one has to use a subset of the integers and be smart enough to guarantee that at least one combination of values for `x` and `y` satisfies the predicate `x <> y and x*x > 9` as well as falsifies it. For simple problems, this is trivial. However, for practical situations, the usual is to have more complex predicates (due to dependencies among them, for example) and the user can loose important properties of the original system if data values are not chosen carefully.

The model checker we propose in this paper follows directly from the SOS of CML in a systematic manner. We use some ideas of [Leu01] and [Lee85] of capturing an SOS by a deep embedding (that is, representing the semantic model as a transition machine stated in terms of logical facts), except for the underlying semantics of our implementation platform (we use the Microsoft Research framework formal modeling using logic programming and analysis (FORMULA) [JLB11]) and, as a consequence, on how the logical rules may be stated. We capture data aspects of a language by a shallow semantics embedding, that is, capturing expressions directly in terms of the elements available in the implementation platform.

[1] More information available at: http://symphonytool.org/.
[2] There are infinite valuations satisfying the predicate.

As we adopt the generic framework FORMULA [JLB11], which has a declarative language front-end similar to Prolog but with a back-end that works in cooperation with the Z3 SMT solver [dMB11], we can build symbolic model checkers able to handle communications and predicates involving data with infinite domains.

The embedding investigated in this work is a result of an evolution of previous embeddings in a product line for developing model checkers. The embedding of CSP has been first produced focused on behavioural aspects [MF13b]. It resulted in an apparatus to be reused to handle behavioural aspects in CML embedding. Then, the idea has evolved by also including data aspects, considering the operational semantics of Circus [MFDW14, MF14]. Afterwards, we have introduced the concepts of VDM in this embedding and added time aspects according to CML's operational semantics.

Although our CML model checker is based on a formal semantics, we developed a complementary testing campaign based on expected behavioural properties (algebraic laws) as a preliminary investigation on the soundness of our model checker. In this campaign the tests covered the key CSP operators and captured features of the CSP SOS, so that we could compare the results of our model checker against the results of FDR. A more detailed comparison of our model checker with others existing tools is pointed out as future work in Sect. 7.

Finally, we employ our CML model checker on a case study provided by the COMPASS industrial partner Insiel.[3]

The main contributions of this paper are:

- a more intuitive and systematic model checker development strategy based on the SOS of a given concurrent language;
- the use of FORMULA for implementing model checkers in a fast, sound and powerful way;
- manipulation of infinite data communications and predicates by the created model checker;
- integration of our model checker with a wider development environment (Symphony IDE);
- tuning model checker performance by simple semantic manipulation;
- applying the proposed CML model checker to an industrial case study.

This work is organised as follows. Section 2 presents a sufficient background on CML. Section 3 provides material regarding the Microsoft FORMULA framework and the embedding of CML in it. In Sect. 4 we present the kind of properties one is usually interested in investigating a system and their corresponding encoding in FORMULA. Section 5 addresses the use of our model checker inside the Symphony IDE. Finally, we present some related work in Sect. 6 and our conclusions and future work in Sect. 7.

2. Background

In this section we provide a background on the theory and on the framework we use in our approach.

2.1. Microsoft FORMULA

We present the underlying framework—Microsoft FORMULA—to build and analyse a CML specification.

FORMULA [JLB11] is a modern framework that follows the principles of model-based development and is based on algebraic data types and strongly-typed constraint logic programming (CLP). It supports concise specifications of abstractions (in a Prolog-like style) and model transformations. As FORMULA uses SMT solving, it has the advantage of providing model finding and design space exploration facilities. Thus, FORMULA can be used to construct system models satisfying complex domain constraints. The internal activity of FORMULA consists of a bottom-up least fixpoint search based on the Z3 SMT solver also from Microsoft.

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 operators ($n \geq 1$), 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;

[3] http://www.insiel.eu/.

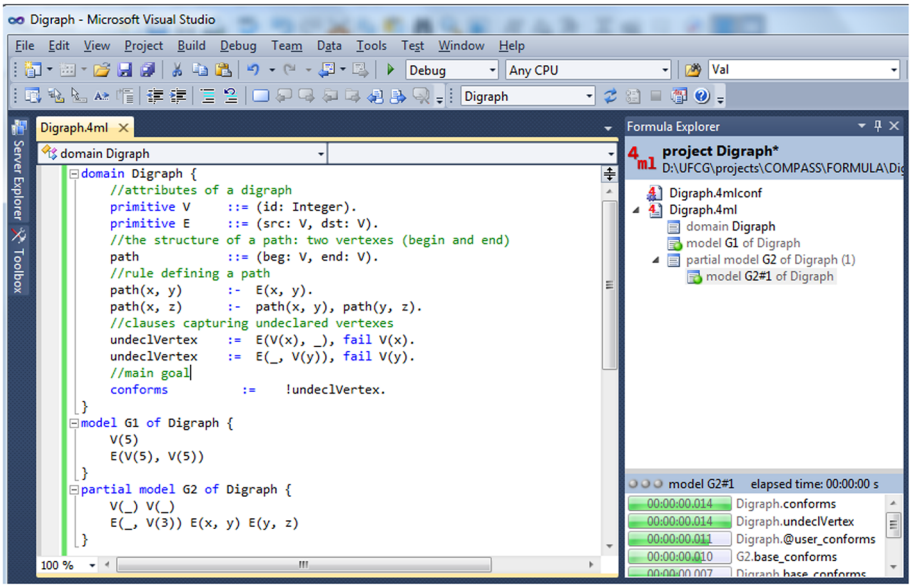

Fig. 1. FORMULA snapshot model analysis

  - *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 right-hand side (RHS) is the head and the left-hand side (LHS) is the body of the rule (a list of facts used to derive the RHS). For every element X used in the RHS, we must have some constructor $Cons(X)$ in the LHS to constrain the possible values of X; FORMULA can only build the head from the elements of the body (bottom-up approach);
  - *queries* predicates in terms of the (primitive) 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 elements similar to 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).

We illustrate the work of FORMULA using an example that captures the essence of a basic Digraph (see Fig. 1). The choice for this example was based on a quick understanding of FORMULA. Although it does not explicitly represent a labelled transition system (LTS) to be analysed, it is useful to allow a brief discussion about searching and solving. Actually, FORMULA creates new facts in its logical database by using the rules on non-primitive constructs (this can be viewed as the searching part). When no new fact can be created, the solver tries to fill the primitive facts with data that yields a valid model (this is the solving part where FORMULA automatically reuses Z3 [JKD+10]).

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 element `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` constraint defines a final 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 undertermined (`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.

2.2. The CML language

We start by introducing CML focusing on its features and operational semantics. CML is the first formal language specifically designed for modelling and analysing SoSs. It is founded on the well-established formalisms Circus [WC02] and VDM [FLV08], and combines a number of aspects required in SoS modelling such as discrete time, concurrency, processes, state and contracts [WCF+12].

A CML model is basically composed by the following elements:

- *types* defining new types (numeric, lists, sequences, sets, records, union types, etc.) to be manipulated along the specification and possible invariants over them;
- *functions* establishing maps between input and output types, possibly containing pre- and post-conditions;
- *channels* useful to define elements over which systems can communicate messages;
- *processes* model elements consisting of:
  - *state* establishing mutable variables to be manipulated by its process;
  - *operations* similar to functions but also working over state variables;
  - *actions* specifying reactive behaviour such as calls, message passing, timeout, choices, etc.

2.2.1. Syntax of CML

The syntax of CML has been defined in [WM12] and contains constructs to provide support for several features. Here, we select the subset of the original grammar to show the embedding of CML constructs in FORMULA. Although we present construct for actions, processes can be combined by using the same operators and have a similar understanding.

```
action =
      Skip | Stop | Diverge                          (Basic actions)
      | Wait expression                               (Delay action)
      | communication -> action                      (Communication action)
      | [ expression ] & action                      (Guarded action)
      | action ; action                              (Sequential composition)
      | action |~| action                            (Internal choice)
      | action [] action                             (External choice)
      | action / [ expression ] \ action             (Timed Interrupt)
      | action, [ [ expression ] > action            (Timed Timeout)
      | action \\ chanset expression                 (Hiding)
      | parallel actions
      | control statements ;

parallel actions =
      | action ||| action                                    (Interleaving)
      | action [| chanset expression |] action   (Generalised parallelism) ;

control statements =
      | if statement                          (Conditional choice statement)
      | call statement                                  (Call statement) ;

if statement =
      if expression then
        action
      [ else
          action ]                                  (Conditional statement);
```

According to the CML semantics, the primitive actions **Skip** and **Stop**, respectively, mean immediate termination and immediate deadlock. The basic action **Diverge** means a *divergent* behaviour—it runs without ever interacting in any observable event (similar to an infinite loop doing nothing). The delay action '**Wait** t' does nothing for 't' units of time and then terminates successfully. The communication action 'comm -> P' offers the communication event 'comm' to its environment, and after its occurrence, it behaves as 'P'. When values may be exchanged between processes, we use the constructs 'ch!exp' to send the value corresponding to expression 'exp' (output event) and 'ch?x' to receive a value and store it in the variable 'x' (input event); otherwise, the event can be simply denoted by the channel name 'ch'. The guarded action '[cond] & P' uses the boolean condition 'cond' to behave as 'P', if 'cond' is valid, or **Stop**, otherwise. The sequential composition 'P;Q' represents a process that behaves as 'P' until 'P' terminates successfully, then the composition behaves as 'Q'. The internal choice 'P |~| Q' nondeterministically behaves either as 'P' or 'Q'. The external choice 'P [] 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 timed interrupt action 'P / t \ Q' allows 'P' to continue for 't' number of time units, after which it will be interrupted by 'Q'. This operator is not invoked if 'P' terminates before the time value 't'. If 't' is not specified, the action 'P /_\ Q' behaves as 'P' until 'Q'; that is, if any of the initial events of 'Q' occurs (at any time), the combined process starts to behave like 'Q'. The timed timeout action 'P [t > Q' behaves as 'P' for 't' units of time, then it behaves as 'Q'. If 'P' fails to begin a communication, 'Q' silently takes over. If 't' is not specified, the action 'P [_> Q' behaves as 'P', but can change the behaviour to 'Q' nondeterministically at any time. The hiding action 'P \\ s' behaves as 'P' but hiding the events in the set 's'. The interleave 'P ||| Q' means the parallel execution (with no synchronisation) of 'P' and 'Q'. On the other hand, the generalised parallelism 'P [|cs|] Q' executes 'P' and 'Q' in parallel synchronising on the events in the set 'cs'. The conditional choice '**if** cond **then** P **else** Q' behaves as 'P' if 'cond' is valid or as 'Q', otherwise. Finally, a call statement 'P(params)' invokes the action 'P' using 'params' as parameters (optional). This construct is useful for handling recursion in actions and processes.[4]

2.2.2. *Operational semantics of CML*

The semantics of CML has been originally defined in terms of Hoare and He's unifying theories of programming (UTP) [HH98]. This CML UTP semantics was formally rewritten [CW13] in terms of a SOS following a Plotkin-style [Plo04]. We present this SOS semantics and use it in a way closer to how a model checker works.

