Model checking CML: Tool development and industrial applications

A. Mota\(^1\) and A. Farias\(^2\) and J. Woodcock\(^3\) and P.G. Larsen\(^4\)

\(^1\)Federal University of Pernambuco, Brazil
\(^2\)Federal University of Campina Grande, Brazil
\(^3\)University of York, United Kingdom
\(^4\)Aarhus University, Denmark

Abstract. A model checker is an automatic tool that traverses the transition system (LTS) representation \(M\) of a given specification \(S\) (that belongs to a formal language \(L\)) to analyse the satisfaction of some (temporal) logical property \(f\); formally \(M \models f\). In practice, however, model checkers are (in general) implementations of very optimised algorithms, aiming at achieving the best time and space complexities, without a clear association to the semantics of \(L\). In this paper we show how to create a semantically guided model checker for \(L\), given its operational semantics, following a hybrid semantics embedding. We use the framework FORMULA to this end and illustrate our strategy considering the formal 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 SMT solving, our model checker can handle infinite data communications and predicates 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 industrial case studies.

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 \(L\).
Although a model checker should obey the Structured Operational Semantics (SOS) of a formal language $L$ \cite{Plo81}, it is created, in general, as any other software. That is, it can be error-prone itself. However, if its development process is strongly associated to the SOS of $L$, one can obtain a model checker for $L$ correct by construction with respect to the correctness of the underlying implementation infrastructure.

In this paper we propose a new model checker because (1) the language we use (CML \cite{WCF12}) is new and (2) it is not only better than existing model checkers, in the sense of trustworthiness, but the proposed model checker cannot be created by some reuse of previous model checkers \cite{MS01,DSL13}, without losing desired properties (for instance, dealing with symbolic state machines).

As a natural consequence of (1), this work is one of several efforts in the context of the COMPASS project \cite{FLW14}. It is implemented as a feature of the Symphony tool platform \cite{CML+12}. 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 CML (the COMPASS Modelling Language), a semantic combination among CSP, VDM and refinement calculus, whose purpose is to provide support for modelling Systems of Systems (SoS).

Concerning (2), consider the following simple CML process.

```plaintext
channels a: int
P = a?x -> a?y ->
   if x <> y and x*x > 9
     then Stop
   else P
```

The process $P$ receives two integers $x$ and $y$ on channel $a$. After that, if these values are different ($x <> y$) and $x^2 > 9$ the system deadlocks; otherwise, it behaves as $P$ again. Process $P$ cannot be handled by traditional CSP model checkers, such as FDR \cite{Ros10} or PAT \cite{LSD11}, because it can accept an infinite possibility of integers ($\text{int}$) as well as the predicate $x <> y$ is completely unbounded. 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 $x^2 > 9$ 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 \cite{Leu01} and \cite{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 FORMULA \cite{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.

As we adopt the generic framework FORMULA \cite{JLB11}, which has a declarative language front-end similar to Prolog but with a back-end that works in terms of the Z3 SMT solver \cite{dMB11}, we can build symbolic model checkers able to solving infinite data communications and predicates.

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 \cite{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 \cite{WCF05}. Afterwards, we have introduced the concepts of VDM in this embedding and added time aspects according to the operational semantics of CML.

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.

Finally we employ our CML model checker on case studies provided by COMPASS industrial partners: Insiel\textsuperscript{2} and Bang & Olufsen\textsuperscript{3}.

The main contributions of this paper are:

---

1 More information available at: http://symphonytool.org/
2 http://www.insiel.eu/
3 http://www.bang-olufsen.com/en
Model checking CML: Tool development and industrial applications

- 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 industrial case studies.

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 Section 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 Section 6 and our conclusions and future work in Section 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 (Formal Modeling Using Logic Programming and Analysis)—to build and analyse a CML specification.

FORMULA [JLB11] is a modern framework that follows the principles of Model-Based Development (MBD) and is based on Algebraic Data Types (ADTs) and strongly-typed Constraint Logic Programming (CLP). It supports concise specifications of abstractions (in a Prolog-like style) and model transformations. 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;
  - 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 \( LHS \leftarrow RHS \), 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). Figure 1 illustrates the work performed by FORMULA in an analysis. It takes the main goal (a **conforms** clause) and the facts given in a (partial) model as starting point. From the (initial) base of facts and the RHS of domain rules, FORMULA tries to generate other facts iteratively (according
Fig. 1. Iterative analysis of FORMULA.

