Open CPS: a Symbolic Model

Farhad Arbab[1,2] and Carolyn Talcott[3]

[1] Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam
farhad@cwi.nl
[2] Leiden University, Netherlands
[3] SRI
carolyn.talcott@gmail.com

Abstract. Modeling of the environment presents a challenge in all open systems, particularly in open cyber-physical systems where the environment is nature. In some cases the environment can be considered and modeled as yet another actor or component in the system. However, in cases of complex systems with intricate interactions with nature as their environment, the unpredictability of this environment and the complexity of the physics involved in its modeling make "environment as a component" an untenable approach.
Concurrent Rules Machines (CRMs) constitute a model for formal specification and analysis of open, distributed cyber-physical systems [3]. The CRM model makes interaction with the environment explicit and offers an algebra of composition and decomposition for construction and analysis of systems through their constituent components. Systematic and automatic verification of properties of systems modeled as CRMs remains a significant challenge. Symbolic representations have been used to model the inherently continuous properties of physics. In this paper we introduce symbolic execution as a means of reasoning about behaviors of CRM models. A symbolic execution represents possibly infinitely many executions and can focus on environments meeting requirements such as physically reasonable actions. We show that symbolic execution is sound, and is complete for a natural class of CRMs. We illustrate our ideas with a simple cyber-physical system example.

Dedication. This paper is dedicated to Gul Agha in celebration of his many contributions to the understanding and development of open distributed systems.

1 Introduction

Cyber-physical systems interact with, affect, and are affected by their environment, which is often unpredictable. Not every aspect of an environment directly affects the behavior of a given CPS that runs in that environment. For instance, the thermostat component of a CPS functions correctly so long as the temperature and humidity in an environment each remains within some prescribed range, allowing the CPS to behave properly in any environment that satisfies those temperature and humidity constraints. Analogously, a concrete environment with specific ranges for its temperature and humidity may be a suitable host for a family of systems whose temperature and humidity requirements fall within those ranges.

Replacing concrete values in the model of the behavior of a CPS with symbols subject to additional constraints allows one to compactly model parameterized families of systems, reason about their behavior starting from potentially infinitely many initial states, and to represent effects of actions by the external environment subject to constraints.

Concurrent Rules Machines (CRM) constitute a model for formal specification and analysis of open, distributed cyber-physical systems [3]. The CRM model makes interaction with the environment explicit and offers an algebra of composition and decomposition for construction and analysis of systems through their constituent components. CRM semantics recognizes the environment as a distinct entity and explicitly represents its interactions with the CRM system.

The CRM model addresses the challenge of making interaction with an external environment explicit. The CRM model supports representation of systems that have both discrete and continuous behavior, which introduces another challenge: how to analyze and verify properties of systems using such models. Differential Dynamic Logic and variants support reasoning about hybrid systems and their composition, but mechanisms for representing and reasoning about unknown external environments are not considered [27,18]. Actors are another model of open systems [1]. In [2] a notion of observational equivalence was introduced to characterize actor system behavior in the context of larger systems. However the set of contexts consists of other actors. Rewriting modulo SMT has been proposed to reason about parameterized families of systems [28,22]. This allows verifying properties of possibly infinite sets of initial system configurations, where some variables may represent aspects of the environment.

Inspired by the work on symbolic rewriting, we define symbolic execution of a CRM, generalizing the CRM operational semantics. Symbolic execution provides the basis for automating analysis of properties of CRM execution in an arbitrary or possibly constrained, possibly hostile environment.

Contributions The main contributions of this paper are

- Definition of symbolic execution of a CRM parameterized by (possibly trivial) constraints on the behavior of the environment.
- A proof of soundness of symbolic execution: every instance of a symbolic execution of a CRM is a valid execution according to the operational semantics.
- Characterization of a class CRMs for which symbolic execution is complete: every execution of the operational semantics is an instance of some symbolic execution.
- The power of symbolic execution is demonstrated: using a simple example, several properties of interaction with the environment are verified using symbolic execution.

Plan In the rest of this introduction we describe our running example. In Section 2 we discuss some related work. The definition of CRM is recapped in Section 3. Symbolic execution of a CRM is defined in Section 4. In Section 5 we use our running example to illustrate how symbolic execution works, and to verify some illustrative properties. Soundness and (restricted) completeness of symbolic execution for CRMs are defined and proved in Section 6. We conclude with a summary and future work in Section 7.