The operational semantics is described by a transition relation for the language constructs. These rules can be used to define an abstract interpreter for the language, giving possible execution steps for CML processes. This description can then be used as a guide for implementing model checking, refinement checking, animation, and test automation [CW13].

The semantics deals with a CML program text and its current state, which is an assignment to all the program variables in scope. This state is structured into global and local variables. So for example, when the program text is the parallel composition of two actions, each may have its own local state as well as there being a global state that persists beyond both their lifetimes [BGW12].

The operational meaning of a CML action is a computation: a sequence of individual steps that the action can make as it executes. These steps are represented by a transition relation between individual machine states: (s, P). Here, s is a text assigning constants to alphabetical variables and P is an action text. The pair represents the current state of the computation, s, and the action yet to be executed, P. It is important to note that the transition relation relates *syntactic* objects, not semantic ones. When we need to relate syntax to semantics, we write P to describe the syntax of a action P with semantics P.

If we allow **Skip** to represent program termination, then (s, **Skip**) is a terminal execution state, where s is the final value of the computation.

A transition relation describes how the system moves from a start state (s, P) to a next state (t, Q) via a transition with label (or event) l. Its generic representation is given by

$$(s, P) \xrightarrow{l} (t, Q).$$

The transition relation will be given inductively over the syntax of the language. Here, the after-state (t, Q) is one possible outcome of executing P starting from s. We hereafter assume basic knowledge in UTP and operational semantics.[5]

[4] There is also a specific constructor (**mu**) for recursion. The work presented here deals with explicit recursion (directly defined in action's body).

[5] More details about firing rules of CML operational semantics can be found in [CW13].

$$\frac{c}{(c \mid s \models \texttt{Div}) \xrightarrow{\tau} (c \mid s \models \texttt{Div})} \quad \textbf{(Div)}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1 \texttt{ ; } B) \xrightarrow{1} (c_2 \mid s_2 \models A_2 \texttt{ ; } B)} \quad \textbf{(Sequence progress)}$$

$$\frac{c}{(c \mid s \models \texttt{Skip ; } A) \xrightarrow{\tau} (c \mid s \models A)} \quad \textbf{(Sequence end)}$$

$$\frac{c}{(c \mid s \models A_1 \texttt{ |~| } A_2) \xrightarrow{\tau} (c \mid s \models A_1)} \quad \textbf{(Nondeterministic Choice left)}$$

$$\frac{c}{(c \mid s \models A_1 \texttt{ |~| } A_2) \xrightarrow{\tau} (c \mid s \models A_2)} \quad \textbf{(Nondeterministic Choice right)}$$

$$\frac{}{(c \mid s \models A_1 \texttt{ [] } A_2) \xrightarrow{\tau} (c \mid s \models (\texttt{loc } c \mid s \texttt{ @ } A_1) \texttt{ [+] } (\texttt{loc } c \mid s \texttt{ @ } A_2))} \quad \textbf{(External Choice begin)}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_3 \mid s_3 \models A_3) \quad l \neq \tau}{(c \mid s \models (\texttt{loc } c_1 \mid s_1 \texttt{ @ } A_1) \texttt{ [+] } (\texttt{loc } c_2 \mid s_2 \texttt{ @ } A_2)) \xrightarrow{1} (c_3 \mid s_3 \models A_3)} \quad \textbf{(External Choice end)}$$

$$\frac{}{(c \mid s \models (\texttt{loc } s_1 \texttt{ @ Skip}) \texttt{ [+] } (\texttt{loc } s_2 \texttt{ @ } A)) \xrightarrow{\tau} (c \mid s_1 \models \texttt{Skip})} \quad \textbf{(External Choice Skip left)}$$

$$\frac{}{(c \mid s \models (\texttt{loc } s_1 \texttt{ @ } A) \texttt{ [+] } (\texttt{loc } s_2 \texttt{ @ Skip})) \xrightarrow{\tau} (c \mid s_2 \models \texttt{Skip})} \quad \textbf{(External Choice Skip right)}$$

$$\frac{c \quad s\,;\,g}{(c \mid s \models [g] \texttt{ \& } A) \xrightarrow{\tau} (c \texttt{ and } (s \texttt{ ; } g) \mid s \models A)} \quad \textbf{(Guard)}$$

$$\frac{(out(\alpha(s)) = \{x_1', x_2'\})}{(c \mid s \models A_1 \texttt{ [|} x_1 \texttt{ | } cs \texttt{ | } x_2 \texttt{|] } A_2) \xrightarrow{\tau} (c \mid s \models (\texttt{loc } (s \mid x_1)_{+x_2} \texttt{ @ } A_1) \texttt{ [|} x_1 \texttt{ | } cs \texttt{ | } x_2 \texttt{|] } (\texttt{loc } (s \mid x_2)_{+x_1} \texttt{ @ } A_2))} \quad \textbf{(Parallel begin)}$$

Fig. 2. Firing rules for CML

Figure 2 shows some firing rules of CML. The rules define a way of evolving a CML process/action from a state to another.[6] This is basically what we need to build LTSs, as explained in Sect. 3.

2.2.3. Classical properties of CML specifications

Similar to other languages like CSP and Circus, CML specifications can also be analysed in terms of three classical properties: deadlock, livelock and nondeterminism. These properties are defined like in CSP and have a strict relation with the graph representation (LTS):

- *deadlock* a process is deadlocked if it reached some state (different from successful termination) where no transitions can be fired;
- *livelock* a process has a livelock if it performs a τ-loop (a loop of internal or τ-transitions);
- *nondeterminism* a process is nondeterministic if it decides to accept or reject a same event.

[6] For a complete understanding of the notation and all rules (including rules for time semantics) please refer to [CW13].

3. Capturing the CML SOS

In this section we present how to capture the SOS of CML in FORMULA. This provides a systematic way of creating a model checker from the operational semantics of a formal language in a fast and very intuitive manner, similar to Leuschel [Leu01] and Verdejo [Ver00].

The work towards model checking assumes that the model M is given and focuses on formally describing what means $M \models f$, or how to check f by traversing M. For languages whose syntax are closer to an LTS, and that do not use complex data representations, the model M can be represented more directly and, thus, it is easy to justify that LTS was expressed (or captured) correctly. Nevertheless, for languages such as CSP [Ros98, Ros10], PROMELA [GM99] and Circus [WCF05], creating a model checker by a direct programming approach can be too error-prone. In practice, most model checkers only creates 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. The same applies to the property f. But for f, there is a standard logical way of creating correct algorithms by reusing traditional graph traversal algorithms.

In this section we show how to capture the firing rules of the CML SOS in FORMULA so that the LTS is directly derived from a conceptual (and formal) model. The semantics of a complex language might have several aspects, such as, for example, data aspects and control (or concurrent). The ideal situation is 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 programming framework. This is frequently the case of data aspects, where the programming framework already provides 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 *shallow embedded* in the programming framework.

3.1. Basic shallow embedding in FORMULA

We propose a way to capture the SOS of CML using a so called hybrid semantics embedding in which behavioural aspects are captured in a deep embedding way and data aspects are not interpreted. They are simply mapped in the available elements, yielding a shallow embedding. Although FORMULA provides basic data types (**`integer`**, **`natural`**, **`real`** and **`string`**), more complex types (like sets, relations, functions, sequences, bags, etc.) are absent and the mapping is not so direct. The domain `AuxiliaryDefinitions` (see Fig. 3) has the purpose of modelling types (basic types and complex) and operations over them.

The type `VOID` (line 3) is just a bottom value for all possible types. The most basic types (natural, integer, real and strings) are directly reused from FORMULA. The sequence type is defined by the constructor (`SeqType` in line 6) that is the union of two types (inductively defining a sequence). The constructor `EmptySeq` defines an empty sequence and `NonEmptySeq` defines a non-empty sequence as a tuple containing the head (an element) and a tail (another sequence).

The set type definition is similar to that of sequences and also provides a recursive description of sets in FORMULA (lines 8–10).

Operations over sequences and sets are also provided as auxiliary definitions. They describe how the usual operations (cardinality, union, intersection, concatenation, etc.) work in terms of FORMULA representations. We omit them here to save space.

The `ProductType` is intended to represent the product of types of CML as a pair of two types.

The user defined types are also added in this FORMULA section. The inclusion of these types is necessary to turn them available to create new types or be simply reused along the specification, which is achieved by union of types (line 18) including basic types (**`integer, natural, real, string`**), complex types (`SeqType, SetType`), product type (`ProdType`) and user defined types. For each basic type and user defined type, there must be a constructor to provide support for instantiation of their values by FORMULA. These constructors are called wrappers (lines 20–28) and have a specific use: they allow their use in facts where the internal types are instantiated by FORMULA instead of given by the user or computed by the specification.

The constructor `ParamW` (line 30) is a wrapper specific for parameters. A parameter has a name (normally the name of the action or process using it) and a value.

```
1  domain AuxiliaryDefinitions {
2    //Types
3    VOID                    ::= {void}.
4    EmptySeq                ::= { emptySeq }.
5    primitive NonEmptySeq   ::= ( first: Type, rest: SeqType ).
6    SeqType                 ::= EmptySeq + NonEmptySeq.
7
8    EmptySet                ::= {emptySet}.
9    NonEmptySet             ::= (head:Type,tail:SetType).
10   SetType                 ::= EmptySet + NonEmptySet.
11
12   //definitions of operations over sets and sequences
13
14   //A representation for product type
15   primitive ProdType ::= (first: Type, second: Type).
16
17   // User defined types
18   Type ::= VOID + Boolean + Integer + Natural + String + Real + SeqType + SetType + ProdType +
19            User Defined Types.
20
21   //wrappers for all types.
22   primitive IntegerW ::= (Integer).
23   primitive NaturalW ::= (Natural).
24   primitive RealW ::= (Real).
25   primitive BooleanW ::= (Boolean).
26   primitive StringW ::= (String).
27
28   //Wrappers for user defined types
29
30   //Wrappers for parameters
31   primitive ParamW ::= (name:String, value:Type).
32
33   // Bindings
34   NullBind            ::= {nBind}.
35   primitive SingleBind ::= (name: String, val:Type).
36   primitive BBinding    ::= (b: SingleBind, rest: Binding).
37   Binding       ::= NullBind + BBinding.
38
39   //operations over bindings
40
41   //Guard evaluation to handle boolean expression evaluation
42   guardDef      ::= (id: Natural, st: Binding).
43   guardNDef     ::= (id: Natural, st: Binding).
44 }
```

Fig. 3. Representations for types and operations over them in FORMULA

The constructor for bindings (lines 33–36) are intended to define a set containing maps from variable to values. Operations over bindings are also included in this section.

