A Symbolic Rewriting Semantics of the COMPASS Modeling Language

Musab A. Alturki
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
musab@kfupm.edu.sa

Abstract

Modern information systems are built through the integration of different, autonomous components that may themselves be complex systems. This Systems-of-Systems (SoS) architecture is foreseen to underlie future information systems primarily because it promotes modularity, facilitating reuse and maintenance. Nevertheless, engineering verifiably dependable SoS is particularly difficult due to their complexity. In this paper, we present a semantics-based approach for the formal modeling and analysis of SoS using the COMPASS Modeling Language or CML (compass-research.eu). We present an executable rewriting specification of CML in rewriting logic that captures its operational semantics. The semantics is based on a variant of CML's structural operational semantics and, therefore, its correctness is straightforward. We then generalize the semantics into a symbolic rewriting semantics with which open systems can be symbolically executed and analyzed through the integration of rewriting with solving satisfiability problems modulo arithmetic theories. We illustrate the approach through an example using a version of the Maude tool that integrates with the CVC3 solver.

1. Introduction

Modern information systems are built through the integration of different, autonomous components that may themselves be complex systems. Through this integration, the components operate cooperatively to provide a larger, more useful whole. This Systems-of-Systems (SoS) architecture is foreseen to underlie future information systems primarily because it promotes modularity, facilitating reuse and maintenance. Consequently, dependability of these systems, especially in mission- and business-critical applications, is increasingly becoming a major concern. Nevertheless, engineering verifiably dependable SoS is a particularly difficult task due to their complexity and the heterogeneity of their constituent parts.

Formal modeling and analysis of SoS requires integrating different facilities and methods to manage their complexity and properly address their dependability properties. In an ambitious effort to develop an integrated framework for this purpose, the COMPASS (Comprehensive Modeling for Advanced Systems of Systems) consortium (compass-research.eu) developed a unified, broadly accessible, formal modeling language, the COMPASS Modeling Language or CML [6, 17]. The language serves as an intermediate language between higher-level modeling languages and lower-level implementation platforms. It integrates three existing frameworks: (1) The Vienna Development Method (VDM) [2], a well-established formal method for the specification and verification of systems; (2) Communicating Sequential Processes (CSP) [7], a calculus for modeling message-passing concurrent systems; and (3) Circus [16], a language for refinements. Furthermore, suitability of CML for modeling large-scale SoS has been demonstrated through several case studies.

We aim in this work to develop automated formal reasoning tools for SoS based on CML. Developing such tools amounts to developing an executable formal specification of CML that captures precisely its operational semantics. For this purpose, we use rewriting logic [8], an expressive computational logic in which systems can be naturally specified. The suitability and usefulness of rewriting logic as a semantic framework has now been well established through a long series of developments within the rewriting logic semantics project (see [10, 12] for recent surveys). It is also well supported by tools in its implementation in Maude [3].

In this paper, we present an executable operational semantics of CML in rewriting logic that is based on a variant of the structural operational semantics of CML given in [5]. The rewriting semantics is obtained through the semantics-preserving transformation of [11] and, thus, its correctness is a straightforward corollary of the correctness of the transformation. We then generalize the semantics into a *symbolic* rewriting semantics with which open systems can be symbolically executed and analyzed. This is enabled by the integration of term rewriting and satisfiability modulo theo-

ries (SMT), through which the technique of *rewriting modulo SMT* emerged [14]. In this technique, a symbolic state is a pair $(t; \varphi)$, with t a symbolic term and φ a symbolic constraint solvable by an SMT solver. The pair $(t; \varphi)$ represents a possibly infinite set of states that are instances of t satisfying φ. As rewriting modulo SMT reduces to standard rewriting [14], the symbolic specification of CML is executable in versions of Maude that integrate with an external SMT solver (CVC3). We illustrate this approach through a simple working example .

The rest of the paper is organized as follows. Section 2 overviews the syntax and operational semantics of CML. A rewrite theory giving concrete semantics of CML in rewriting logic is presented in Section 3. This is followed in Section 4 with a generalization of that theory into a symbolic semantics of CML and an example showing how the semantics may be used. Section 5 overviews related work. The paper concludes with a discussion of future work.

2. The COMPASS Modeling Language

The COMPASS Modeling Language or CML [6, 17] is an intermediate language for modeling and analyzing SoS. It integrates three existing frameworks: VDM [2], CSP [7] and Circus [16]. An integrated tool built around CML has also been developed [4]. Being based on three different languages, CML is quite expansive, including many constructs and features from all three languages. CML has rich state, rich operations, concurrency, communication, real-time, object-orientation, and process-algebraic combinators. Although being expansive helps target a wider user-base, it presents a challenge for formal reasoning and analysis. However, considering all features of CML for formal verification is not necessary. We can identify a *kernel* of CML with which we can define all other features in the language as derived constructs, and apply formal reasoning techniques on this kernel. This was already done for CML in [15], when developing CML's formal semantics in UTP. Here, we identify a slightly different kernel, denoted CML_k, and describe it next.

2.1. Syntax of CML_k

Figure 1 shows the abstract syntax of CML_k. We assume a syntactic category of names (*Name*), which may represent program variables (abstractions of values) or channel names. We also assume a syntactic category of expressions, which are terms that denote values of possibly different types (e.g. integers, booleans, reals, ...).