1.1 Motivating examples

To illustrate CRM symbolic execution we use the water tank example introduced in [17,18] and used in [3]. A water tank has an input valve, an output valve, a water level sensor and a controller that operates the input valve. The output valve is either set to a fixed value or left to control by the environment. The water tank is characterized by global constants `WL.mx`, the maximum water level and `WL.mn`, the minimum water level. Units are chosen so that the water level dynamics obeys a simple equation

$$\texttt{wl}(t) = \texttt{wl}(t0) + (\texttt{fin} - \texttt{fout}) * (t - t0)$$

where `fin` (`fout`) is the rate of flow through the input (output) valve and ranges between 0 and 1.

The water tank model has two components, the water level sensor, and a flow controller. We consider a scenario in which the output valve is controlled by the environment. Using symbolic execution we can model the behavior of an input controller in an arbitrary environment using different control parameters.

2 Related Work

Origins of Symbolic Execution. Symbolic execution was proposed in [9] as a software testing technique. In this use of symbolic execution, programs are executed on symbolic inputs (variables) which get propagated by symbolic application of operators. The idea has been incorporated in a number of tools for testing, static analysis, and debugging. Two examples are Symbolic Java PathFinder [26] and KLEE [6,29]. Programs that interact with the external world such as through reading/writing files or reading sensors present a challenge. KLEE addresses this challenge by providing an explicit model of the external resources.

The work in this paper concerns symbolic execution of executable models of cyber-physical systems rather than code. Here the focus is on dynamic execution in unknown or unpredictable environments. An important aspect of modeling CPS is representing change over time. Treating time symbolically allows one to explore a range of sampling rates to trade efficiency of lazy sampling with greater safety obtained by frequent sampling. Another benefit of symbolic execution of CPS is identifying (ideally, the strictest set of) constraints on the environment under which an executable specification can meet its requirements, including both safety and sufficient satisfaction of task requirements. The results can be used to prevent operation in unsuitable environments, or to adapt the design to be more resilient.

Symbolic Execution in Rewriting Logic. Rewriting Logic (RWL) [19,20] is a logic for specifying and reasoning about concurrent and distributed systems. A system is specified by a rewrite theory (Σ, E, R) where Σ is an order sorted signature, E is a set of equations used to define functions and relations, and R is a set of rewrite rules. RWL and CRM share the feature that in both formalisms behavior is specified by rewrite rules that can be applied concurrently subject to non-conflict constraints.

The rewrite relation has been generalized for large classes of rewrite theories to symbolic rewriting either as rewriting modulo SMT [28] or symbolic reachability (using unification rather than matching for rule application) [23,7]. In [23] generalized rewrite theories are defined and used to develop the semantic foundations for several forms of symbolic execution. Symbolic execution supports representation and reasoning about open systems and system interaction with the environment by rewriting terms with variables. Using symbolic reachability analysis one can often reduce an infinite state space to a finite set of state patterns.

Example uses of rewriting modulo SMT include: automating verification of safe controllers for autonomous vehicles [25], semantics, verification and parameter synthesis for Parametric Timed Petri Nets [4] and Parametric Timed Automata [5]. Rewriting modulo SMT has been used to formally analyze a variety of real-time systems including PLC ST programs [14], virtually synchronous cyber-physical systems [13,12,15], and soft agents [24]. In [8] a method for combining narrowing and SMT is proposed and used to analyze parametric timed automata with additional features. This approach allows reasoning about symbolic states that have logical variables of sorts not handled by SMT.

Modeling challenges for engineering of open CPS. Lee discusses challenges to developing models for engineering cyber-physical systems in [10,11]. He identifies three classes of modeling challenges: chaotic behavior, continuous behavior, and tension between simplicity and incompleteness of deterministic models. Traditional cyber models do not deal well with continuous aspects of the physical world and continuous modeling techniques do not deal well with discreteness. Traditional abstractions do not treat concurrency or time inherently.

CRMs are a step toward addressing these modeling challenges. To model effects of the environment it is important to represent the environment explicitly. The environment has charge of time in the physical world, while CRMs can also have internal clocks. Variables of a CRM state can be discrete or continuous. The challenge is to find the right rate of sampling to not miss anything important, while being as simple as possible. Symbolic execution has potential to model chaos and to address the tension between non-determinism and simplicity.

3 CRM Background

In this section we summarize the concepts and definitions of a Concurrent Rules Machine introduced in [3]. In this summary, we slightly change some notation and refine certain concepts used in [3] to establish the necessary background to support definition of symbolic execution in Section 4.