Finally, the constructors `guardDef` and `guardNDef` represent the evaluation of an associated boolean expression (the `id` identifies the evaluation). That is, each expression to be evaluated can have `guardDef` and `guardNDef` facts describing the effect over bindings when the expression is valid or not, respectively.

3.2. Capturing the CML syntax

The CML SOS is captured as in the real scenario: the syntax and semantics are described in two separated domains (syntax and semantics). The syntax domain defines the structures (building blocks) necessary to represent CML constructs for events and processes. Figure 4 illustrates the FORMULA representation for some CML constructs presented in Sect. 2.2.1.

The support for describing communications includes channel definition (line 3), the communication event itself (line 4) and its effect over the bindings (lines 5–6). In this representation, communications without values use the value **`void`** for the type being communicated. Events are represented by different constructs. The set of all possible visible events[7] is defined by union of types (line 7). Special events are defined in lines 8–9 and the type for all possible events (visible, internal and time) is defined in line 10.

[7] All events except for τ (internal event).

```
1  domain CMLSyntax extends AuxiliaryDefinitions {
2    //Useful definitions related to communications
3    Channel             ::= (chN : String, chT: Type).
4    primitive CommEv    ::= (chName: String, chExp: String, val: Type).
5    primitive IOComm    ::= (id: Natural, chName: String, chExp:String, val: Type).
6    IOCommDef           ::= (id: Natural, val: Type, st: Binding, st_: Binding).
7    Sigma               ::= CommEv + IOComm.
8    Tau                 ::= {tau}.              //Internal event
9    Tock                ::= {tock}.             //Time event
10   SigmaTauTock        ::= Sigma + Tau + Tock. //All possible events
11
12   //Actions and Processes
13   BasicProcess        ::= {Stop, Skip, Div}.
14   primitive Prefix    ::= (ev : Sigma, proc : CMLProcess).       //Prefix
15   primitive iChoice   ::= (lProc:CMLProcess,rProc:CMLProcess).   //Internal choice
16   primitive eChoice   ::= (lProc:CMLProcess,rProc:CMLProcess).   //External choice
17   primitive intrpt    ::= (lProc:CMLProcess,rProc:CMLProcess).   //Interrupt
18   primitive uTimeout  ::= (lProc:CMLProcess,rProc:CMLProcess).   //Timeout
19   //Timed Interrupt
20   primitive tIntrpt   ::= (lProc:CMLProcess,t:Natural,rProc:CMLProcess).
21   //Timed Timeout. Useful to model Wait action. Wait N = Stop [_ N _> Skip
22   primitive tTimeout  ::= (lProc : CMLProcess, t:Natural, rProc : CMLProcess).
23   //A special operator of CML.The usual external choice is transformed into it
24   extraChoice   ::= (lSt:Binding,lProc:CMLProcess,rSt:Binding,rProc:CMLProcess).
25   //Conditional choice. Guarded choice are mapped to  this constructor
26   primitive condChoice  ::= (id:Natural,procTrue:CMLProcess,procFalse:CMLProcess).
27   //Sequential composition
28   primitive seqC      ::= (lProc : CMLProcess, rProc : CMLProcess).
29   primitive hide      ::= (proc : CMLProcess, hideS : String).   //Hiding
30
31   //Generalised parallelism
32   primitive genPar ::= (lProc: CMLProcess, SyncS : String, rProc: CMLProcess).
33   primitive par    ::= (lSt:Binding, lProc:CMLProcess, SyncS:String,
34                         rSt:Binding, rProc:CMLProcess).
35
36   primitive proc       ::= (name : String, param:Type).          //Process call
37   primitive operation  ::= (opName: String,opPar:Type).          //Operation call
38
39   //operations
40   operationDef         ::= (opName: String, opParam:Type,st: Binding, st_: Binding).
41   preOpOk              ::= (opName: String, opParam:Type,st: Binding).
42
43   //All possible actions and processes
44   CMLProcess  ::= BasicProcess + Prefix + iChoice + eChoice + extraChoice +
45                   condChoice + seqC + hide + par + genPar +  proc + var + let +
46                   operation + assign + intrpt + uTimeout + tIntrpt + tTimeout.
47
48   //Action/process definition
49   ProcDef     ::= (name:String, params:Type, proc:CMLProcess).
50
51   //Constructor related to synchronisation
52   lieIn         ::= (ev : Sigma, sourceSet: String).
53 }
```

Fig. 4. CML syntax represented in FORMULA

The representation of actions (and processes) starts by describing the basic actions **`Stop`**, **`Skip`** and **`Diverge`** in the element `BasicProcess` (line 13). The `Prefix` (communication action) is represented as a pair of an event (from `Sigma`) and the next behaviour. 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 (lines 15–16). The constructors for interrupt (`intrpt`) and untimed timeout (`uTimeout`) are binary and contain a left and a right process (lines 17–18). Their timed versions (`tIntrpt` and `tTimeout`) also contain a natural number to capture the units of time. Particularly, the constructor `tTimeout` is used to represent the delay action (**`Wait`** `N`) as it can be expressed as **`Stop`** `[_ N _>` **`Skip`**. The constructor `extraChoice` represents a special operator used by the semantic rules of external choice (see Fig. 2), where the usual external choice (`[]`) is translated into such

an operator (`[+]`) to which the firing rules are defined. The conditional choice constructor (`condChoice`) has three components: a boolean condition identifier, a process defining the behaviour if the condition is valid and another process defining the behaviour if the condition is invalid. The evaluation of the condition is captured by the constructor (`guardDef` and `guardNDef`) defined in the auxiliary domain to avoid direct interpretation in FORMULA.[8] 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 the 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. Specially the facts establishing that an event belongs to a set are stated by the constructor `lieIn` (line 52), which has a visible event and a set of events.

The generalised parallelism is represented by the constructor `genPar` (line 32). It contains a left process, a synchronisation set and a right process. The constructor `par` (lines 33–34) represents a special operator used by the semantic rules of generalised parallelism (see Fig. 2), where bindings are considered in each constituent part of the parallel composition.

A process call is represented by the constructor `proc` (line 36) that contains a process name and its parameters.

The constructor `operation` (line 37) represents an operation call anc contains the operation name and its parameters.

An operation itself is represented by the constructor `operationDef(opName,opParam,st,st_)` (line 40), where `opName` is the name of the operation, `opParam` its parameter and `st` and `st_` are the bindings representing the state before and after executing the operation. The constructor `preOpOk(opName,opPar,st)` (line 41) syntactically represents the precondition and actually is used to establish a fact to be created if the precondition is holds. Ideally, we should embed the operation body once and calculate the validity of its precondition. However, this is not possible due to some FORMULA requirements.[9] As a solution, we use the precondition expression to generate specific facts separately from operation's body.

The constructor `CMLProcess` (line 44) defines (syntactically) all possible processes as the union of all kinds of processes previously defined.

Finally, the constructor `ProcDef` (line 49) captures a process definition containing a name, its parameters and the corresponding body.

3.3. Deep embedding of CML 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, Ver00]: one-to-one mapping for each firing rule. The deep embedding rules are defined in a specific section (the semantic domain) that extends the syntax domain.

The first definitions of the semantic domain are related to the underlying LTS structure: states and transitions.

```
1 domain CMLSemantics extends CMLSyntax {
2   State ::= (b: Binding, p: CMLProcess).
3   trans ::= (source: State, ev: SigmaTauTock, target: State).
4   primitive Clock ::= (Natural).
5   primitive GivenProc ::= (name: String).
6   ...
7 }
```

The constructor `State` captures any possible state (or context) of a CML 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. Transitions also consider the time event (**`tock`**) due to the timed semantics of CML. Note that these constructors are derived because these elements will be generated during the LTS construction. Thus, one can think about the LTS being represented by FORMULA facts where `State` are the nodes of the LTS and `trans` are the transitions between states.

The constructor `Clock` is useful to consider time events during the analysis, where `Clock(N)` means that `N` units of time have passed.

The constructor `GivenProc` has the purpose of defining the starting point of the analysis: the name of the process to be analysed. From the name, FORMULA recovers the body and starts the LTS generation.

[8] Design decision due to performance.

[9] This actually originates non-stratified models, that are not allowed by FORMULA.

Below the representation of each firing rule presented in Fig. 2 in terms of FORMULA transitions and states is shown. The transitions involving time are not represented here but follow a similar way.

No transition

Similar to CSP, the action **`Stop`** does not have associated transition because it represents a final state. In CML, **`Skip`** also represents a final state and has no transition (differently from CSP that defines a special transition for termination).

On demand state creation

Except for the initial state, the LTS creation requires instantiation of new states and transitions. The existence of a transition is associated with the existence of its source state (besides other conditions as defined by the firing rules). Thus, whenever a new transition is created we need to provide a dynamic creation of the (next) states as well (which are essential to keep going on the LTS expansion). This is achieved by the general rule

```
State(st,body):- trans(S,ev,State(st,body)).
```

The above rule guarantees the creation of the final state of a transition (and, more importantly, the initial state of a new transition).

Divergent behaviour

The divergent behaviour originates a transition to itself. This is represented by

```
trans(iS,tau,iS):- iS is State(st,Div).
```

Prefix

The representation for communication action in FORMULA is intuitive as it simply creates a transition labelled with an event to the next behaviour:

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

As long as there is a state where a communication is ready to be performed (`IOComm(id,chName,...)`) and its respective effect over bindings (`IOCommDef(id,...)`) also exists, there must be a transition from such a state via a corresponding event (`CommEv(chName,chExp,chType)`) to the next state (`State(st_,P)`). Although the dynamic creation of states would be responsible for creating the `State(st_,P)`, its occurrence in the LHS of the rule is to avoid such a creation only in the next FORMULA iteration.

Nondeterministic choice

Recall from the firing rule for nondeterministic choice that internal progress can occur in both elements. This originates two rules in FORMULA:

```
State(st,P),
trans(State(st,iChoice(P,Q)),tau,State(st, P)):- State(st,iChoice(P,Q)).
State(st, Q),
trans(State(st,iChoice(P,Q)),tau,State(st, Q)):- State(st,iChoice(P,Q)).
```

Note that the rules of the internal choice create both next states and internal transitions to them.