The basic unit of a CML_k specification is a process. Primitive processes can be: (1) **skip**, representing normal termination, (2) the divergent process, **dive**, that never terminates, or (3) the assignment process, $x := e$, assigning

$x, c \in \mathit{Name} \quad e \in \mathit{Expression}$

$a \in$ Action	::=	$c\,?\,x : e$	Input
	|	$c\,!\,e$	Output
$P \in$ Process	::=	**skip**	Termination
	|	**dive**	Divergence
	|	$x := e$	Assignment
	|	$[e, e]$	Design
	|	$a \rightarrow P$	Prefixed process
	|	$P \lhd e \rhd P$	Conditional
	|	$P \setminus c$	Hiding
	|	$P; P$	Sequencial composition
	|	$P \sqcap P$	Internal choice
	|	$P \boxdot P$	External choice
	|	$P [\![\vec{c}]\!] P$	Parallel composition
	|	$\mu x.P(x)$	Recursion

Figure 1. Abstract Syntax of CML_k

the value of the expression e to the variable x. A process may also be specified abstractly as a design (specification) $[p, q]$, where p is the pre-condition and q is the post-condition, representing all processes that when executed at a state satisfying p will terminate in a state that satisfies q.

Furthermore, as in CSP, larger processes may be formed using one or more of the following combinators. The prefixed process, $a \rightarrow P$, performs the (observable) action a and then behaves as P. An action can be either an *input* action $c?x : e$, where e is an expression that evaluates to a set of values that may be input through the channel c and assigned to x, or an *output* action $c\,!\,e$, where the value of e is output through c. A process may also be the sequential composition of two processes $P_1; P_2$, which executes P_1 to completion, and then behaves as P_2. The internal choice of two processes $P_1 \sqcap P_2$ non-deterministically selects to execute either P_1 or P_2. The external choice $P_1 \boxdot P_2$, on the other hand, leaves this selection to the environment, so that both processes are executed, but once a process makes an observable transition (or finishes execution), the other process is aborted. A process may also be the synchronized parallel composition of two processes $P_1 [\![\vec{c}]\!] P_2$, in which P_1 and P_2 are executed concurrently while synchronizing on actions performed through the channels $\vec{c}$. The abstraction $P \setminus c$ hides actions through c in P so that they are externally unobservable. Finally, $\mu x.P(x)$ is the recursive process for defining potentially infinite behavior.

We now give a simple example to illustrate the semantics. This is a working example that will be used throughout the rest of the paper to illustrate the different formal semantics presented (It is intentionally kept simple given the space limitations). Consider the sequential (but non-deterministic) reactive process defined as:

$$c\,?\,x : \{7+3, 3*10, 9\} \rightarrow (x := x+1; c\,!\,x \rightarrow \mathbf{skip}) \quad (\star)$$

The process first receives a value from the set of possible values from the environment through channel c and binds

it to x. It then increments the value of x and outputs the new value through the same channel, before it concludes execution. Despite being sequential, the process is non-deterministic, as it models an open system that receives input from the environment.

2.2. Operational Semantics

Automated formal analysis of SoS in CML amounts to devising an executable operational semantics of CML_k. For this purpose, we adopt the Structural Operational Semantics (SOS) developed in the COMPASS Project for (a kernel of) CML [5]. The semantics formally specifies program behavior as a state transition system by defining a transition relation on program states. The transition relation specifies the computation steps exhibited by the different constructs in the language. Furthermore, the operational semantics is *symbolic*, using symbolic integers (called *loose constants* in [5]) enabling the succinct representation of a set of related transitions as a single abstract transition.

In our SOS of CML_k, which is directly based on the SOS given in [5], a state is represented by a configuration of the form $\langle c \mid s \models P \rangle$, where c is a symbolic constraint, s is an environment mapping variables to values (or symbolic integers) and P is a CML_k process. A labeled transition relation $\overset{l}{\hookrightarrow}$ on configurations defines an interleaving semantics of CML_k processes. The label l denotes the type of action taken when the corresponding transition occurs. Assuming d is a channel name and w is a symbolic integer, there are four possible types of actions: the input action $d?w$, the output action $d\,!\,w$, the input/output (synchronization) action $d.w$ and the internal (unobservable) action τ. The labeled transition relation is defined inductively over the structure of a CML_k process as the smallest relation that can be generated by a specific set of SOS rules, a sample of which is listed in Figure 2. The full list of SOS rules can be found online at `www.ccse.kfupm.edu.sa/~musab/cml-fmi`. Some representative rules will be described in Section 3 in the context of their rewriting specifications.