3.1 Notation

We use $\mathbb{B}$ to denote the set of Boolean values $\{\bot, \top\}$, $\mathbb{N}_0$ to denote the set of natural numbers including zero, $\mathbb{R}_0^+$ the set of real numbers greater than or equal to zero, and

$\mathbf{D} \supseteq \mathbb{B} \cup \mathbb{N}_0 \cup \mathbb{R}_0^+$ to denote a data domain. By **Name** we denote the set of all variable names.[4] We use *fresh*(**Name**) to obtain a fresh new variable name in **Name**.

To support symbolic execution of a CRM we assume the data domain, **D**, is partitioned into constants ($\mathbf{D}_{Const}$) and symbols ($\mathbf{D}_{Sym}$). As for names, we assume a mechanism to generate fresh symbols, $fresh(\mathbf{D}_{Sym})$.

For a set X, we use $\mathcal{P}(X)$ to denote the set of all finite subsets of X, and X^k, X^*, and X^ω to denote, respectively, the set of all sequences of length $k \in \mathbb{N}_0$, all finite-length sequences, and all infinite-length sequences of elements of X. We use angular brackets to denote concrete sequences, with $\langle\rangle$ denoting the null sequence. We also treat a sequence σ of length $0 \leq k \leq \infty$ over X as a (partial) function $\sigma \in \mathbb{N}_0 \to X$, in order to refer to its $i+1^{st}$ element as $\sigma(i)$.

We use $f : D \rightsquigarrow R$ to denote f as a *pseudo function* that maps every element of the (domain) set D into a nondeterministically selected element of the (range) set R, where for a $d \in D$, two evaluations of $f(d)$ may or may not map into the same $r \in R$.

We use $Dom(f)$ and $Range(f)$ to refer to the domain and the range of a (pseudo) function f, respectively, and use $\overrightarrow{\emptyset}$ to denote the special function whose domain and range are both empty, i.e., $\overrightarrow{\emptyset} : \emptyset \to \emptyset$.

We do not specify a syntax for defining functions that appear in a CRM; we allow any well-defined mathematical notation that is convenient.

3.2 Utility functions

A *valuation function* $v : N \mapsto \mathbf{D}$ maps a subset $N \in \mathcal{P}(\mathbf{Name})$ of variable names to values $d \in \mathbf{D}$.[5] We use $\mathbf{V}$ to denote the set of all valuation functions.

The function $revise : \mathbf{V} \times \mathbf{V} \times \mathcal{P}(\mathbf{Name}) \to \mathbf{V}$, below, updates an old valuation function W with a new one, Z, excluding the latter's value bindings for the names in $N \in \mathcal{P}(\mathbf{Name})$, such that:

$$revise(W, Z, N)(x) = \begin{cases} Z(x) & \text{if } x \in Dom(Z) \setminus N \\ W(x) & \text{otherwise} \end{cases} \tag{1}$$

For convenience, by $W \triangleleft Z$ we denote the result of updating W by Z defined as:

$$(W \triangleleft Z)(x) = revise(W, Z, \emptyset)(x) \tag{2}$$

Given sets X and Y, we use the pseudo function $fold : (X \times Y \to X) \times X \times \mathcal{P}(Y) \rightsquigarrow X$ to accumulate the images of the elements of a set $S \in \mathcal{P}(Y)$ under an accumulator function $f : X \times Y \to X$ with an initial accumulator value $x \in X$.

$$fold(f, x, S) = \begin{cases} x & \text{if } S = \emptyset \\ fold(f, f(x, s), S') & \text{otherwise, where } s \in S \text{ and } S' = S \setminus \{s\} \end{cases} \tag{3}$$

[4] The CRM model supports arrays and allows $\mathbf{D} \supset \mathbf{Name}$ in order to support *indirection* (i.e., the value of a variable may be the name of another variable) and *dereferencing*. To simplify our presentation, we ignore such "dynamically computed variable names" in this paper.

[5] We assume the existence of a type system that associates a data type with each variable name in **Name** and ensures that the **D** value image of every $n \in \mathbf{Name}$ under every $v \in \mathbf{V}$ is compatible with the data type of n.

Intuitively, $fold(f, x, S)$ takes an accumulator function, f, an initial accumulator value, x, and a set, S, and accumulates the images of all elements of S under f to return an element of the same type as x. The accumulator function f itself takes two parameters (the first of the same type as that of x, and the second of the same type as that of the elements of S), and maps them to an image of the same type as x.