External choice

The firing rules for external choice have some subtleties that are explained accordingly. First, the usual external choice (`[]`) moves to another operator (`[+]`), where the binding is copied to each constituent part. As long as

the state `State(st,eChoice(P,Q))` state is reached, the system moves (via a τ transition) to `State(st, extraChoice(st,P,st,Q)`. The creation of the states in lines 1–3 is an optimization to create, in the current iteration, the premises for the next expansions.

```
1 State(st,P),
2 State(st,Q),
3 State(st, extraChoice(st,P,st,Q)),
4 trans(iS, tau, State(st, extraChoice(st,P,st,Q))):- iS is State(st,eChoice(P,Q)).
```

If one of the operands terminates, the external choice also terminates. That is, the `extraChoice (st1,Skip,_,_)` evolves to `State(st1,Skip)` (or `extraChoice(_,_,st2,Skip)` evolves to `State (st2,Skip)`) via a τ transition.

```
1 State(st1,Skip),
2 trans(iS,tau,State(st1,Skip)):- iS is State(st,extraChoice(st1,Skip,_,_)).
3 State(st2,Skip),
4 trans(iS,tau,State(st2,Skip)):- iS is State(st,extraChoice(_,_,st2,Skip)).
```

If one of the operands have internal progress, the external choice does so accordingly. This is expressed as follows:

```
1  State(stP_,P_),
2  State(st,extraChoice(stP_,P_,stQ,Q)),
3  trans(iS,tau,State(st,extraChoice(stP_,P_,stQ,Q))):- //the combination evolves
4     iS is State(st,extraChoice(stP,P,stQ,Q)),
5     trans(State(stP,P),tau,State(stP_,P_)).      //there is internal progress
6
7  State(stQ_, Q_),
8  State(st,extraChoice(stP,P,stQ_,Q_)),
9  trans(iS,tau,State(st,extraChoice(stP,P,stQ_,Q_))):-
10    iS is State(st,extraChoice(stP,P,stQ,Q)),
11    trans(State(stQ,Q),tau,State(stQ_,Q_)).
```

Whenever one of the constituent parts can evolve via a visible event, the external choice behaves accordingly. That is, the entire combination evolves, via a visible event, to the next behaviour of the corresponding constituent part. This is expressed as follows:

```
1 State(st3,P_),
2 trans(iS,ev,State(st3,P_)):- iS is State(st,extraChoice(st1,P,st2,Q)),
3           trans(State(st1,P),ev,State(st3,P_)), ev != tau, ev != tock.
4 State(st3,Q_),
5 trans(iS,ev,State(st3,Q_)):- iS is State(st,extraChoice(st1,P,st2,Q)),
6           trans(State(st2,Q),ev,State(st3,Q_)), ev != tau, ev != tock.
```

Note that the conditions `ev != tau` and `ev != tock` are necessary to exclude internal and time events from this kind of progress.

Parallelism

The general parallelism is also defined in terms of two constructors: one considering only processes (`genPar`) and another considering the internal states of each component process (`par`). The first firing rule concerns the beginning of the parallel execution and it is represented by:

```
1 State(st,P),
2 State(st,Q),
3 trans(iS,tau,State(st,par(st,P,X,st,Q))):- iS is State(st,genPar(P,X,Q)).
```

The parallel execution (`genPar`) starts by originating a parallel execution (`par`) with copies of the original bindings to each component part.

Once a parallel state `par(...)` is created, it can progress only if its parts (premises) are available. The following rule has this purpose.

```
1 State(stP,P), State(stQ,Q):- State(st,par(stP,P,X,stQ,Q)).
```

The parallelism of two independent processes evolves only one of its internal parts if the event to be performed does not belong to the synchronisation set. This rule is written distinctly for both parts as follows:

```
State(st,par(stP_,P_,X,stQ,Q)),
trans(iS, ev, State(st,par(stP_,P_,X,stQ,Q))):-iS is State(st,par(stP,P,X,stQ,Q)),
   trans(State(stP,P),ev,State(stP_,P_)),fail lieIn(ev, X).

State(st,par(stP,P,X,stQ_,Q_)),
trans(iS, ev, State(st,par(stP,P,X,stQ_,Q_))):-iS is State(st,par(stP,P,X,stQ,Q)),
   trans(State(stQ,Q), ev, State(stQ_,Q_)), fail lieIn(ev, X).
```

Note that the fact `fail lieIn(ev,X)` belongs to the right-hand side of both rules. It means that the left-hand side facts are created only if the event `ev` is not a member of the set `X`.

The parallel synchronised, on the other hand, requires both parts agree on the same event (a member of the synchronisation interface).

```
State(st,par(stP_,P_,X,stQ_,Q_)),
trans(iS, ev1, State(st,par(stP_,P_,X,stQ_,Q_))):- iS is State(st,par(stP,P,X,stQ,Q)),
   trans(State(stP,P),ev1,State(stP_,P_)), trans(State(stQ,Q),ev2,State(stQ_,Q_)),
   ev1!=tau, ev2 != tau, lieIn(ev1, X), lieIn(ev2, X),
   ev1 = CommEv(chanName,_,value), ev2 = CommEv(chanName,_,value).
```

Finally, the synchronised parallelism ends when both parts reach a successful termination.

```
State(st,Skip),
trans(iS, tau, State(st,Skip)):- iS is State(st,par(stP,Skip,X,stQ,Skip)).
```

Sequential composition

The rules of sequential composition captures the two situations specific of the operator: when the first process evolves (lines 1–2) and when it terminates (lines 4–5). In the first situation, the composition moves to another composition where only the first process evolves. In the second situation, the composition simply evolves to the second process.

```
State(st_,seqC(P_,Q)),
trans(iS,ev,State(st_,seqC(P_,Q))):- iS is State(st,seqC(P,Q)),trans(State(st,P),ev,State(st_,P_))
    .
State(st,Q),
trans(iS,tau,State(st,Q)):- iS is State(st,seqC(Skip,Q)).
```

Process call

In terms of operational semantics, a process call is represented as a transition from the call itself to the body of the process. In FORMULA, this is expressed as follows:

```
trans(n,ev,State(st_,P_)):- n is State(st,proc(P,pPar)), RecoverBody(n,pBody),
trans(State(st,pBody),ev,State(st_,P_)).

State(st,pBody):- n is State(st,proc(P,PPar)), ProcDef(P,PPar,pBody).

RecoverBody ::= (s:State,p:CMLProcess).
RecoverBody(n, pBody):- n is State(st,proc(P,PPar)), ProcDef(P,PPar,pBody).
RecoverBody(n, pBody_):- n is State(st,proc(P,PPar)), ProcDef(P,PPar,pBody), pBody = proc(N,NPar),
     RecoverBody(State(st,pBody),pBody_).
```

Once the state `State(st,proc(P,pPar))` (line 1) has been reached during the LTS generation, we recover the body associated with the process name `P` (`RecoverBody(n,pBody)` in line 1) and require that there is a transition from such a process fragment `trans(State(st,pBody),ev,State(st_,P_))` (line 2). This causes the creation of a new transition from the initial state to the next state (after the body has been recovered). This is an optimisation to avoid body recovery via internal transitions and has improved the performance of FORMULA when handling parallelism. The constructor `RecoverBody` is useful for recovering the real body of a call, even when there are successive calls.

Operations

Recall from Sect. 3.2 that operations are syntactically represented by constructors capturing its precondition and another capturing its post condition. Furthermore, each precondition originates one rule that is useful to create a trigger fact to the transition associated with an operation execution. Such a fact is created only if the precondition holds. The general form of a precondition rule is

```
1 preOpOk(opName,opParam,st):- pre opName.
```

In the above rule, `pre opName` is a generic representation of the real precondition of operation `opName`. Indeed, it is the FORMULA sentence for the real precondition and, obviously, it depends on each operation. As it is in the right hand side of the rule, it enables the creation of the fact `preOpOk(opName,opParam,st)` only if the real precondition holds.

On the other hand, the generic representation of the transition associated with an operation execution is

```
1 trans(n,tau,State(st_,Skip)):- n is State(st,operation(opN,opPar)), preOpOk(opN,opPar,st),
2                                operationDef(opN,opPar,st,st_).
```

Is it worth noting that a transition associated to an operation leads to a state there the process component is `Skip`. This comes from the semantics of a CML operation. As long as the there is a state containing an operation call (`n is State(st,operation(opN,opPar))` whose corresponding precondition holds (`preOpOk(opN,opPar,st)`) and according to the corresponding operation definition (`operationDef(opN,opPar,st,st_)`), the transition is created. Furthermore, it is worth mentioning that the fact `operationDef` is operation dependent and is created for each operation defined in a CML specification. It incorporates the changes performed by the operations in variables and has the general form

```
1 operationDef(opN,opPar,st,st_):- n is State(st,operation(opN,opPar)), st=bindings, st_=bindings_,
2                                  var_ = update_expr.
```

An operation definition is captured by a fact that is created only if there is an operation call and the variables of the bindings after (with suffix _) are updated according with the expression `update_expr` (extracted from the post condition and represented in FORMULA).

4. Classical properties

In this section we show how to check classical properties of CML specifications in FORMULA. Our goal here is to show how to perform similar analysis as provided by FDR and PAT, but for unbounded state spaces (infinite data communications and predicates).

The rules presented previously guide FORMULA to build the LTS for a CML process according to its operational semantics. Once the LTS is created, FORMULA can walk through it to find specific properties. We focus on three classical properties because they are supported by CSP traditional tools like FDR and PAT:

The most suitable way to represent properties in FORMULA is using a specific domain that extends the semantic one, as illustrated by Fig. 5.

The notion of reachability is captured by the constructor `reachable` (lines 2–7). The first reachable state is initial one (associated with the body of the process to be analysed). Furthermore, all states reached by a transition or by recovering the associated state (for calls) are also reachable.

A deadlock (lines 9–10) is captured by the existence of a reachable state (different from `Skip`) from which we cannot recover a body (the state is not a call) neither go to any other state by a transition. The term `fail` means the absence of a specific fact in the language of FORMULA.

The constructor `tauPath` (lines 13–15) captures sequences of τ-transitions between two states. A `tauPath` can have only one transition or a `tauPath` to an intermediate state and another `tauPath` to the final state.

The constructor `accepts` (lines 19–21) captures the acceptances in a state. For a state `P`, this can be done by capturing directly the events of all outgoing transitions from `P` or from all states reached from `P` by invisible transitions.