Fig. 2. FORMULA snapshot model analysis

to the LHS of domain rules) until the base stops increasing. Afterwards, it checks the goal. If any SMT-solving activity (instantiation, evaluation, etc.) is required, FORMULA invokes Z3 automatically [JKD+10].

We illustrate the work of FORMULA using an example that captures the essence of a basic digraph (see Figure 2).

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(y))). 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 Systems of Systems (SoSs). It is founded on the well-established formalisms Circus \cite{WC02} and VDM \cite{FLV08}, and combines a number of aspects required in SoS modelling such as discrete time, concurrency, processes, state and contracts \cite{WCF12}.

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 \cite{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.

```plaintext
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 / [ expression ] \ action  (External choice)
    | action \ chanset expression  (Hiding)
    | parallel actions
    | control statements ;
```

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

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

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

The primitive actions `Skip` and `Stop`, respectively, mean immediate termination and immediate deadlock (allowing only time to pass). 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 `communication => P` offers the communication event `communication` 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 envi-
ronment (that is, the context outside ‘P’ and ‘Q’ decides which of ‘P’ or ‘Q’ should evolve). The timed interrupt action
‘P / t \ Q’ means the process behaves as ‘P’ for ‘t’ units of time, then it behaves as ‘Q’. If ‘t’ is not specified, the
action ‘P /__\ Q’ behaves as ‘P’ until ‘Q’ takes control (at any time), and the action behaves as ‘Q’. The timed timeout action
‘P [_> Q’ behaves as ‘P’ for ‘t’ units of time, then it behaves as ‘Q’. If ‘t’ is not specified, the
action ‘P \_\ s’ behaves as ‘P’ but hides the events in the set ‘s’. The interleave ‘P ||| Q’ means the parallel
evaluation (with no synchronisation) of ‘P’ and ‘Q’. On the other hand, the generalised parallelism ‘P [icsi] 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”.

2.2.2. Operational Semantics of CML

The semantics of CML has been originally defined in terms of Hoare & He’s Unifying Theories of Programming
(UTP) [HH98]. This CML UTP semantics was formally rewritten [CW13] in terms of an Structured Operational
Semantics following a Plotkin-style [Plo81]. We present this SOS semantics and use it to be closer to the way 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 de-
cscription 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). Its generic
representation is given by

(s, P) → (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. Of course, if P is deterministic, then there will be a unique
outcome; but if P is nondeterministic, then there may be many different outcomes, of which (t, Q) may represent one
or more, but not necessarily all. We hereafter assume basic knowledge in UTP and Operational Semantics.

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

---

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].
6 For a complete understanding of all rules please refer to [CW13].
Fig. 3. Firing rules for CML

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 reaches some state (different from successful termination) from which it goes nowhere;
- **Livelock** - a process has a livelock if it performs a τ-loop (a loop of infinitely many internal or τ-transitions);
- **Nondeterminism** - a process is nondeterministic if it has a livelock or if it decides to accept or reject a same event.
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].

3.1. Capturing CML constructs in FORMULA

The work towards model checking assumes that $M$ 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, such as LTSA [MK99] or Petri Nets [Mur89], the model $M$ is easily achievable and (usually) is correct. Nevertheless, for languages such as CSP [Ros10], PROMELA [GM99] and Circus [WCF05], creating a model checker by a direct programming approach can be too error-prone. In practice, most model checkers only create $M$ from $L$ using some black-box implementation susceptible to programming errors. However, if the model $M$ is systematically created from the SOS of $L$ (that is, $M \in SOS\{L\}$) the model checker becomes a semantics-preserving model checker for $L$ relative to the semantics of the framework used to encode the SOS of $L$. 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.

3.1.1. Basic Shallow Embedding in FORMULA

We propose a way to capture the structured operational semantics 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 ($\text{Integer}$, $\text{Natural}$, $\text{Real}$ and $\text{String}$), more complex types (like sets, relations, functions, sequences, bags, etc.) are absent and the mapping is not so direct. The domain $\text{AuxiliaryDefinitions}$ (see Figure 4) has the purpose of modelling types (basic types and complex) and operations over them.