Observe that the choice of $s \in S$ to split $S = \{s\} \cup S'$ in each round of recursion of *fold* is nondeterministic. Therefore, every recursive path that $fold(f, x, S)$ takes to exhaustively visit all elements of S reflects a nondeterministic choice out of the $n!$ possible paths, $n = |S|$, in the set of all permutations of the elements of S, mapping each permutation, $s_1, s_2, ..., s_n$ to its image $f(...(f(f(x, s_1), s_2), ...), s_n)$. We call the images of these permutations *alleles*. Thus, generally, $fold(f, x, S)$ is a pseudo function that maps its arguments to one nondeterministically selected allele in its set of all possible image alleles of S under f. However, in the special case where $f(x, s)$ is associative and commutative within the domain of its second argument (or at least over S), the set of image alleles of $fold(f, x, S)$ becomes a singleton and we can treat *fold* as a function: regardless of the nondeterministic splittings of S, the image of $fold(f, x, S)$ is the same unique $y \in X$.

3.3 Guards

A *guard* $g = p(n_1, n_2, ..., n_k)$ is a function of $k \in \mathbb{N}_0$ arguments that given a valuation function $v \in \mathbf{V}$, maps a sequence of values $d_i = v(n_i), 0 \leq i \leq k$ to a Boolean. We denote the set of all guards as $\mathbf{G}$. For $g = p(n_1, n_2, ..., n_k)$, we use g^N to denote the arguments (read variables) of p. We use $v \models g$ to denote the evaluation of a guard g in the context of a valuation function v, which produces a Boolean value.

3.4 Actions

An *action* $a \in \mathbf{Name}^k \times (\mathbf{D}^k \to \mathbf{V})$ is a pair of a finite sequence $\langle n_1, n_2, ..., n_k \rangle$ of $k \geq 0$ argument variable names $n_i \in \mathbf{Name}, 1 \leq i \leq k$, and an *update function* u. We denote the set of all actions as $\mathbf{A}$.

For $a = (\langle n_1, n_2, ..., n_k \rangle, u)$, we use a^R to denote the set $\{n_1, n_2, ..., n_k\}$ of its *arguments* or *read* variables $n_i \in \mathbf{Name}, \forall 1 \leq i \leq k$ of a; a^U to denote the instance of u, the update function of a. We use a^W to denote the set of *write variables* of a, and refer to $a^N = a^R \cup a^W$ as the set of *variables* of a. Moreover, we use $A^W = \bigcup_{a \in A} a^W$ to denote the set of write variables of a set of actions $A \subseteq \mathbf{A}$.

Example 1. Consider action $a_1 = (\langle\rangle, \lambda()(\texttt{fin} := 1))$ where $\texttt{fin} \in \mathbf{Name}$. Then we have $a_1^R = \emptyset$ and $a_1^W = \{\texttt{fin}\}$.

Example 2. Consider action $a_2 = (\langle \texttt{wl}, \texttt{fin0}, \texttt{fout0}, \texttt{wl0}, \texttt{t0}, \texttt{t} \rangle, \lambda(wl, fin_0, fout_0, wl_0, t_0, t)(wl := wl_0 + (fin_0 - fout_0) \times (t - t_0)))$ where $\texttt{fin0}, \texttt{fout0}, \texttt{wl0}, \texttt{t0}, \texttt{t} \in \mathbf{Name}$. Then we have $a_2^R = \{\texttt{fin0}, \texttt{fout0}, \texttt{wl0}, \texttt{t0}, \texttt{t}\}$ and $a_2^W = \{\texttt{wl}\}$.

Performing an action $a = (\langle n_1, n_2, ..., n_k \rangle, u) \in \mathbf{A}$ in the context of a valuation function $v \in \mathbf{V}$ such that $a^N \subseteq Dom(v)$, applies the update function u on n_i's to yield a

new valuation function, v', where generally $Dom(v') \neq Dom(v)$ and the intersection $Dom(v') \cap Dom(v)$ may or may not be empty. We use the function $perform : \mathbf{V} \times \mathbf{A} \rightarrow \mathbf{V}$ to obtain v' by performing an action $(\langle n_1, n_2, ..., n_k \rangle, u) \in \mathbf{A}$ in the context of $v \in \mathbf{V}$:

$$v' = perform(v, (\langle n_1, n_2, ..., n_k \rangle, u)) \equiv u(v(n_1), v(n_2), ..., v(n_k))) \tag{4}$$

Observe that v', the image of *perform*(), represents merely the "change" that results from performing the action $(\langle n_1, n_2, ..., n_k \rangle, u)$ in the context of v. Intuitively, v' is *not* an "updated version" of v, but rather the "change delta" to v. We use the function *revise*() or $\triangleleft$ (Equations 1 and 2) to incorporate the image of a *perform*() into another valuation function. See the CRM semantics defined in Table 1 in Section 3.10.