The nondeterminism property (lines 23–25) is captured by checking the existence of two transitions with the same event (possibly τ-transitions) from the same state `L` (`trans(L, ev1, S1)` and `trans(L, ev1, S2))` leading to different states (`S1!=S2`) in which the process can accept (`accepts(S1, ev)`) or reject (`fail accepts(S1, ev)`) the same visible event (`ev!= tau`).

```
domain CMLProperties extends CMLSemantics {
  reachable    ::= (fS:State).
  //The initial binding is captured at initialisation
  reachable(State(b,PBody)) :- State(b,PBody), GivenProc(P),ProcDef(P,pPar,PBody).
  reachable(State(b,realBody)) :- reachable(State(b,P)),
                                  RecoverBody(State(b,P),realBody).
  reachable(Q)  :- reachable(R), trans(R,_,Q).

  Deadlock := reachable(State(st,L)),fail RecoverBody(State(st,L),_),
              fail trans(State(st,L),_,_), L != Skip.

  // Capturing tau-loops
  tauPath      ::= (iS:State,fS:State).
  tauPath(P,Q) :- trans (P,tau,Q).
  tauPath(P,Q) :- tauPath(P,S),tauPath(S,Q).
  Livelock := tauPath(L,L).

  // Nondeterminism property
  accepts    ::= (iS:State,ev:SigmaTauTock).
  accepts(P, ev) :- trans(P,ev,_), ev != tau.
  accepts(P, ev) :- trans(P,tau,R),accepts(R,ev).

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

Fig. 5. Capturing classical properties in FORMULA

It is worth noting that representing classical properties in FORMULA is almost a direct transcription from its definition. This is a result of reasoning with FORMULA and its language is similar to reasoning about First-Order Logic (Clark completion). A detailed discussion on how FORMULA rules are associated to First-Order Logic formulae by Clark completion is presented in [MF13a, MFDW14], including the derivation of FORMULA expressions for each classical property. Actually, FORMULA is a combination between 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. Moreover, as FORMULA is based on First-Order Logic and LTL is a subset of this logic, we can encode LTS in terms of FORMULA and perform LTL model checking. However, this is not in the scope of this paper. We restricted ourselves to perform those properties check simultaneously supported in FDR and PAT. For instance, PAT supports LTL model checking but FDR does not.

5. Using the model checker

To use our proposed model checker in practice, it is expected that the user can write specifications using CML syntax and get error messages in a user-friendly way. Thus, this section provides an overview about how the model checker presented in this article fits into the Symphony IDE: an open source tool supporting systematic engineering of SoSs using the CML.

5.1. The Symphony IDE tool

Symphony IDE is a tool built on top of the Eclipse platform, that integrates several features and auxiliary tools to provide support for dealing with SoSs. Detailed information about the Symphony IDE can be found in [CMDC+14]. Relevant material and the download link are available on the Symphony tool website.[10]

[10] http://symphonytool.org.

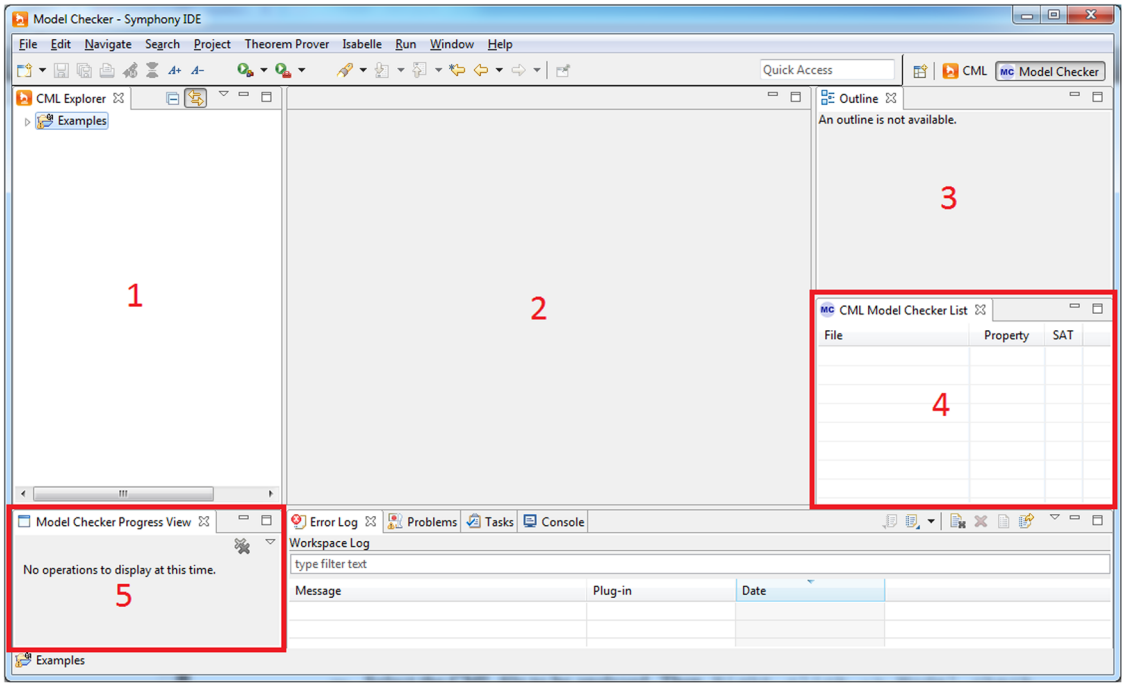

Fig. 6. CML model checker perspective

Broadly, the Symphony IDE integrates all of the available CML analysis functionality and provides editing abilities. The integration with Artisan Studio provides the ability to model SoSs using SysML. CML model files can be generated directly from SysML models and recognised within the Symphony tool automatically. SysML models may also have static fault analyses performed on them using the external HiP-HOPS tool [PWP+11].

The main Symphony tool contains many submodules [abstract syntax tree (AST), parser, type-checker and editor] and plug-ins (the larger grey boxes). Concerning external tools, Symphony IDE uses RT-Tester, the Isabelle theorem prover, the Microsoft FORMULA model checker and Maude. For each external tool there is a corresponding plug-in. Furthermore, the model checker plug-in also provides support for the fault tolerance plug-in to analyse fault tolerance properties of CML models.

The simulator plugin is capable of simulating CML models within the Symphony IDE with no need for external components, but it is also capable of co-simulating a model that does have external components. This is done via a set of libraries that are embedded into the external component, and which allow for communication with the simulator plug-in also used for incorporation of passive testing as well as external code gradually [LrNK14].

Furthermore, as the Symphony IDE is based on Eclipse platform, it provides a common look and feel to a large collection of extension products and the use of *views* and *perspectives*.

5.1.1. The model checker plugin

The model checker functionalities are available through the CML model checker perspective (see Fig. 6), which is composed by the Symphony Explorer (1), the CML Editor (2), the Outline view (3), the internal Web browser (visible when the user wish to see possible counterexamples) and two further specific views: the CML model checker list view (4) to show the overall result of the analysis and the model checker progress view (5) to show the execution progress of the analysis, which is invoked through the context menu when the CML or the MC perspectives are active.

The way the model checker instantiates data from infinite domains is configurable in Symphony through the menu `Window -> Preferences`. There is a section "model checker setup" (see Fig. 7) that provides a field containing the number of instances that the model checker will use for all infinite types used in any CML model. The default value is 1. The suitable value depends on the specification. For example, if the piece of CML code `ch?x -> Stop` is used, only and `ch` is a channel supporting an infinite type, only one instance is enough to detect the deadlock. However, if the CML code `a?x -> b?y -> if (a <> b) then ...` (`a` and `b` support infinite type) is used, the model checker needs two instances (at least) to perform the correct analysis.

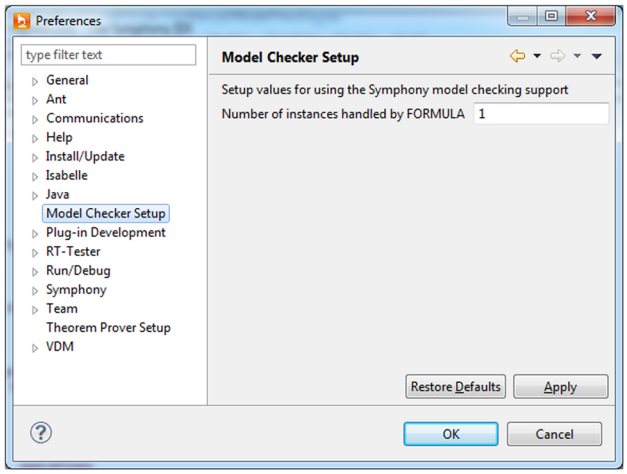

Fig. 7. CML model checker preferences

In general guidelines, the number of instances can be determined by looking at the necessary values to explore all branches of the analysed process.

Right click on the CML project (or file) to be analysed and select *Model check* → *Property to be checked*. The option Check MC Compatibility allows a previous check if the constructs used in the model are supported by the model checker. If some constructor is not supported by the model checker, the Symphony IDE shows a warning message (a popup) and the user can see more details by accessing the problems view.

When invoking the model checker, the result is shown in the model checker list view (✓ satisfiable or **X** for unsatisfiable models). The progress view shows the FORMULA invocation, where the user can stop the analysis by pressing the cancel button. For satisfiable models, the model checker plug-in provides a graph visualisation using the internal browser (by a double click in a specific item of the model checker list view component). The graph of the counterexample is generated by an internal component (graph builder) that obtains the output of FORMULA and uses is built by processing the output of FORMULA and algorithms considering the shortest path from the initial state to the state validating the property. Thus, although there might be other counterexamples, it shows the shortest one.

5.2. Examples and industrial case studies

The Symphony IDE provides some public examples that can be imported. This is achieved by a right click on the CML Explorer component and choosing the *Import* option. Then select the option *Symphony* → *Symphony examples*. The examples accepted by the model checker contain the suffix "`_MC`"in their names.

Simple bit register

The `BitRegister_MC` project contains the model of a bit register (a simple and didactic example), whose specification contains types, values and functions as follows.

```
values
  MAX : nat = 4
  increment : nat = 1

functions
 oflow : int*int -> bool
 oflow(i,j) == i+j > MAX

 uflow : int*int -> bool
 uflow(i,j) == i-j < 0
```

```
types
  MYINT = nat
   inv i == i in set {increment}

channels
   init, overflow, underflow
   inc, dec : MYINT
```

The MAX value is intended to limit the number of bits manipulated by the register; its value is 4. The increment value denotes the number to be used in increment or decrement operations. The functions uflow and oflow are useful to detect underflow and overflow, respectively. And the type MYINT is a natural number with a constraint (invariant) limiting the possible values to the set {1}.

In terms of channels, the bit register uses a init (to represent an event specific for initialisation purposes), inc and dec. The last two channels support the type MYINT and are used to communicate the value of increment.

Concerning the main description, the bit register is modelled as a process (RegisterProc) that contains a state, operations and actions. The state contains one variable (reg) that stores a natural number with default value 0. Three operations are defined to change the state variable: INIT simply assigns MAX−1 to reg, whereas ADD and SUB define sum and subtraction of a value (argument) over reg.

Concerning actions, the RegisterProc has an auxiliary action REG that is an external choice between two actions: one involving increments and other involving decrements. The first action offers an event on channel inc and then checks if an overflow occurs (before performing the actual increment on reg). If so, the process deadlocks; otherwise, the operation ADD is executed and the process recurses in action REG again. The second action is similar but offers an event on channel dec, checks for underflows and executes the operation SUB and recurses again (as long as underflow does not occurs). Finally, the main action of RegisterProc establishes that the process performs the init action and then behaves like the sequential composition INIT();REG, where the initialisation is executed and the system behaves as the action REG.