The type $\text{VOID}$ (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 ($\text{SeqType}$ in line 6) that is the union of two types (inductively defining a sequence). The constructor $\text{EmptySeq}$ defines an empty sequence and $\text{NonEmptySeq}$ 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 $\text{ProductType}$ 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 using union types (line 19) including basic types ($\text{Integer}$, $\text{Natural}$, $\text{Real}$, $\text{String}$), complex types ($\text{SetType}$, $\text{SeqType}$), product type ($\text{ProdType}$) 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 (line 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.
AuxiliaryDefinitions {
  // Types
  VOID ::= {void}.
  EmptySeq ::= {emptySeq}.
  primitive NonEmptySeq ::= {first: Type, rest: SeqType}.
 (SeqType ::= EmptySeq + NonEmptySeq.
  EmptySet ::= {emptySet}.
  NonEmptySet ::= {head:Type, tail:SetType}.
  SetType ::= EmptySet + NonEmptySet.
  // definitions of operations over sets and sequences
  // A representation for product type
  primitive ProdType ::= (first: Type, second: Type).
  // User defined types
  primitive IntegerW ::= (Integer).
  primitive NaturalW ::= (Natural).
  primitive RealW ::= (Real).
  primitive BooleanW ::= (Boolean).
  primitive StringW ::= (String).
  // Wrappers for user defined types
  primitive ParamW ::= (name: String, value:Type).
  // Wrappers for parameters
  // Bindings
  NullBind ::= {nullBind}.
  primitive SingleBind ::= (name: String, value:Type).
  primitive BBinding ::= (b: SingleBind, rest: Binding).
  Binding ::= NullBind + BBinding.
  // operations over bindings
  guardDef ::= (id: Natural, st: Binding).
  guardNDef ::= (id: Natural, st: Binding).
}

Fig. 4. Representations for types and operations over them in FORMULA

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.

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.

3.1.2. 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 former defines the structures (building blocks) necessary to represent CML constructs for events and processes. Figure 5 illustrates the FORMULA representation for some CML constructs presented in Section 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 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.
domain CMLSyntax extends AuxiliaryDefinitions {
  //Useful definitions related to communications
  Channel ::= (chn : String, chT: Type).
  primitive CommEv ::= (chnName: String, chExp: String, val: Type).
  primitive I2OComm ::= (id: Natural, chName: String, chExp: String, val: Type).
  I2OCommDef ::= (id: Natural, val: Type, st: Binding, st_: Binding).
  Sigma ::= CommEv + I2OComm.
  Tau ::= [tau]. //Internal event
  Tock ::= [tock]. //Time event
  SigmaTauTock ::= Sigma + Tau + Tock. //All possible events

  //Actions and Processes
  BasicProcess ::= {Stop, Skip, Div}.
  primitive Prefix ::= (ev : Sigma, proc : CMLProcess). //Prefix
  //Timed interrupt. Useful to model Wait action. Wait N = Stop [_ N > Skip
  //A special operator of CML. The usual external choice is transformed into it
  //Conditional choice. Guarded choice are mapped to this constructor
  //Sequential composition
  primitive hide ::= (proc : CMLProcess, hideS : String). //Hiding

  //Generalised parallelism

  primitive proc ::= (name : String, param:Type). //Process call
  //All possible actions and processes
  CML Process ::= BasicProcess + Prefix + iChoice + eChoice + extraChoice +
    condChoice + seqC + hide + par + genPar + proc + var + let +
    operation + assign + intrpt + uTimeout + tIntrpt + tTimeout.

  //Action/process definition
  ProcDef ::= (name: String, params:Type, proc: CMLProcess).

  //Constructor related to synchronisation
  lieIn ::= (ev : Sigma, sourceSet: String).
}

Fig. 5. 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 Figure 3), where the usual external choice ([]) is transformed 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\textsuperscript{7}. The constructor for

\textsuperscript{7} Design decision due to performance.
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 47), 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 Figure 3), 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 CMLProcess defines (syntactically) all possible processes.

Finally, the constructor ProcDef (line 44) captures a process definition containing a name, its parameters and the corresponding body.

3.1.3. 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.

```prolog
\text{domain} \text{CMLSemantics} \text{extends} \text{CMLSyntax} \{ \\
\text{State} ::= (b: Binding, p: CMLProcess). \\
\text{trans} ::= (source: State, ev: SigmaTauTock, target: State). \\
\text{primitive Clock} ::= (Natural). \\
\text{primitive GivenProc} ::= (name: String). \\
\text{...} \\
\}
```

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.

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.

Below the representation of each firing rule for CML in terms of FORMULA transitions and states is shown.