Example 3. Consider action $a = (\langle d, b \rangle, u)$ where $u = \lambda(y, z)(c := y * z)$, and the valuation function v such that $\{b, d\} \subseteq Dom(v)$, where $v(b) = 6$, and $v(d) = 3$. Since $(\langle d, b \rangle, \lambda(y, z)(c := y * z))^W = \{c\}$, we have:

$$\begin{aligned} v' = perform(v, (\langle d, b \rangle, u)) \equiv \\ \lambda(y, z)(c := y * z)(v(d), v(b)) = \\ \lambda(y, z)(c := y * z)(3, 6) = \\ (c := 3 * 6) = [c \mapsto 18] \end{aligned}$$

Note that $Dom(v) \cap Dom(v') \subseteq \{c\}$, i.e., v' does not contain value bindings for all names in $Dom(v)$; it contains only the name-value bindings that performing the update function of the action changes.

We use the left-associative operator ";" to denote the sequential composition of actions, where for $a_1, a_2, a_3 \in \mathbf{A}$: $perform(v, a_1; a_2) = perform(perform(v, a_1), a_2)$, and $perform(v, a_1; a_2; a_3) = perform(perform(v, a_1; a_2), a_3)$.

3.5 Non-conflicting actions

Consider two actions $a_1, a_2 \in \mathbf{A}$ and update valuation functions $v_i = perform(v, a_i), i \in \{1, 2\}$ that result from performing them in the context of a pre valuation function $v \in \mathcal{V}$. Recall that v_i contains only those name-value bindings that performing a_i changes in the context of v (see *perform* in Section 3.4 and Example 3). In order to obtain a valuation function that reflects how performing both a_1 and a_2 changes v, we need to "update" v with v_1 and v_2 in some order using *revise* (Equation 1). In the special case where the result of updating v with v_1 and v_2 is order independent, we can perform the actions a_1 and a_2 concurrently in the context of v.

We say actions a_1 and a_2 conflict in the context of a valuation function v if $(v \triangleleft v_1) \triangleleft v_2 \neq (v \triangleleft v_2) \triangleleft v_1$, and say they are non-conflicting in the context of v, otherwise. We use *nonconflicting* : $\mathbf{A} \times \mathbf{A} \times \mathbf{V} \rightarrow \mathbb{B}$ to denote whether or not two actions conflict in the context of a specific valuation function.

Intuitively, a_1 and a_2 are non-conflicting in the context of a valuation function v if performing a_1 and a_2 concurrently in v either involves no race condition, or cannot produce different results due to a race condition. For instance, consider the two actions $a_1 \equiv y := 2 \times x$ and $a_2 \equiv y := 2 + x$ and the valuation function $v = [x \mapsto 2, y \mapsto 6]$. In

this case because $2 \times x = 2 + x = 4$, we have $v_1 = \mathit{perform}(v, a_1) = [y \mapsto 4]$ and $v_2 = \mathit{perform}(v, a_2) = [y \mapsto 4]$. Therefore $(v \triangleleft v_1) \triangleleft v_2 = (v \triangleleft v_2) \triangleleft v_1 = [x \mapsto 2, y \mapsto 4]$, which means performing a_1 and a_2 concurrently in the context of v produces the same end result regardless of (the existence of) a race condition. On the other hand, if we perform a_1 and a_2 concurrently in the context of the valuation function $v' = [x \mapsto 3, y \mapsto 6]$, we get $v_1 = \mathit{perform}(v', a_1) = [y \mapsto 6]$ and $v_2 = \mathit{perform}(v', a_2) = [y \mapsto 5]$. In this case, we have $(v' \triangleleft v_1) \triangleleft v_2 = [x \mapsto 2, y \mapsto 5]$ whereas $(v' \triangleleft v_2) \triangleleft v_1 = [x \mapsto 2, y \mapsto 6]$: the race condition between a_1 and a_2 in the context of v' does not produce a unique result.

We use $\mathit{neverconflicting} : \mathbf{A} \times \mathbf{A} \to \mathbb{B}$ to denote whether or not two actions have absolutely no conflict in the context of any valuation function.

$$\mathit{neverconflicting}(a_1, a_2) \iff (v \in \mathbf{V} \implies \mathit{nonconflicting}(a_1, a_2, v)) \tag{5}$$

For instance $\mathit{neverconflicting}(a_1, a_2)$ holds for the actions $a_1 \equiv y := 2 \times x$ and $a_2 \equiv z := 2 + x$ because there is no valuation function in the context of which performing these two actions concurrently can lead to a race condition.[6]

We use $\mathit{nonconflicting}(A, v)$ to extend the notion of non-conflicting pairs of actions to sets of actions $A \in \mathcal{P}(\mathbf{A})$. Intuitively, a set of actions $A \in \mathcal{P}(\mathbf{A})$ is non-conflicting in the context of a valuation function $v \in \mathbf{V}$ if for $a_k \in A, 1 \leq k \leq |A|$, cumulatively revising v with all $v_k = \mathit{perform}(v, a_k)$ in any sequential order yields the same final valuation function.