```
process RegisterProc =
 begin
  state
  reg : int := 0

  operations
   INIT : () ==> ()
   INIT() == reg := MAX - 1

   ADD: int ==> ()
   ADD(i) == reg := reg + i

   SUB: int ==> ()
   SUB(i) == reg := reg - i

  actions
   REG = (inc.increment -> [not oflow(reg,increment)] & ADD(increment);REG)
         []
         (dec.increment -> [not uflow(reg,increment)] & SUB(increment);REG)
   @ init -> INIT(); REG
 end
```

It is not difficult to see that the RegisterProc process can deadlock in two situations: after one increment (two events inc.1 are performed but only one ADD operation is executed) there is an overflow, or after MAX−1 decrements (MAX events dec.1 are performed but the operation SUB is executed MAX−1 times). Figure 8 shows the deadlock for the first situation (the shortest path to deadlock). If one changes the values of increment to 2, only one inc.2 event is performed before reaching a deadlock.

Handling infinite types

An example of simple CML processes handling infinite types in communications and predicates is also available in the project InfComm_MC. Although the processes are quite simple, they are useful to show the power of SMT solving inside the model checker to handle the infinite types. The processes used here are not analysable by FDR or PAT.

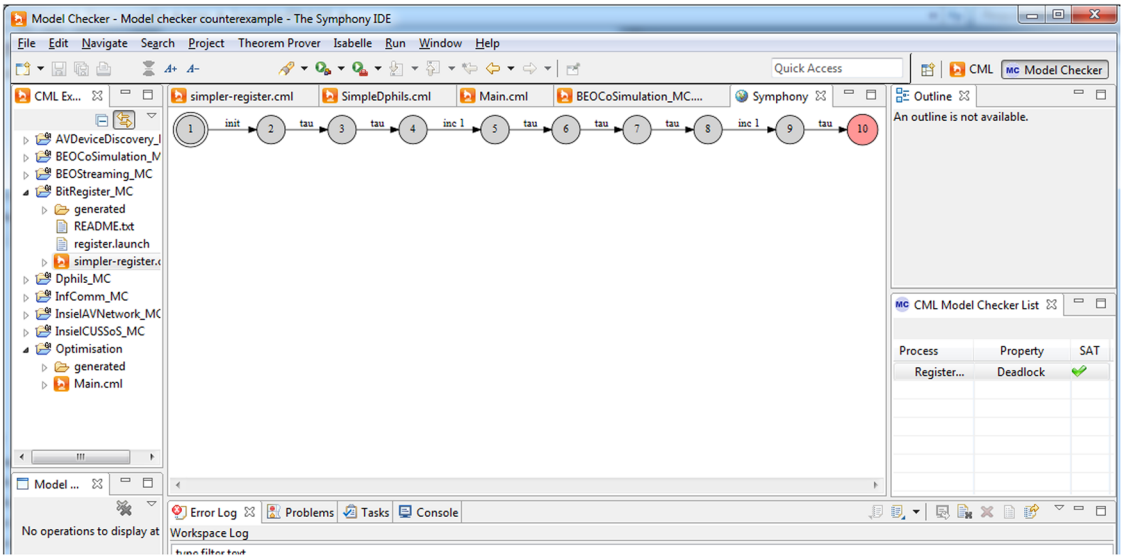

Fig. 8. Graph of bit register

The types manipulated by this example (`NATA` and `REALB`) are declared just as alias for natural and real numbers. The declared channels `a` and `b` respectively support the types `NATA` and `REALB`.

```
types
  NATA = nat
  REALB = real

channels
  a:NATA
  b:REALB
```

The first process has a state variable (`n`) of type `NATA` and it is initialised with `0`. It also declares an operation (`Double`) that receives a natural number and puts its double value in the state variable. The behaviour is described by the action `MAIN`: it performs the input event `a?x` and calls the operation `Double`. This puts `2*x` in the state variable `n`. Then the process receives another input (`a?y`) and checks if `n = y and y > 2` holds. If so, the process deadlocks; otherwise, it terminates successfully.

```
process P =
begin
  state
   n : NATA := 0
  operations
   Double:nat ==> ()
   Double(k) == n := 2*k
  actions
   MAIN = a?x -> Double(x); a?y -> if (n = y and y > 2) then Stop else Skip
 @ MAIN
end
```

The input events `a?x` and `a?y` correspond to an infinite family of events (channel `a` accepts any natural number). This is not allowed in FDR or PAT. Our model checker instantiates symbolic values and tries to find the deadlock. It is quite intuitive that two instances are necessary to validate the predicate `n = y and y > 2` and detect the deadlock. This information is put in the model checker setup preferences and the deadlock check is performed. After the analysis, the graph representation shows the values `2` and `4` as assigned to `x` and `y`, respectively. Indeed, they are the least natural numbers that satisfy the predicate `n = y and y > 2`. It is not clear in the documentation how Z3 finds these values internally. However, they just need to satisfy all predicates and constraint they are involved in.

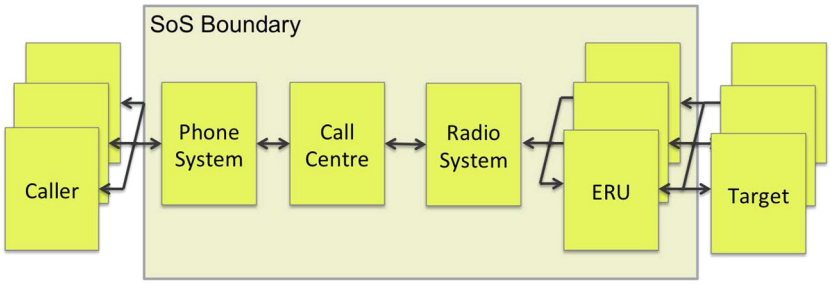

Fig. 9. Outline of the emergence response system

The second process is a variation of the previous one, but instead of terminating via `Skip`, it has the possibility of executing infinitely (by a recursion call). The process receives an input (`a?x1`) and calls the `Triple` operation. Then, it receives a real number as input (b?y1) and tests if the predicate `value = y1 and y1 > 2` holds. If it is true the process deadlocks; otherwise, the process repeats the same behaviour again.

```
process PRec =
begin
  state
   value : NATA := 0
  operations
   Triple:nat ==> ()
   Triple(v) == value := 3*v
  actions
   MAIN = a?x1 -> Triple(x1); b?y1 -> if (value = y1 and y1 > 2) then Stop else
        MAIN
 @ MAIN
end
```

This process is not analysable in FDR or PAT. The infiniteness of the input events disable the analysis by both tools. Furthermore, the use of real numbers in events disables the analysis in FDR. By manipulating only one instance of `NATA` and one of `REALB` our model checker was able to find de deadlock. In this case, the values found for `x1` and `y1` were `1` and `3`, respectively.

Industrial case study

Although the COMPASS project has several case studies, all examples chosen to be analysable by the model checker do not use communications and predicates involving data of infinite domain. We have selected the emergence response system (ERS) to evaluate the model checker because it has a detailed description of its requirements and due to its high complexity (as a SoSs). The ERS is a compound system introduced in [APR+13, ADP+13] (see its Outline view in Fig. 9). It is supplied by the Italian company Insiel, which supports an SoS in northern Italy incorporating separate emergency services (such as fire departments and ambulance services). The constituent systems are operationally and managerially independent (provided and developed by external organisations). They are the Phone System, Call Centre, Radio System and Emergency Response Unit (ERU). Each constituent may be geographically distributed and may evolve. The entities in the environment with which the SoS interacts include Callers and Targets. The purpose of this simplified Insiel SoS is to meet one high-level requirement: for every call received, send an ERU with correct equipment to the correct target. We extract the code that corresponds to the activation, detection and recovery of faults.

We add controller processes `ERUs_0`, `ERUs_1` to complete the entire behaviour of the Call Centre, controlling the number of ERUs currently allocated (see Fig. 10). They are similar but have a subtle difference in their operation `deallocate()`. The controller process `ERUs_0` has a state containing the number of allocated ERUs (`allocated_0`) and the total number of controlled ERUs (`total_erus_0`). Both variables have `0` as default values. The defined operations `allocate_0()` and `deallocate_0()` have the purpose of modifying the variable `allocated_0`, by maintaining or decreasing its value accordingly.

```
process ERUs_0 =
begin
  state
    allocated_0: int := 0
    total_erus_0: int := 0
  operations
    allocate_0: () ==> ()
    allocate_0() == allocated_0 := allocated_0
    deallocate_0: () ==> ()
    deallocate_0() == allocated_0 := allocated_0 - 1
  actions
    InitControl = allocated_0 := 0; total_erus_0 := 2
    Choose =  if allocated_0 = 0 then Allocate
              else if allocated_0 = total_erus_0 then Service
              else (Allocate [] Service)
    Allocate = allocate_idle_eru -> allocate(); Choose
    Service = service_rescue -> deallocate(); Choose
  @ InitControl; Choose
end
process ERUs_1 =
begin
  state
    allocated_1: int := 0
    total_erus_1: int := 0
  operations
    allocate_1: () ==> ()
    allocate_1() == allocated_1 := allocated_1 + 1
    deallocate_1: () ==> ()
    deallocate_1() == allocated_1 := allocated_1 - 1
  actions
    InitControl = allocated_1 := 0; total_erus_1 := 2
    Choose =  if allocated_1 = 0 then Allocate
              else if allocated_1 = total_erus_1 then Service
              else (Allocate [] Service)
    Allocate = allocate_idle_eru -> allocate_1(); Choose
    Service = service_rescue -> deallocate_1(); Choose
  @ InitControl; Choose
end
```

Fig. 10. CML specification of the ERUs

Concerning the behaviour, the process `ERUs_0` has four auxiliary actions: `InitControl` that simply initiates the state variables with convenient values; `Choose` that behaves like action `Allocate`, if there is no allocated ERU (`allocated_0 = 0`), `Service`, if all ERUs are already allocated (`allocated_0 = total_erus_0`) or like (`Allocate [] Service`), otherwise; `Allocate` that communicates the allocation of an available ERU (through event `allocate_idle_eru`), allocates it (by calling operation `allocate_0`) and then behaving like `Choose` again (by using sequential composition); and `Service` that communicates the execution of a rescue service (through event `service_rescue`), makes an ERU available (by calling operation `deallocate_0`) and then behaving like `Choose` again. The main behaviour of the `ERUs_0` process is defined by the sequential composition between `InitControl` and `Choose`. The process `ERUs_1` has almost the same description. It has its own state variables and operations. However, operation `allocate_1()` changes the corresponding state variable differently (Fig. 10).

The processes `InitiateRescueFault1Activation` and `Recovery1` (see Fig. 11) establish activities of fault rescue and recovery, respectively. They are useful to detect faults in the system and start a recovery procedure in order to guarantee the allocation of an ERU. The two versions of the entire ERS are described by `ERSystem_0` and `ERSystem_1`. In version `ERUs_0` the operation `allocate_0` should add `1` to the previous `allocated_0` value. This simple mistake causes a deadlock on process `ERSystem_0`—because channel `service_rescue` is never offered by `ERUs_0`—that is successfully detected by the model checker. On the other hand, the process `ERUs_1` fixes this problem and `ERSystem_1` in Fig. 11 is deadlock-free.