No Transition

The actions Stop and Skip do not have associated transitions because they represent a final state (with no outgoing transition).

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

\[ \text{State(st, body)} :- \text{trans(S, ev, State(st, body))}. \]

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

\[
\text{trans}(iS, \tau, iS) :- iS \text{ is } \text{State}(st, \text{Div}).
\]

Prefix

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

\[
\text{State}(st, P), \\
\text{trans}(\text{ini}, \text{CommEv}(\text{chName}, \text{chExp}, \text{chType}), \text{State}(st, P)) :- \\
\text{ini is } \text{State}(st, \text{Prefix}(	ext{IOComm(id,chName,chExp,chType),P})), \\
\text{IOCommDef(id,chType,st,st_)}.
\]

As long as there is a state where a communication is ready to be performed (\text{IOComm(id,chName,...)}) and its respective effect over bindings (\text{IOCommDef(id,...)}) also exists, there must be a transition from such a state via a corresponding event (\text{CommEv(chName,chExp,chType)}) to the next state (\text{State}(st_,P)). Although the dynamic creation of states would be responsible for creating the \text{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:

\[
\text{State}(st, F), \\
\text{State}(st, Q), \\
\text{trans}(\text{State}(st, \text{iChoice}(F, Q)), \tau, \text{State}(st, F)) :- \text{State}(st, \text{iChoice}(F, Q)), \\
\text{trans}(\text{State}(st, \text{iChoice}(F, Q)), \tau, \text{State}(st, Q)) :- \text{State}(st, \text{iChoice}(F, Q)).
\]

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 (\text{[+]}) moves to another operator (\text{[+]}, where the binding is copied to each constituent part. As long as the state \text{State}(st, \text{eChoice}(F, Q)) state is reached, the system moves (via a \(\tau\) transition) to \text{State}(st, \text{extraChoice}(st, F, 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.

\[
\text{State}(st, F), \\
\text{State}(st, Q), \\
\text{trans}(iS, \tau, \text{State}(st, \text{extraChoice}(stP_, F, stQ_))) :- iS \text{ is } \text{State}(st, \text{eChoice}(F, Q)).
\]

If one of the operands terminate, the external choice also terminates. That is, the \text{extraChoice}(st1, Skip, _, _) evolves to \text{State}(st1, Skip) (or \text{extraChoice}(_, _, st2, Skip) evolves to \text{State}(st2, Skip)) via a \(\tau\) transition.

\[
\text{State}(st1, \text{Skip}), \\
\text{trans}(iS, \tau, \text{State}(st1, \text{Skip})) :- iS \text{ is } \text{State}(st, \text{extraChoice}(st1, \text{Skip}, _, _)). \\
\text{State}(st2, \text{Skip}), \\
\text{trans}(iS, \tau, \text{State}(st2, \text{Skip})) :- iS \text{ is } \text{State}(st, \text{extraChoice}(_, _, st2, \text{Skip})).
\]

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

\[
\text{State}(stF_, F), \\
\text{State}(st, \text{extraChoice}(stF_, F, stQ_)), \\
\text{trans}(iS, \tau, \text{State}(st, \text{extraChoice}(stF_, F, stQ_))) :- \text{//the combination evolves}
\]
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:

\[
\text{State}(st, \text{extraChoice}(stP, P, stQ, Q)), \\
\text{trans}(\text{State}(stP, P), \tau, \text{State}(stP_, P_)). \quad // \text{there is internal progress}
\]

Parallelism

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

\[
\text{State}(st, P, Q), \\
\text{trans}(\text{IS, ev, State}(st, \text{par}(st, P, X, st, Q))), \\
\text{trans}(\text{State}(st, P), \text{ev, State}(st, P_, X, P_)), \quad \text{ev} \neq \text{tau}, \text{ev} \neq \text{tock}.
\]

The parallel execution (\text{genPar}) starts by originating a parallel execution (\text{par}) with copies of the original bindings to each component part. Once a parallel state \text{par}(...) is created, it can progress only if its parts (premises) are available. The following rule has this purpose:

\[
\text{State}(st, \text{par}(st, P, X, st, Q), Q), \\
\text{trans}(\text{IS, ev, State}(st, \text{par}(st, P, X, st, Q))), \\
\text{trans}(\text{State}(st, P), \text{ev, State}(st, P_, X, P_)), \quad \text{fail lieIn}(\text{ev}, X).
\]