To illustrate how the SOS rules capture the symbolic semantics of CML_k, we present below the execution trace of our example process $\star$ introduced in Section 2.1:

$$
\begin{aligned}
&\langle \top \mid \emptyset \models \star \rangle \\
&\overset{c\,?\,w_0}{\hookrightarrow} \langle w_0 \in \{10,30,9\} \mid [x \leftarrow w_0] \models \\
&\qquad\qquad \mathbf{block}_x\ (x := x+1; c\,!\,x \rightarrow \mathbf{skip}) \rangle \\
&\overset{\tau}{\hookrightarrow} \langle w_0 \in \{10,30,9\} \wedge w_1 = w_0 + 1 \mid [x \leftarrow w_1] \models \\
&\qquad\qquad \mathbf{block}_x\ (c\,!\,x \rightarrow \mathbf{skip}) \rangle \\
&\overset{c\,!\,w_2}{\hookrightarrow} \langle w_0 \in \{10,30,9\} \wedge w_1 = w_0 + 1 \wedge w_2 = w_1 \mid [x \leftarrow w_1] \models \\
&\qquad\qquad \mathbf{block}_x\ \mathbf{skip} \rangle \\
&\overset{\tau}{\hookrightarrow} \langle w_0 \in \{10,30,9\} \wedge w_1 = w_0 + 1 \wedge w_2 = w_1 \mid \emptyset \models \mathbf{skip} \rangle
\end{aligned}
$$

The first transition, which is a consequence of applying the input rule (invoked by the prefixed process rule), involves generating a fresh symbolic constant, mapping it to x and adding the constraint that it can only be one of the values given by the set $\{10, 30, 9\}$. The internal construct $\mathbf{block}_x$ is used in the program text to keep track of the scope of x. The second transition generates w_1 (for the value of the expression $x + 1$) and appropriately updates both the mapping to x in the environment and the symbolic constraint. This is followed by generating an output event in the third transition. Finally, the last transition drops the scope of x. We note that each step of execution of the process $\star$ (except the last) involves generating a fresh symbolic constant and accumulating constraints on these constants. We also note that all these steps are feasible, having constraints that can be easily checked satisfiable using an SMT solver.

$$\frac{}{\langle c \mid s \models x := e \rangle \overset{\tau}{\hookrightarrow} \langle c \wedge (w_0 = e \Downarrow_s) \mid s[x/w_0] \models \mathbf{skip} \rangle}\ c \quad \text{ASSIGN}$$

$$\frac{}{\langle c \mid s \models d?x : e \rightarrow P \rangle \overset{d?w_0}{\hookrightarrow} \langle c \wedge w_0 \in T \mid s[x \leftarrow w_0] \models \mathbf{block}_x\ P \rangle}\ c \quad T = e \Downarrow_s \quad T \neq \phi \quad \text{INPUT}$$

$$\frac{}{\langle c \mid s \models d\,!\,e \rightarrow P \rangle \overset{d\,!\,w_0}{\hookrightarrow} \langle c \wedge (w_0 = e \Downarrow_s) \mid s \models P \rangle}\ c \quad \text{OUTPUT}$$

$$\frac{\langle c_1 \mid s_1 \models P_1 \rangle \overset{l}{\hookrightarrow} \langle c_2 \mid s_2 \models P_2 \rangle}{\langle c_1 \mid s_1 \models P_1; Q \rangle \overset{l}{\hookrightarrow} \langle c_2 \mid s_2 \models P_2; Q \rangle}\ \text{SEQ-PROGRESS}$$

$$\frac{}{\langle c \mid s \models \mathbf{skip}; Q \rangle \overset{\tau}{\hookrightarrow} \langle c \mid s \models Q \rangle}\ c\ \text{SEQ-END}$$

Figure 2. Sample SOS Rules of CML_k (see [5])

3. Rewriting Semantics of CML_k

We aim to develop a formal semantics of CML_k that is both provably correct and executable, so that the semantics may be used to prototype, simulate and formally analyze its programs. For this, we describe in this section a rewriting semantics obtained by transforming the SOS of CML_k outlined in Section 2.2 into a rewriting logic specification using the correctness-preserving transformation given in [11].

3.1. Preliminaries

The unit of specification in rewriting logic is a *rewrite theory* $\mathcal{R}$, which is a triple $\mathcal{R} = (\Sigma, E \cup A, R)$ consisting of the following components: (1) Σ, a signature declaring kinds, sorts and operators; (2) E, a set of Σ-sentences, which are universally quantified equations $(t = t')$ or memberships $(t : s)$, (3) A, a set of equational axioms, such as

commutativity, associativity and identity; and (4) R, a set of universally quantified labeled *rewrite rules* of the form: $(\forall X)\ r : t \to t'$ **if** C where r is a label and C is (possibly) a conjunction of equational and rewrite conditions. The theory $(\Sigma, E \cup A)$ is a membership equational logic [9] theory, which gives a formal description of the static state structure. The rules R, on the other hand, specify the dynamic behavior (transitions) of the system. A rewrite theory $\mathcal{R}$ proves a statement of the form $(\forall X)\ t \to t'$, meaning that, in $\mathcal{R}$, any instance of the state term t can *reach* the corresponding instance of the state term t' in a finite number of, possibly concurrent, steps. A detailed discussion of rewriting logic as a unified model of concurrency and its inference system can be found in [8] (see also the survey [10]).

A high-performance and feature-full implementation of rewriting logic is provided by Maude [3]. Theories, specified as modules, can be executed (under certain reasonable executability assumptions) and analyzed using its arsenal of formal verification tools. Furthermore, recent versions of Maude integrates with the CVC SMT solver [1] providing an implementation for rewriting modulo SMT [14], which we exploit in this work.