```
process InitiateRescueFault1Activation =
begin
  actions
    CallCentreStart = start_rescue -> FindIdleEru
    FindIdleEru = find_idle_erus -> (IdleEru [] wait -> FindIdleEru)
    IdleEru = allocate_idle_eru -> send_rescue_info_to_eru -> IR1
    IR1 = (process_message -> FAReceiveMessage) [] (fault_1_activation -> IR2)
    FAReceiveMessage = receive_message -> ServiceRescue
    ServiceRescue = service_rescue -> CallCentreStart
    IR2 = IR2Out [] (error_1_detection -> FAStartRecovery)
    IR2Out = drop_message -> target_not_attended -> CallCentreStart
    FAStartRecovery = start_recovery_1 -> end_recovery_1 -> ServiceRescue
  @ CallCentreStart
end
process Recovery1 =
begin
  actions
    Recovery1Start = start_recovery_1 -> process_message ->
      receive_message -> end_recovery_1 -> Recovery1Start
  @ Recovery1Start
end
process ERSystem_0 =  InitiateRescueFault1Activation [| ERUsSignals |] ERUs_0
process ERSystem_1 =  InitiateRescueFault1Activation [| ERUsSignals |] ERUs_1
```

Fig. 11. The fault rescue and recovery activities of ERS

6. Related work

The efforts toward increasing the power of CSP verification have focused on auxiliary techniques to go beyond FDR capabilities. The use of SAT-solving, for example, as an auxiliary technique has been investigated in [POR12], where a prototype called SymFDR implements a bounded model checker for CSP. The authors compare with the FDR tool to show that SymFDR can deal with problems beyond FDR, such as combinatorial complex problems. SymFDR uses FDR as expansion engine but deals with specific SAT-based analysis. For some problems FDR outperforms SymFDR and for other problems the reverse occurs. In [POR12] the authors attempts at explaining why such situations happen. For our context, what matters is that although SymFDR is based on SAT, it can only handle finite state space systems because it reuses the LTS representation of CSP processes provided by FDR.

The approach proposed in [Leu01] is strongly related to ours and consists of an implementation of the CSP language based on SICStus Prolog (a variation of Prolog). Its main goal is to provide a CSP interpreter and animator (instead of a model checker). According to the authors, with a little effort, their solution could be combined with an LTL model checker (e.g. SPIN) to 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 we focus on model checking instead of interpretation. Nevertheless, the LTS generated by FORMULA in our approach is also used to show the (symbolic) execution of the process (the entire graph). Furthermore, as our model checker can handle infinite state systems, we indeed concretise a future work pointed out in [Leu01].

The ProB tool [LB03] provides animation and model checking support for the B method [Abr96]. In its internal structure, ProB contains several modules: front-end, B-kernel, interpreter, animator, state-based model checker and temporal model checker. The last two modules enable ProB to support temporal model checking and constraint-based model checking. Furthermore, the tool provides verification facilities for CSP_M such as refinement checking, LTL model checking, and CTL model checking as well. Hence, CSP_M specifications can be analysed and verified in ProB by means of refinement checking and classical model checking (using temporal properties). A component of ProB is used (particularly the solver) in Symphony (by the CML interpreter) to solve constraints related to the data aspects of CML.[11] Although ProB is a sophisticated debugging tool even for very large machines (non-exhaustive exploration of the state space and capability of finding potential problems are strong points of ProB [LB03]), it can handle only finite sets and variables with finite range of values. In our model checker, the use of symbolic values and SMT solving also allows one to handle data with infinite domain in communications or predicates (using any variable). Then, Z3 provides these concrete values automatically. Furthermore, our model checker is able to cope with time aspects as well. Nevertheless, we consider ProB a strong candidate to be used as underlying framework, instead of FORMULA, in our approach (as pointed out as future work in Sect. 7).

[11] This use is indeed present in the VDM interpreter of the Overture tool, which is reused by the CML interpreter included in Symphony.

A model checker for CSP_M has been proposed in [LF08]. It considers a subset of CSP_M and has the advantage of being FDR-compliant. Thus, FDR has been used to validate the output produced by the tool. Under the point of view of embedding, this work goes beyond [Leu01] by also providing verification capabilities. The tool deals with some issues in a more efficient way than FDR does (the use of lazy evaluation, for example). However, it does not support time aspects (this is pointed out as future work).

The approach presented in [DHSZ06] is another interesting work similar to ours. It allows verification of properties and refinement between Timed CSP specifications by using CLP [JL87]. The approach starts by encoding the semantics (operational and denotational) of Timed CSP in CLP. Although this allows a systematic translation, no tool support was presented to allow the user to handle Timed CSP directly (and derive the CLP code automatically). The approach is able to cope with state-based specifications (even with more complex data) and also allows refinement checking via CLP. However, it is not able to handle communications involving data from infinite domains.[12] In this sense, our work goes beyond by using the power of STM solving to also handle communications and predicates involving (not complex for the moment) data with infinite domains.

The idea of using SMT-solving for model checking purpose is not new either, mainly because the advances of SMT solvers bring a new level of verification. For example, the approach proposed in [BMR12] extends the SMT-LIB to describe rules and declare recursive predicates, which can be used by symbolic model checking. Moreover, that work investigates the strong similarity between property verification and reachability analysis. We use this result to encode queries in FORMULA as reachability questions. Another approach is presented in [ABG+13] proposes an SMT-based specification language to improve the verification of safety properties. Our work, on the other hand, brings a new perspective for reasoning about infinite systems by using a high level specification language; we maintain CSP as the specification language and provide automatic translation from specification to FORMULA code.[13] Our work differs from that of [ABG+13] by using an SMT-solving to increase the expressiveness of CSP and provides a powerful tool for verification and reasoning of programs.

There is also a similar approach proposed in [Ver00] that uses MAUDE for executing and verifying concurrent communicating systems (CCS). According to that work, only behavioural aspects can be handled, whereas we handle data aspects even if they come from an infinite domain and are involved in communications and in predicates. Moreover, that work also considers temporal logic, whereas we do not (it is not a CSP culture but FORMULA can handle it). We point out that MAUDE can be more powerful than FORMULA but it can be harder to guarantee convergence when applying rewriting rules. Our work is free of convergence problems because the engine of FORMULA focuses on finding the least fixed-point using SMT solving.

7. Conclusions and future work

This paper has presented a systematic way for creating a model checker from the syntax and from the SOS of the CML language. We have used the Microsoft FORMULA framework as the core engine for the analysis. The FORMULA tool has two main advantages: (i) its language is fully declarative and allows one to create high-level implementations (or abstractions); (ii) its integration with the Z3 SMT-based solver allows one to create model checkers that can deal with infinite aspects whose underlying logics is decidable. We illustrated our strategy considering the process algebra CSP, which is widely used for specifying concurrent systems.

The ideal way to capture all semantic aspects of the language is by using a deep embedding. However, due to the degradation of performance and the available infrastructure for data types provided by FORMULA, we proposed a hybrid embedding approach that captures data aspects directly in terms of FORMULA (exhibiting a shallow semantics embedding) and fully interpreting the behavioural aspects (following a deep semantics embedding). This allows us to create a semantics preserving model checker. Of course, the one-to-one mapping of the CML semantics used in our deep embedding is a strong evidence of semantics preservation (not a formal proof). However, this absence of formal proof also occurs with other works in the literature (like [Leu01], for example) where the underlying framework Prolog does not have a public available formal semantics. Our hypothesis is that FORMULA is sound with respect to its intended semantics. To attest the evidence of semantic preservation, we performed a test suite containing about 170 examples (property verification and traces refinement). The examples were focused on capturing the essence of the semantics with its subtleties (mainly with complex constructs like parallelism, hiding, sequential composition and recursion). For those that could also be analysed in FDR and PAT we obtained the same results.

[12] The authors mention nothing about the infiniteness of communication data, but present an example where input communications are restricted.

[13] The implementation of a tool that converts a CSP specification into a FORMULA script is a still in progress and is not available yet.

To provide a real integration of FORMULA in the CML development platform—Symphony IDE—we have developed a plugin that provides a user friendly way to automatically analyse CML specifications. The advantage of making our model checker as part of the Symphony tool has the advantage of integration with other tools (plugins) that allow other features for CML like parsing, animation, debugging, proof obligation generation, discharging proof obligations (via theorem proving), test generation, fault tolerance analysis, refinement analysis, graph visualisation, etc.

As our focus was in correctness[14] and handling communications and predicates with infinite data, the comparison (in terms of performance) of our model checker with FDR and PAT is an intended topic for future work. Indeed, good performance is desirable for model checkers. Our preliminary results indicate that our model checker can outperform FDR and PAT for data-intensive communicating systems, that is, systems using big sets of values involved in communication and in predicates. This is rather reasonable because FORMULA manipulates the LTS differently: generation on demand and interaction with SMT-solving. Regarding performance comparison, we already have some results regarding efficiency, but they are still inconclusive. Concerning time to produce a model checker, our experience shows that the creation with FORMULA is smaller, however, this still depends on the user experience. We also intend to generalise our model checker creation technique to consider only two formal descriptions: syntax and semantics, expressed in a domain specific language (DSL). This will allow us to provide automatic generation of model checkers for languages whose syntax and SOS are specified in such a DSL.[15]

We also intend to provide an embedding for temporal logic in FORMULA; this allows specifying other properties rather than the classical properties addressed in this work (deadlock, livelock and nondeterminism). Although in CSP, any property verification is performed by refinement, the direct analysis of a property avoids using the refinement checking approach in FORMULA (where two LTSs are instantiated).