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:

\[
\text{State}(st, \text{par}(st, P, X, st, Q), Q), \\
\text{trans}(\text{IS, ev, State}(st, \text{par}(st, P, X, st, Q))), \\
\text{trans}(\text{State}(st, P), \text{ev, State}(st, P_, X, P_)), \quad \text{fail lieIn}(\text{ev}, X).
\]

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

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

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:

Once the state \( \text{State}(st, \text{proc}(P, pPar)) \) (line 1) has been reached during the LTS generation, we recover the body associated with the process name \( P \) (\( \text{RecoverBody}(n, pBody) \) in line 1) and require that there is a transition from from such a process fragment \( \text{trans}(\text{State}(st, pBody), ev, \text{State}(st, P)) \) (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 \( \text{RecoverBody} \) is useful for recovering the real body of a call, even when there are successive calls.

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).

4.1. Classical Properties in FORMULA

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 Figure 6. The notion of reachability is captured by the constructor \( \text{reachable} \) (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 \( \text{Skip} \)) from which we cannot recover a body (the state is not a call) neither go to any other state by a transition.
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. 6. Capturing classical properties in FORMULA

The constructor \texttt{tauPath} (lines 13-15) captures sequences of $\tau$-transitions between two states. A \texttt{tauPath} can have only one transition or a \texttt{tauPath} to an intermediate state and another \texttt{tauPath} to the final state.

The constructor \texttt{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 $\tau$-transitions) from the same state $L$ (\texttt{trans(L,ev1,S1) and trans(L,ev1,S2)}) leading to different states ($S1!=S2$) in which the process can accept (\texttt{accepts(S1,ev)}) or reject (\texttt{fail accepts(S1,ev)}) the same visible event ($ev!=tau$).

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

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 System of Systems (SoSs) using the CML.

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\textsuperscript{8}.

\textsuperscript{8} http://symphonytool.org.
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.

The model checker functionalities are available through the CML Model Checker perspective (see Figure 7), 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.

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

5.1. 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, whose specification contains types, values and functions as follows.

```plaintext
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.

channels
  init
  inc, dec : MYINT

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 than 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 recures 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.
To evaluate our model checker, we consider the Emergence Response System (ERS) introduced in [APR+13, ADP+13]. The example 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 ERS model (see its outline view in Figure 9) is a set of components. We extract the code that corresponds to the activation, detection and recovery of faults.

We add controller processes ERUs_0, ERUs_1. This adds details of the behaviour of the Call Centre, controlling the number of ERUs currently allocated.

The processes InitiateRescueFault1Activation and Recovery1 (see Figure 11) establish activities of fault rescue and recovery. They are used in the entire ERS (described by ERSystem_0 and ERSystem_1. The version ERUs_0 has a flaw on the implementation of the operation allocate_0: it 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 Figure 11 is deadlock-free.

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. Moreover, they found that FDR outperforms SymFDR when a counter-example does not exist. This is because SymFDR uses FDR as expansion engine but deals with specific analysis issues more efficient than FDR. In this sense, SymFDR can handle the same systems as FDR (that is, finite state space systems). Our work, on the other hand, extends the class of problems analysable by SymFDR to also incorporate some infinite state space systems. This extension is possible due to the use of SMT-solving and FORMULA to create the LTS instead of FDR.

The work found in the literature that is closest to ours was proposed in [Leu01] 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 a CTL 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 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+12] 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\(^9\). Our work differs from that of [ABG+12] 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 CCS (Concurrent Communicating Systems). 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.

\(^9\) We are implementing a Java tool that converts a CSP specification into a FORMULA script. This is a still in progress and the tool is not available yet.
Fig. 11. CML specification of the ERUs

7. Conclusions

This paper has presented a systematic way for creating a model checker from the syntax and from the structured operational semantics 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 about a test suite containing about 170 examples (property verification and traces refinement). The examples where 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.

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.

The comparison of our model checker with FDR and PAT is an intended topic for future work. Actually two interesting aspects can be taken into account for this task: efficiency and time to produce a model checker. We already have some results regarding efficiency, but they are still inconclusive. Concerning time to produce a model checker, our experience show that time is smaller with FORMULA, 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 DSL.

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).

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 [BK08]. This will certainly represent an important step towards overcoming the state explosion problem.

Acknowledgment

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.

References


10 There is a DSL for this purpose. However we are investigating if its expressiveness is enough for describing any SOS.
Model checking CML: Tool development and industrial applications