3.2. A Rewrite Theory of CML_k

In this section, we describe an initial rewrite theory $\mathcal{R}_c = (\Sigma_c, E_c \cup A_c, R_c)$ that captures the *concrete* semantics of CML_k without symbolic constants. The extension to the symbolic semantics is deferred to Section 4. The full specification has been developed in Maude and is available online at `www.ccse.kfupm.edu.sa/~musab/cml-fmi` with instructions on how it may be used. In the description below, we only highlight representative parts of the theory and refer the reader to the specification available online for other details.

Expressions. The signature Σ_c declares sorts for integer and Boolean values and expressions with the expected subsorting hierarchy. In particular, for integers, the sorts IntCns and IntName represent, respectively, integer values and integer names (or variables). The integer expressions sort Int includes values and names as subsorts, in addition to larger expressions that are constructed using standard arithmetic operators, such as negation, addition, and so on. An example operator declaration is shown below:

$$\_ + \_ \ :\ \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int} \quad [\mathsf{assoc}\ \ \mathsf{comm}\ \ \mathsf{prec}\ 35]$$

The operator is declared in mixfix notation, where the underscores specify locations of arguments. Operators may also have equational attributes (constituting A_c in the theory $\mathcal{R}_c$), such as assoc for associativity and comm for commutativity for integer addition. To minimize use of parenthesis, a precedence value may also be specified using prec: the lower the value, the higher the precedence. A similar set of sort, subsort and operator declarations are provided for Boolean expressions, represented by the sort Bool. The sort Expr, a supersort of both Int and Bool, models all valid expressions in CML_k.

Auxiliary functions on expressions are defined inductively over the structure of an expression using equations (in E_c). $\mathcal{R}_c$ defines a collection of these functions to facilitate the specification of the semantics of CML_k.

Environment. To evaluate integer expressions, which may contain names of the sort IntName, an environment providing a mapping of names to values is needed. In $\mathcal{R}_c$, the environment (of the sort Env) is conveniently constructed as the associative empty juxtaposition of mappings of the form $[\mathsf{x} \mapsto \mathsf{v}]$, mapping the name x to the value v. There are two constructors for the list of mappings:

$$\_\_ \ :\ \mathsf{Env} \times \mathsf{Env} \to \mathsf{Env}\ [\mathsf{assoc\ id} : \mathbf{mt}] \qquad \mathbf{mt}\ : \to \mathsf{Env}$$

The empty juxtaposition operator is declared associative with identity element $\mathbf{mt}$, the constructor of the empty environment. We assume through this representation a model of the environment that is aligned as a stack whose top element is the rightmost element. Not only does this allow for a convenient and succinct definition of operations on the environment (as we will see shortly), it also provides a mechanism for managing name scopes, with the rightmost mapping of a name representing the innermost scope in which this name is declared.

Environment lookup and update operations are declared and equationally defined. The lookup function $\sigma[x]$ returns the value given by the rightmost (most recent) mapping for x in σ, while the update function $\sigma[x/v]$ returns an environment that is almost the same as σ, but with the most recent mapping for x updated to have the value v. The equations defining these and other operators on the environment exploit matching modulo associativity and identity algorithms (implemented in Maude). For example, the definition of the lookup operation is shown below:

$$(\sigma[x \mapsto v])[x] = v$$
$$(\sigma[y \mapsto v])[x] = \sigma[x] \text{ if } y \neq x$$

The first equation applies if the name being looked up matches the rightmost mapping of the current environment, in which case the value given by this mapping is used. The second equation applies when the name does not match, and in this case, the name is recursively looked up in the rest of the environment. Other operations on environments are similarly defined.

Now that we have an environment, we may now define the semantics of integer and Boolean expressions through the (overloaded) evaluation operator:

$$\_ \Downarrow \_ : \mathsf{Int} \times \mathsf{Env} \to \mathsf{IntCns}$$
$$\_ \Downarrow \_ : \mathsf{Bool} \times \mathsf{Env} \to \mathsf{BoolCns}$$

Evaluation is deterministic and is, thus, defined using equations by induction over the structure of expressions. Its definition uses the operations on environments as appropriate.

Processes. To model processes, the signature Σ_c declares a sort Process whose elements are constructed by operators matching the syntax of CML_k given in Figure 1. We show below the prefixed process operator as an example:

$$\_ \to \_ : \mathsf{Action} \times \mathsf{Process} \to \mathsf{Process}$$

where the sort Action defines input and output actions of the forms $d?x : e$ and $d\,!\,e$, respectively.

Labels. The transition relation defined by the SOS of CML is labeled, where a label scheme represents the type of action taken by a process when making a transition. In $\mathcal{R}_c$, we declare a sort hierarchy for labels:

$$\mathsf{SLabel}\ \mathsf{NSLabel} < \mathsf{Label} < \mathsf{MaybeLabel}$$

with SLabel the sort for the silent τ transition and NSLabel the sort for non-silent (input/output/synchronization) labels. The MaybeLabel is a supersort that includes the possibility of having a "no label" ϵ. The following operators construct labels of the above sorts:

$$\tau : \to \mathsf{SLabel} \qquad \epsilon : \to \mathsf{MaybeLabel}$$
$$\langle \_\ \_\ \_ \rangle : \mathsf{IntName} \times \mathsf{Mode} \times \mathsf{Expr} \to \mathsf{NSLabel}$$

with Mode being either the input indicator $(?)$, the output indicator $(!)$ or the synchronized input/output indicator $(.)$.