We are currently implementing a model checker for Circus using the K framework [RŞ10]. It is an executable semantic framework (a rewriting system) in which programming languages, calculi, as well as type systems or formal analysis tools can be defined, making use of configurations, computations and rules. We also intend to compare the CML model checker with the K model checker, FDR and PAT together to produce a more general guide to help decisions based on performance, class of problems, aspects (behaviour, data, time, etc.), available platforms and scalability.

Another interesting topic for future work is the manipulation of specifications involving processes that originate an infinite LTS (like for example the process `P(k) = ··· P(k+1)`) by using data abstraction [DFM09, FMS08], induction [BK08], and optimization techniques like OBDD [Bry92] and partial order reduction [God91, Pel93, BK08]. This will certainly represent an important step towards overcoming the state explosion problem.

Finally, the use of ProB in Symphony has revealed a point that deserves deeper investigation: the use of ProB as underlying framework for other model checkers. Indeed, as the ProB interpreter is written in a SOS style, it fits well to the ideas of the systematic embedding used here. Moreover, there are a lot of components ready to use and material about ProB (available at http://stups.hhu.de/ProB/).

Acknowledgments

The work presented here is partly supported by the EU Framework 7 Integrated Project: COMPASS, Grant Agreement 287829. The authors would also like to thank all the other participants involved in the COMPASS project for their collaboration. This work was also partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, Grants 573964/2008-4 and APQ-1037-1.03/08.

References

[ABG+13] Alberti F, Bruttomesso R, Ghilardi S, Ranise S, Sharygina N (2013) Reachability modulo theory library (extended abstract). In: Fontaine P, Goel A (eds) SMT 2012. EPiC Series, vol 20. EasyChair, pp 67–76

[Abr96] Abrial JR (1996) The B-book. Cambridge University Press, Cambridge

[14] The results of our model checker, FDR and PAT where the same for all examples used in our test campaign.

[15] There is a DSL for this purpose. However we are investigating if its expressiveness is enough for describing any SOS.

[ADP+13] Andrews Z, Didier A, Payne R, Ingram C, Holt J, Perry S, Oliveira M, Woodcock J, Mota A, Romanovsky A (2013) Report on timed fault tree analysis—fault modelling. Technical report D24.2, COMPASS

[APR+13] Andrews Z, Payne R, Romanovsky A, Didier A, Mota A (2013) Model-based development of fault tolerant systems of systems. In: IEEE international systems conference (SysCon), pp 356–363. doi:10.1109/SysCon.2013.6549906

[BGW12] Bryans J, Galloway A, Woodcock J (2012) CML definition 1. Technical report, COMPASS deliverable, D23.2

[BK08] Baier C, Katoen J-P (2008) Principles of model checking (representation and mind series). The MIT Press, USA

[BMR12] Bjørner N, McMillan K, Rybalchenko A (2012) Program verification as satisfiability modulo theories. In: Fontaine P, Goel A (eds) SMT@IJCAR. EPiC Series, vol 20. EasyChair, pp 3–11

[Bry92] Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput Surv 24(3):293–318. doi:10.1145/136035.136043

[CGL94] Clarke E, Grumberg O, Long D (1994) Model checking and abstraction. ACM Trans Program Lang Syst 16(5):1512–1542

[CMDC+14] Coleman JW, Malmos AK, Couto LD, Larsen PG, Payne R, Foster S, Schulze U, Cajueiro A (2014) Fourth release of the COMPASS tool—Symphony IDE user manual. Technical report D31.4a, COMPASS deliverable, D31.4a

[CML+12] Coleman JW, Malmos AK, Larsen PG, Peleska J, Hains R, Andrews Z, Payne R, Foster S, Miyazawa A, Bertolini C, Didier A (2012) COMPASS tool vision for a system of systems collaborative development environment. In: IEEE SoSE, pp 451–456

[CW13] Cavalcanti A, Woodcock J (2013) CML definition 3—Timed Operational semantics. Technical report D23.4b, COMPASS deliverable, D23.4b. http://www.compass-research.eu/Project/Deliverables/D234b.pdf. Accessed 24 Aug 2015

[DFM09] Damasceno A, Farias A, Mota A (2009) A mechanized strategy for safe abstraction of CSP specifications. In: SBMF, pp 118–133. doi:10.1007/978-3-642-10452-7_9

[DHSZ06] Dong JS, Hao P, Sun J, Zhang X (2006) A reasoning method for timed CSP based on constraint solving. In: Zhiming L, Jifeng H (eds) Formal methods and software engineering. Lecture notes in computer science, vol 4260. Springer, Berlin, pp 342–359

[DMB08] De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Tools and algorithms for the construction and analysis of systems. LNCS, vol 4963. Springer, Berlin, pp 337–340

[dMB11] De Moura L, Bjørner N (2011) Satisfiability modulo theories: introduction and applications. Commun ACM 54(9):69–77

[FLV08] Fitzgerald JS, Larsen PG, Verhoef M (2008) Vienna development method. In: Wah B (ed) Wiley encyclopedia of computer science and engineering. Wiley, New York

[FLW14] Fitzgerald J, Larsen P, Woodcock J (2014) Foundations for model-based engineering of systems of systems. In: Aiguier M (eds) Complex systems design and management. Springer, Berlin, pp 1–19

[FMS08] Farias A, Mota AC, Sampaio A (2008) Compositional abstraction of CSPZ processes. J Braz Comput Soc 14(2):23–44

[GM99] Gallardo MDM, Merino P (1999) A framework for automatic construction of abstract promela models. In: Dams D, Gerth R, Leue S, Massink M (eds) Theoretical and practical aspects of SPIN model checking. Lecture notes in computer science, vol 1680. Springer, Berlin, pp 184–199

[God91] Godefroid P (1991) Using partial orders to improve automatic verification methods. In: Proceedings of the 2nd international workshop on computer aided verification (CAV'90), London. Springer, Berlin, pp 176–185

[HH98] Hoare CAR, He J (1998) Unifying theories of programming. In: Hoare CAR, Bird R (eds) Series in computer science. Prentice Hall, USA

[JKD+10] Jackson EK, Kang E, Dahlweid M, Seifert D, Santen T (2010) Components, platforms and possibilities: towards generic automation for MDA. In: EMSOFT, pp 39–48

[JL87] Jaffar J, Lassez J-L (1987) Constraint logic programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'87). ACM, New York, pp 111–119

[JLB11] Jackson EK, Levendovszky T, Balasubramanian D (2011) Reasoning about metamodeling with formal specifications and automatic proofs. In: MoDELS, pp 653–667

[JSD+09] Jackson EK, Seifert D, Dahlweid M, Santen T, Bjørner N, Schulte W (2009) Specifying and composing non-functional requirements in model-based development. In: Bergel A, Fabry J (eds) Software composition. Lecture notes in computer science, vol 5634. Springer, Berlin, pp 72–89. doi:10.1007/978-3-642-02655-3_7

[LB03] Leuschel M, Butler M (2003) ProB: A model checker for B. In: Araki K, Stefania G, Mandrioli D (eds) FME 2003: formal methods. LNCS, vol 2805. Springer, Berlin, pp 855–874

[Lee85] Lee S (1985) On executable models for rule-based prototyping. In: Proceedings of the 8th international conference on software engineering (ICSE'85). IEEE Computer Society Press, pp 210–215

[Leu01] Leuschel M (2001) Design and implementation of the high-level specification language CSP(LP). In: PADL. LNCS, vol 1990. Springer, Berlin, pp 14–28

[LF08] Leuschel M, Fontaine M (2008) Probing the depths of CSP-M: a new FDR-compliant validation tool. In: Liu S, Maibaum T, Araki K (eds) Formal methods and software engineering. Lecture notes in computer science, vol 5256. Springer, Berlin, pp 278–297

[LrNK14] Lausdahl K, Nielsen CB, Kristensen K (2014) Including running system implementations in the simulation of system of systems models. In: Formal methods: foundations and applications. Lecture notes in computer science. Springer, Berlin

[LSD11] Liu Y, Sun J, Dong JS (2011) PAT 3: an extensible architecture for building multi-domain model checkers. In: Dohi T, Cukic B (eds) IEEE 22nd international symposium on software reliability engineering (ISSRE'11). IEEE, pp 190–199

[MF13a] Mota A, Farias A (2013) COMPASS tool—model checking support. Technical report, COMPASS deliverable, D33.1. http://www.compass-research.eu/Project/Deliverables/D331.pdf. Accessed 24 Aug 2015

[MF13b] Mota A, Farias A (2013) Implementing an SMT-based model checker for CSP from its operational semantics. In: Proceedings of the 16th Brasilian symposium on formal methods (SBMF'13), pp 36–41

[MF14] Mota A, Farias A (2014) A rapid approach for building a semantically well founded *circus* model checker. In: Proceedings of the XXI tools session of the Brazilian conference on software (CBSoft'14), pp 77–84

[MFDW14] Mota A, Farias A, Didier A, Woodcock J (2014) Rapid prototyping of a semantically well founded circus model checker. In: Giannakopoulou D, Salaün G (eds) Software engineering and formal methods. Lecture notes in computer science, vol 8702. Springer International Publishing, Berlin, pp 235–249. doi:10.1007/978-3-319-10431-7_17

[Pel93] Peled D (1993) All from one, one for all: on model checking using representatives. In: Courcoubetis C (ed) Computer aided verification. Lecture notes in computer science, vol 697. Springer, Berlin, pp 409–423
[Plo04] Plotkin GD (2004) A structural approach to operational semantics. J Log Algebr Program 60–61:17–139
[POR12] Palikareva H, Ouaknine J, Roscoe AW (2012) SAT-solving in CSP trace refinement. Sci Comput Program 77(10–11):1178–1197
[PWP+11] Papadopoulos Y, Walker M, Parker D, Rüde E, Hamann R, Uhlig A, Grätz U, Lien R (2011) Engineering failure analysis and design optimisation with HiP-HOPS. Eng Fail Anal 18:590–608
[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice Hall, USA. The text book teaching material can be found at http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf. Accessed 24 Aug 2015
[Ros10] Roscoe AW (2010) Understanding Concurrent Systems. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition
[RŞ10] Roşu G, Şerbănuţă TF (2010) An overview of the K semantic framework. J Log Algebraic Program 79(6):397–434
[SLDC09] Sun J, Liu Y, Dong JS, Chen C (2009) Integrating specification and programs for system modeling and verification. In: Chin WN Qin S (eds) TASE 2009, third IEEE international symposium on theoretical aspects of software engineering. IEEE Computer Society, Tianjin, pp 127–135
[ST02] Schneider S, Treharne H (2002) Communicating B machines. In: Bert D, Bowen JP, Henson MC, Robinson K (eds) ZB 2002: formal specification and development in Z and B. Lecture notes in computer science, vol 2272. Springer, Berlin, pp 416–435
[Ver00] Verdejo A (2000) Implementing and verifying CCS in MAUDE. In: The international federation for information processing (IFIP)
[WC02] Woodcock J, Cavalcanti A (2002) 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). Springer, London, pp 184–203
[WCF05] Woodcock JCP, Cavalcanti ALC, Freitas L (2005) Operational semantics for model checking circus. In: Fitzgerald J, Hayes IJ, Tarlecki A (eds) FM 2005: formal methods, international symposium of formal methods Europe. Lecture notes in computer science, vol 3582. Springer, Berlin, pp 237–252
[WCF+12] Woodcock J, Cavalcanti A, Fitzgerald J, Larsen P, Miyazawa A, Perry S (2012) Features of CML: a formal modelling language for systems of systems. In: IEEE SoSE
[WM12] Woodcock J, Miyazawa A (2012) CML definition 0. Public document. Deliverable Number: D23.1, Version: 1.0, COMPASS Project, University of York
[ZCW11] Zeyda F, Cavalcanti A, Wellings AJ (2011) The safety-critical Java mission model: a formal account. In: ICFEM, pp 49–65

Received 5 November 2014
Accepted in revised form 11 August 2015 by Michael Butler and Cliff Jones
Published online 7 September 2015