Formally, let $A^!$ be the set of all permutations of all elements of $A \in \mathcal{P}(\mathbf{A})$, with $n = |A|$. For $1 \leq k \leq n!$, denote ordered sequences in $A^!$ as $A^k = [a_1^k, a_2^k, ..., a_n^k] \in A^!$. The set of actions in A are non-conflicting in the context of a valuation function $v \in \mathbf{V}$ if we have:

$$\begin{array}{l} \mathit{nonconflicting}(A, v) \iff \\ \quad \left(\begin{array}{l} \forall 1 \leq i, j \leq n!, n = |A|, \\ (((v \triangleleft \mathit{perform}(v, a_1^i)) \triangleleft \mathit{perform}(v, a_2^i)) \triangleleft ...) \triangleleft \mathit{perform}(v, a_n^i) = \\ (((v \triangleleft \mathit{perform}(v, a_1^j)) \triangleleft \mathit{perform}(v, a_2^j)) \triangleleft ...) \triangleleft \mathit{perform}(v, a_n^j) \end{array} \right) \end{array} \tag{6}$$

We use $\mathit{neverconflicting} : \mathcal{P}(\mathbf{A}) \to \mathbb{B}$ to denote whether or not a set of actions have absolutely no conflict with each other in the context of any valuation function.

$$\mathit{neverconflicting}(A) \iff (v \in \mathcal{V} \implies \mathit{nonconflicting}(A, v)) \tag{7}$$

3.6 Concurrent actions

To define the valuation function that results from performing a set of actions $A \subseteq \mathbf{A}$ concurrently in the context of a valuation function v, we first define an auxiliary function. The auxiliary function $\mathit{corevise} : \mathbf{V} \to (\mathbf{V} \times \mathbf{A} \to \mathbf{V})$ maps a valuation function

[6] The concepts of non-conflicting and never-conflicting actions become more significant and less trivial with "computed names" such as array elements and indirection incorporated in the model (see Footnote 4). For instance, $\mathit{nonconflicting}(a_1, a_2, v)$ may or may not hold for $a_1 \equiv y[i] := 2 \times x$ and $a_2 \equiv y[j] := 2 + x$ depending on the value-bindings for i and j in v.

$z \in \mathbf{V}$ to another function $corevise(z) \in \mathbf{V} \times \mathbf{A} \rightarrow \mathbf{V}$ where:

$$(corevise(z))(v, a) = v \triangleleft perform(z, a) \tag{8}$$

See Equation 2 for $\triangleleft$.

Using *corevise*, we extend the definition of *perform* from performing a single action $a \in \mathbf{A}$ (Equation 4) to performing a set of actions $A \subseteq \mathbf{A}$ concurrently in the context of a valuation function v as follows:

$$perform(v, A) = fold(corevise(v), v, A) \tag{9}$$

Strictly speaking, the image of $perform(v, A)$ in Equation 9 is one nondeterministically selected alternative out of a set of the image alleles of the pseudo function *fold* (Equation 3). However, in the special case where $nonconflicting(A, v)$ holds (Equation 6), $perform(v, A)$ maps to a unique image, in which case we can treat the pseudo function $perform(v, A)$ as if it were a true function analogous to $perform(v, a)$ [3]. Note that the meaning of *perform* when applied to a set of actions in the context of a valuation function, actaully performs the update, while in the context of a single action it simply computes the update!

3.7 Sequential composition of concurrent actions

We have assumed that the syntax of actions supports sequential composition of actions $a_1, a_2 \in \mathbf{A}$, which we denote here as $a_1; a_2$ (see the text that follows Equation 4). We now extend the sequential composition of actions to sequential composition of sets of actions $A_1, A_2 \in \mathcal{P}(\mathbf{A})$, which we denote as $A_1; A_2$. Intuitively, $A_1; A_2$ is a single (composite) action that first performs all actions $a \in A_1$ in some arbitrary order, and then performs all actions $a \in A_2$ in some arbitrary order.