Configurations. A state in the execution of a CML_k process is modeled by a *configuration*, which is a pair of the form $\langle R \models P \rangle$, with P the process to be executed and R a record structure consisting of indexed semantic fields that are needed for properly capturing the semantics of a CML_k process. In the concrete semantics $\mathcal{R}_c$ of CML_k described here, the record R consists only of two fields: (1) the *environment field*, which maintains the current environment σ mapping names to values, and (2) the *label field*, which maintains the label λ of the last transition that led to the current state. Therefore, a configuration of a CML_k process has the following form: $\langle \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \models P \rangle$.

Rewrite rules. Now that we have the semantic infrastructure defined, we may specify the semantics of processes in CML_k using rewrite rules R_c in the theory $\mathcal{R}_c$.

$$\textsc{Assign} : \langle \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \mid R \models x := e \rangle \Rightarrow \langle \mathsf{e} : (\sigma[x/v]) \mid \mathsf{l} : \tau \mid R \models \mathbf{skip} \rangle \ \text{if } v := e \Downarrow \sigma$$

$$\textsc{Input} : \langle \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \mid R \models c?x : e \to P \rangle \Rightarrow \langle \mathsf{e} : (\sigma[x \leftarrow v]) \mid \mathsf{l} : \langle c?v \rangle \mid R \models \mathbf{block}_x\ P \rangle \ \text{if } v \in e \Downarrow \sigma$$

$$\textsc{Output} : \langle \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \mid R \models c\,!\,e \to P \rangle \Rightarrow \langle \mathsf{e} : \sigma \mid \mathsf{l} : \langle c\,!\,v \rangle \mid R \models P \rangle \quad \text{if } v := e \Downarrow \sigma$$

$$\textsc{Seq-progress} : \langle R_1 \models P_1; P \rangle \Rightarrow \langle R_2 \models P_2; P \rangle \ \text{if } \langle R_1 \models P_1 \rangle \Rightarrow \langle R_2 \models P_2 \rangle$$

$$\textsc{Seq-end} : \langle R \mid \mathsf{l} : \lambda \models \mathbf{skip}; P \rangle \Rightarrow \langle R \mid \mathsf{l} : \tau \models P \rangle$$

Figure 3. Sample rewrite rules in R_c

This is achieved systematically by applying the semantics-preserving transformation from (Modular) SOS to rewriting logic given in [11]. The transformation maps the SOS rules into corresponding topmost conditional rewrite rules operating on CML_k configurations that are very similar in structure to their SOS counterparts. Furthermore, for every SOS rule in the semantics of CML_k, there is exactly one corresponding rewrite rule in R_c. Figure 3 lists the rewrite rules that correspond to the rules in Figure 2.

To illustrate how the rules operate, we use the example process $\star$ presented in Section 2.1. An execution of $\star$ in $\mathcal{R}_c$ starting from its initial configuration $\langle \mathsf{e} : \emptyset \mid \mathsf{l} : \lambda \models \star \rangle$ is shown below:

$$\begin{aligned}
&\langle \mathsf{e} : \emptyset \mid \mathsf{l} : \lambda \models \star \rangle \\
&\Rightarrow \langle \mathsf{e} : [x \leftarrow 10] \mid \mathsf{l} : \langle c\,?\,10 \rangle \models \mathbf{block}_x\ (x := x + 1; c\,!\,x \to \mathbf{skip}) \rangle \\
&\Rightarrow \langle \mathsf{e} : [x \leftarrow 11] \mid \mathsf{l} : \tau \models \mathbf{block}_x\ (c\,!\,x \to \mathbf{skip}) \rangle \\
&\Rightarrow \langle \mathsf{e} : [x \leftarrow 11] \mid \mathsf{l} : \langle c\,!\,11 \rangle \models \mathbf{block}_x\ \mathbf{skip} \rangle \\
&\Rightarrow \langle \mathsf{e} : \emptyset \mid \mathsf{l} : \tau \models \mathbf{skip} \rangle
\end{aligned}$$

It is worth noting that this is one possible execution (in which the environment gives the value 10 to the process). There are potentially two other possible executions.

4. Symbolic Semantics of CML_k

The concrete semantics given by $\mathcal{R}_c$ captures most of CML_k features and, through its implementation in Maude, can be very useful for prototyping SoS specifications and analyzing their properties. However, the semantics is too detailed allowing only explicit states with each possible behavior modeled by an independent execution path. This can render formal analysis of specifications very inefficient and sometimes unnecessarily detailed. Furthermore, being concrete, the semantics allows only trivial designs $[p, q]$ in which ground instances of p and q (in which all references to names have been resolved) are (equationally) equivalent to a Boolean value (either true or false).

By integrating term rewriting with satisfiability modulo theories (SMT), the technique of *rewriting modulo SMT*

emerged [14]. In this technique, a symbolic state is a pair $(t; \varphi)$, with t a symbolic term and φ a symbolic constraint solvable by an SMT solver. The pair $(t; \varphi)$ represents a possibly infinite set of states that are instances of t satisfying φ. Rewriting modulo SMT symbolically rewrites the pair $(t; \varphi)$ to another using rewrite rules of the form:

$$t(\vec{x}) \to t'(\vec{x}, \vec{y}) \text{ if } \varphi(\vec{x}, \vec{y})$$

where $\vec{y}$ are new variables (appearing on the right-hand side of the rule) that capture non-deterministic changes in the symbolic state (e.g. environment input).

We generalize the theory $\mathcal{R}_c$ specifying concrete semantics of CML_k to a new theory $\mathcal{R}_s = (\Sigma_s, E_s \cup A_s, R_s)$ that captures its symbolic semantics. Like $\mathcal{R}_c$, the symbolic semantics is obtained through the standard transformation of SOS into rewriting logic [11], through which correctness of the semantics can be easily established. Furthermore, the specification of the theory $\mathcal{R}_s$ in Maude, integrated with CVC3, enables symbolic verification of CML_k programs.

4.1. Changes in the Semantic Infrastructure

Symbolic constants. The signature Σ_s introduces the sort IntVar for symbolic integer constants, a subsort of the sort of integer expressions Int. Symbolic constants are constructed using the operator $i : \mathsf{Nat} \to \mathsf{IntVar}$, providing a countably infinite supply of them as desired. We note that, with this extension, expressions may now include both integer names (or variables) and symbolic integer constants, in addition to other operators, e.g. $x + 1 > w$, with x a name and w a symbolic constant. Therefore, constraints, which are Boolean expressions such as $x + 1 > w$, can now be symbolic and their satisfiability will need to be checked through a satisfiability checker modulo the theory of arithmetic. As a consequence, this enables the specification of designs that are much more interesting and useful than the ones available through the concrete semantics. Furthermore, the environment is now extended so that mappings of names to symbolic constants are also possible. Finally, labels are also modified so that non-silent labels include a symbolic constant component instead of an integer value.

Configurations. A state in the symbolic execution of a CML_k program must now maintain the constraints accumulated so far on symbolic constants. This is achieved by adding a new field in the record of a configuration in the symbolic semantics $\mathcal{R}_s$ of the form $c : \varphi$, with φ a Boolean expression that may contain names and symbolic constants. For efficiency and convenience, we add two more fields that help in managing the semantics of symbolic constants. The first field is the satisfiability flag field $s : b$ (with b a Boolean value), which indicates whether the currently accumulated constraint is satisfiable or not. The second field $w : n$ maintains a counter n for the generation of fresh symbolic constants whenever needed by the configuration. Consequently, a configuration in $\mathcal{R}_s$ has the following form:

$$\langle \mathsf{s} : b \mid \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \mid \mathsf{c} : \varphi \mid \mathsf{w} : n \mid\models P \rangle$$

4.2. Rewrite Rules in R_s

Most of the rewrite rules in R_s giving symbolic semantics to CML_k are very similar to those in R_c. This is primarily due to the modular structure of the rules promoted by rewriting logic and the transformation of [11], in which rules mention only the fields they need and allow, through matching modulo associativity and commutativity, unspecified additional fields in a configuration. There are a few exceptions, however, since a few rules, namely those for assignments, input and output, designs and conditionals, require fundamental changes. We highlight next the main changes needed in the ASSIGN and INPUT rules below as representative examples. Changes in other rules are similar.

The symbolic assignment rule ASSIGN is as follows:

$$\begin{aligned} &\text{ASSIGN} : \langle \mathsf{s} : \top \mid \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \mid \mathsf{c} : \varphi \mid \mathsf{w} : n \mid R \models x := e \rangle \\ &\Rightarrow \langle \mathsf{s} : sat?(\varphi') \mid \mathsf{e} : \sigma[x/i_n] \mid \mathsf{l} : \tau \mid \mathsf{c} : \varphi' \mid \mathsf{w} : s(n) \mid R \models \mathbf{skip} \rangle \\ &\qquad\qquad \text{if } \varphi' := (\varphi \wedge (i_n = e \Downarrow_\sigma)) \end{aligned}$$

Compared with the assignment rule in the concrete semantics shown in Figure 3, we first note that the rule is conditional with a *matching equation* in its condition (A matching equation $p := t$ evaluates the right-hand side term t to its canonical form and then matches it with the pattern p on the left-hand side, instantiating its variables as needed). The rule applies only when the satisfiability flag is set, meaning that the configuration represents a reachable state that can be progressed further. When it fires, the rule creates a fresh symbolic constant i_n and conjoins the constraint φ with the requirement that i_n is equal to the expression E evaluated in the current state σ. The simplified form of this augmented constraint instantiates the variable φ', which is used in the rule to update the constraint of the configuration. The satisfiability flag is also updated using the operator $sat?$, which invokes the SMT solver, as described in Section 4.3 below. Finally, the constant to which x is mapped in Σ is updated to the new symbolic constant i_n. It is worth noting here that, unlike the concrete semantics, the evaluation of the expression $e \Downarrow \sigma$ may not necessarily result in a value, as the expression may contain symbolic constants.