In Section 3.5 we showed the valuation function that results from performing the composite action $a_1; a_2$ in the context of a valuation function $v \in \mathcal{V}$. We now use Equation 9 to define the valuation function that results from performing the action $A_1; A_2$ in the context of a valuation function v as:

$$perform(v, A_1; A_2) = perform(perform(v, A_1), A_2) \tag{10}$$

Recall that the image of $perform(v, a_1)$ is an "update" valuation function for revising v, i.e., this image does not include the name-value bindings that are not changed by the action a_1 (see the paragraph after Equation 4 and Example 3). On the other hand, Equations 9 and 8 mean that the image of $perform(v, A_1)$ is a "replacement" valuation function for v (i.e., this image already includes name-value bindings that are not changed by actions in A_1, as well as those that are). For convenience, for $a \in \mathbf{A}$ and $A \in \mathcal{P}(\mathbf{A})$ we also define $perform(v, A; a) = perform(perform(v, A), \{a\})$ and $perform(v, a; A) = perform(perform(v, \{a\}), A)$.

3.8 Rules

A *rule* $r \in \mathbf{G} \times \mathcal{P}(\mathbf{A})$ is a pair of a guard and a set of concurrent actions. We denote the set of all rules as $\mathbf{R}$.

For $r = (g, A) \in \mathbf{R}$, we use r^G and r^A to denote the guard and the set of actions of r. We call $r^N = (r^A)^N$, $r^R = (r^A)^R$, and $r^W = (r^A)^W$ the *variables*, the *read-only variables*, and the *write variables* of r, respectively. Moreover, we extend our r^G, r^A, r^N, r^R, r^W notation to sets of rules in the obvious way.

A rule $r \in \mathbf{R}$ is *enabled* in the context of a valuation function $v \in \mathbf{V}$ only if $v \models r^G$ and $nonconflicting(r^A, v)$.

3.9 Concurrent Rules Machines

A Concurrent Rules Machine (CRM) is a concurrent transition system that interacts with its environment. A CRM consists of an interface, a set of initialization actions, a set of rules, and a set of termination actions [3].

Definition 1 (Concurrent Rules Machine). *A Concurrent Rules Machine (CRM) consists of a tuple* $(I, A_0, R, T) \in (\mathcal{P}(\mathbf{Name}) \times \mathcal{P}(\mathbf{Name}) \times \mathcal{P}(\mathbf{Name})) \times \mathcal{P}(\mathbf{A}) \times \mathcal{P}(\mathbf{R}) \times \mathcal{P}(\mathbf{A})$ *where:*

- $I = (Exp, Imp, Sh) \in \mathcal{P}(\mathbf{Name}) \times \mathcal{P}(\mathbf{Name}) \times \mathcal{P}(\mathbf{Name})$ *is its interface where:*
  - *Exp, Imp, and Sh are mutually disjoint,*
  - $\{envstep\} \subseteq Sh$,
- $A_0 \subseteq \mathbf{A}$ *is the finite set of its initialization actions such that* $neverconflicting(A_0)$.
- $R \subseteq \mathbf{R}$ *is its finite set of rules,*
- $T \subseteq \mathbf{A}$ *is a finite set of termination actions such that* $neverconflicting(T)$.

The sets of variable names Exp, Imp, and Sh in the interface I constitute the sets of exported, imported, and shared variable names of the CRM, respectively.

The distinguished shared Boolean variable $envstep \in Sh$ coordinates the interaction between a CRM and its environment; see Section 3.10.

We denote the set of all concurrent rules machines by **CRM**. We use C^I, C^{A_0}, C^R, and C^T to refer to the interface, the set of initialization actions, the set of rules, and the set of termination actions of a CRM $C \in \mathbf{CRM}$, respectively. We use C^{Exp}, C^{Imp}, and C^{Sh} to refer to their respective elements in C^I.

Intuitively, the exported variables in C^{Exp} are the interface variables that C *offers as read-only* variables to its environment: C, another CRM, or the environment can read the values of the variables in C^{Exp}, but only C has the right to change the values of variables in C^{Exp}. The imported variables in C^{Imp} are the interface variables that C *uses as read-only* variables: another CRM or the environment has the right to read or change the values of variables in C^{Imp}, but C may use them only as read-only variables. The shared variables in C^{Sh} are the interface variables that C *shares with* its environment for both read and write: C, another CRM, or the environment may atomically read and/or change the values of variables in C^{Sh}.

The initialization actions in C^{A_0} must be never-conflicting and they must initialize the exported interface variables in C^{Exp}. Neither initialization actions in C^{A_0} nor rules in C^R may modify the values of the read-only interface variables in C^{Imp}.