The symbolic input rewrite rule INPUT is as follows:

$$\begin{aligned} &\text{INPUT} : \langle \mathsf{s} : \top \mid \mathsf{e} : \sigma \mid \mathsf{l} : \lambda \mid \mathsf{c} : \varphi \mid \mathsf{w} : n \mid R \models c?x : e \to P \rangle \\ &\Rightarrow \langle \mathsf{s} : sat?(\varphi') \mid \mathsf{e} : \sigma[x \leftarrow i_n] \mid \mathsf{l} : \langle c\,?\,i_n \rangle \mid \mathsf{c} : \varphi' \mid \mathsf{w} : s(n) \mid \\ &\qquad\qquad R \models \mathbf{block}_x\ P \rangle \\ &\qquad\qquad \text{if } \varphi' := (\varphi \wedge (i_n \in e \Downarrow \sigma)) \end{aligned}$$

The expression e is a set of subexpressions representing potential inputs from the environment. The (nondeterministic) choice of input is captured by a fresh symbolic constant i_n. Moreover, a mapping for x to i_n is added to the environment, and the constraint that i_n belongs to the set e is added to the constraint φ. We note that the label captures the input event of the symbolic constant i_n on the channel c.

As an example, we show below the rewriting steps involved in the execution of our working example process $\star$:

$$
\begin{array}{l}
\langle \mathsf{s} : \top \mid \mathsf{w} : 0 \mid \mathsf{e} : \emptyset \mid \mathsf{l} : \lambda \mid \mathsf{c} : \top \models \star \rangle \\
\Rightarrow \langle \mathsf{s} : \top \mid \mathsf{w} : 1 \mid \mathsf{e} : [x \leftarrow i_0] \mid \mathsf{l} : \langle c\,?\,i_0 \rangle \mid \\
\quad \mathsf{c} : i_0 \in \{10, 30, 9\} \models \mathbf{block}_x\ (x := x + 1; c\,!\,x \rightarrow \mathbf{skip}) \rangle \\
\Rightarrow \langle \mathsf{s} : \top \mid \mathsf{w} : 2 \mid \mathsf{e} : [x \leftarrow i_1] \mid \mathsf{l} : \tau \mid \\
\quad \mathsf{c} : i_0 \in \{10, 30, 9\} \wedge i_1 = s(i_0) \models \mathbf{block}_x\ (c\,!\,x \rightarrow \mathbf{skip}) \rangle \\
\Rightarrow \langle \mathsf{s} : \top \mid \mathsf{w} : 3 \mid \mathsf{e} : [x \leftarrow i_1] \mid \mathsf{l} : \langle c\,!\,i_2 \rangle \mid \\
\quad \mathsf{c} : i_0 \in \{10, 30, 9\} \wedge i_1 = s(i_0) \wedge i_2 = i_1 \models \mathbf{block}_x\ \mathbf{skip} \rangle \\
\Rightarrow \langle \mathsf{s} : \top \mid \mathsf{w} : 3 \mid \mathsf{e} : \emptyset \mid \mathsf{l} : \tau \mid \\
\quad \mathsf{c} : i_0 \in \{10, 30, 9\} \wedge i_1 = s(i_0) \wedge i_2 = i_1 \models \mathbf{skip} \rangle
\end{array}
$$

This sequence of rewrites captures precisely the symbolic semantics of $\star$, and is identical to that given by the SOS semantics (shown in Section 2.2). Note that this sequence is a generalization of the sequence of rewrites generated by the concrete rewriting semantics $\mathcal{R}_c$ (shown in Section 3.2), as desired. It captures all possible concrete executions through symbolic constants and constraints. We also note that feasibility of every step of the sequence is indicated by the satisfiability flag.

4.3. Integration with the SMT Solver

In the symbolic semantics $\mathcal{R}_s$, each rewrite step that updates the constraint field in a configuration, such as the steps induced by the ASSIGN and INPUT rules, requires invoking an SMT solver to check satisfiability of the constraint. The implementation of the semantics in Maude integrated with CVC3 facilitates execution of its induced rewrite relation, enabling dynamic formal analysis of CML_k programs.

The interface to the SMT solver is algebraically specified using the operator `check-sat`, which is declared as a special operator that can invoke the external solver (it is hooked in Maude to the procedure responsible for invoking the solver). The operator takes as input the constraint formatted as a valid string of input to be supplied to the solver, and returns its output string. The input string that corresponds to a given symbolic Boolean expression is generated using an operator $translate : \mathsf{Bool} \rightarrow \mathsf{String}$, which is defined inductively over the structure of a Boolean expression. The $sat?$ operator, which we saw in Section 4.1 above, is a wrapper function for `check-sat` that translates the input expression (using $translate$) into a valid CVC3 string and passes it to `check-sat`. $sat?$ then examines the output of the solver and returns either true or false accordingly.

Through this interface, the specification in $\mathcal{R}_s$ is fully executable in Maude. Dynamic formal verification of CML_k can now be achieved by leveraging the full arsenal of generic formal analysis tools available in Maude, including searching and model checking. As an example, searching for the terminal state(s) of the process $\star$ (given in Section 2.1) gives the following result:

```
search in CML : {c ? x : {3 + 7, 9, 3 * 10} ->
    (x := 1 + x ; c ! x -> skip)} =>! C:Config .
Solution 1 (state 5)
C:Config -->
  < s : true | w : 3 | e : mt | l : tau |
    c : (9 = i(0) v 10 = i(0) v 30 = i(0)) ^
        i(1) = i(2) ^ i(1) = 1 + i(0) |= skip >)
No more solutions.
states: 6  rewrites: 269 in 8ms cpu (16ms real)
```

with conjunction and disjunction denoted by `^` and `v` respectively in the constraint field. The value `true` in the satisfiability field is the result of checking this constraint using the external SMT solver. To see details about the execution path that led to this state, we may use the following command (only partial output is shown):

```
Maude> show path 5 .
state 0, Config: ... ===[ input ]===>
state 1, Config: ... ===[ block-progress ]===>
state 2, Config: ... ===[ block-progress ]===>
state 3, Config: ... ===[ block-progress ]===>
state 4, Config: ... ===[ block-end ]===>
state 5, Config: ...
```

The output shows the execution steps and the labels of rules used in each. We note here the assignment and output operations in $\star$ occur inside a block, and hence the applications of the `block-progress` rules.

5. Related Work

CML has been well documented in the deliverables of the COMPASS project, which are publicly available online at the COMPASS Consortium website `compass-research.eu`. In addition to the timed, denotational semantics in UTP [15], a particularly relevant report is Deliverable D23.4c [5], which documents in detail a timed, symbolic, structural operational semantics of CML, which is the basis for the rewriting semantics described in this work. Furthermore, this structural operational semantics was later used to develop a symbolic model checking tool for CML specifications through an embedding of the semantics in Microsoft's FOrmal Modeling Using Logic programming and Analysis (FORMULA) framework [13]. The embedding uses FORMULA's abstract data type specification and constraint logic programming facilities to define the labeled transition system given by the SOS rules of CML. Moreover, symbolic constraint solving in FORMULA is enabled through the integration with the Z3 SMT

solver. The symbolic rewriting semantics described in this work provides a natural platform for developing a symbolic model checker for CML without the need to do any kind of embedding as the rewriting specification is directly executable and rewrite theories have natural Kripke structure representations based on which model checking can directly be performed.

Several recent developments within the *rewriting logic semantics project* that are closely related to this work appeared in the literature. Representative examples include the symbolic rewriting specifications of the CASH real-time scheduling algorithm and NASA's Plan Execution Interchange Language (PLEXIL) [14]. Through their implementations in Maude and the integration with the the external SMT solver CVC3, the specifications were used to build symbolic model checking tools capable of verifying temporal logic properties against generic initial configurations.

6. Concluding Remarks

We presented a semantics-based approach for the formal modeling and analysis of specifications in the COMPASS Modeling Language (CML). The approach is based on developing an executable formal semantics of CML in rewriting logic, that is based directly on CML's structural operational semantics. The semantics is also symbolic, enabling symbolic reasoning of CML specifications. Using the Maude tool, and its integration with the CVC3 SMT solver, the analysis is mechanized. A simple working example was used to illustrate the semantics presented.

There are several directions for further developments and research, including (1) extending the semantics to a timed semantics, capturing timed behaviors of CML processes, and (2) developing a symbolic model checking tool based on the timed rewriting semantics of CML and using Maude's model checking facilities, among others.

Acknowledgments. I thank Jim Woodcock and Simon Foster for hosting me at the University of York in the summer of 2013 and for their very helpful discussions that eventually led to this work. I also thank José Meseguer and Camilo Rocha for their help with symbolic rewriting modulo SMT. This work was supported by King Fahd University of Petroleum and Minerals through Grant IN141016, and through the Post-Doctoral Summer Research program of the British Council in Saudi Arabia.

References

[1] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification, 19th International Conference, CAV, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[2] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer-Verlag, Secaucus, NJ, USA, 2007.

[4] J. W. Coleman, A. K. Malmos, P. G. Larsen, J. Peleska, R. Hains, Z. Andrews, R. Payne, S. Foster, A. Miyazawa, C. Bertolini, and A. Didierk. Compass tool vision for a system of systems collaborative development environment. In *System of Systems Engineering (SoSE), 2012 7th International Conference on*, pages 451–456, July 2012.

[5] COMPASS Consortium. COMPASS deliverable number D23.4b: CML definition 3 – timed operational semantics. Available Online, September 2013. `http://www.compass-research.eu/Project/Deliverables/D234b.pdf`.

[6] COMPASS Consortium. COMPASS deliverable number D23.5: CML definition 4. Available Online, March 2014. `http://www.compass-research.eu/Project/Deliverables/D23.5-final-version.pdf`.

[7] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[8] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[9] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.

[10] J. Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012.

[11] J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. *Algebraic Methodology and Software Technology*, pages 364–378, 2004.

[12] J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. *Information and Computation*, 231(1):38–69, 10 2013.

[13] A. Mota, A. Farias, J. Woodcock, and P. G. Larsen. Model checking cml: tool development and industrial applications. *Formal Aspects of Computing*, 27(5):975–1001, 2015.

[14] C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming*, pages –, 2016.

[15] J. Woodcock, J. Bryans, S. Canham, and S. Foster. The compass modelling language: Timed semantics in utp. In *Communicating Process Architectures 2014*. Open Channel Publishing Ltd, 2014.

[16] J. Woodcock and A. Cavalcanti. A concurrent language for refinement. In *Proceedings of the 5th Irish Conference on Formal Methods*, IW-FM'01, pages 93–115, Swindon, UK, 2001. BCS Learning & Development Ltd.

[17] J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: A formal modelling language for systems of systems. In *System of Systems Engineering (SoSE), 2012 7th International Conference on*, pages 1–6, July 2012.