3.10 CRM Operational Semantics

Informally, a CRM C iterates indefinitely, performing the actions of some of its rules, in mutual interaction with the environment. In each round, it interacts with the environment to obtain new observation values for its imported and shared interface variables, then nondeterministically picks a subset of its enabled rules whose action sets are non-conflicting, and atomically performs those actions to revise its current valuation function for its next round (see Table 1). Meanwhile, the environment independently, perhaps continuously, changes the values of some of the interface variables of C. We use the distinguished Boolean variable *envstep* to coordinate the interaction between a CRM and its environment (see multi-steps, below)[7].

Pick $\mathcal{V}^{-1} \in \mathbf{V}$ such that $RequiredVars \subseteq Dom(\mathcal{V}^{-1})$; see Equation 11 for $RequiredVars$.

$$\mathcal{V}^i = perform(\mathcal{E}^i, S^i; T)$$

$$\mathcal{E}^i = \begin{cases} \mathcal{V}^{-1} \triangleleft [envstep \rightarrow \top] & \text{if } i = 0 \\ interact(\mathcal{V}^{i-1}, C^{Exp} \cup C^{Priv} \cup \{envstep\}) & \text{otherwise} \end{cases}$$

$$interact(V, P) = \begin{cases} revise(V, xchange(V), P) & \text{if } V(envstep) \\ V & \text{otherwise} \end{cases}$$

$$S^i = pickedacts(E^i_R, \emptyset, \mathcal{E}^i, i > 0)$$

$$pickedacts(F, S, V, I) = \begin{cases} S & \text{if } F = \emptyset \vee (I \wedge Toss()) \\ annex(A, F \setminus \{A\}, S, V) & \text{otherwise, where } A \in F \end{cases}$$

$$annex(A, F, S, V) = \begin{cases} pickedacts(F, S \cup A, V, \top) & \text{if } nonconflicting(S \cup A, V) \\ pickedacts(F, S, V, \top) & \text{otherwise} \end{cases}$$

$$E^i_R = \begin{cases} \{A_0\} & \text{if } i = 0 \\ \{A \mid (g, A) \in R, \mathcal{E}^i \models g \wedge nonconflicting(A, \mathcal{E}^i)\} & \text{otherwise} \end{cases}$$

Table 1. Semantics of a CRM $C = (I, A_0, R, T)$

Formally, we define the dynamic behavior of $C = (I, A_0, R, T) \in \mathbf{CRM}$ as an iterative revision of its valuation function, $\mathcal{V}$, where for $i \in \mathbb{N}_0$, its revised version, $\mathcal{V}^i$, for the i^{th} iteration is derived from its $i-1^{st}$ iteration version, $\mathcal{V}^{i-1}$, as defined in Table 1. The iteration starts by the environment providing a valuation function $\mathcal{V}^{-1} \in \mathbf{V}$ to initialize those interface variables of C that are not initialized by its initialization actions A_0, i.e., by picking a $\mathcal{V}^{-1}$ such that $Dom(\mathcal{V}^{-1}) \subseteq RequiredVars$ where:

$$RequiredVars = C^{Imp} \cup C^{Sh} \setminus (A_0^W \cup T^W) \tag{11}$$

[7] Observe that by Definition 1, $envstep \in C^{Sh}$ for every CRM C.

The pseudo function $Toss : \emptyset \rightsquigarrow \mathbb{B}$ in the definition of *pickedacts* nondeterministically maps to $\top$ or $\bot$. See [3] for details.

Conceptually, in every round the equations in Table 1 specify two sequential steps through which first the environment of C and then C take turns to change the values of the variables in C^I, the interface of C. For $i \in \mathbb{N}_0$, $\mathcal{E}^i$ represents the valuation function that reflects the incremental modifications that the environment makes to the values of the variables in C^I as round i begins (i.e., before C acts).

4 Symbolic Execution

Symbolic execution is a mechanism for reasoning about CRM behavior. It allows one to reason about reachability from possibly infinitely many initial states and to represent effects of action by the external environment as symbols subject to constraints.

Recall that $\mathbf{D}_{Const}$ and $\mathbf{D}_{Sym}$ represent partitions of the data domain $\mathbf{D}$. In a purely concrete execution the set $\mathbf{D}_{Sym}$ is empty, and application of functions and predicates is assumed to return a concrete value (i.e., a constant in $\mathbf{D} = \mathbf{D}_{Const}$). Thus update actions produce (concrete) update valuations. In the case of symbolic execution, applications of functions/predicates generally produce symbolic expressions (i.e., expressions involving operations applied to both constants and symbols). Actions produce updates that are valuations whose range may include symbols together with constraints on the allowed values of the symbols. Symbolic execution of a CRM transforms symbolic valuation–pairs consisting of a valuation function and a constraint on the variables in the range of the valuation via the symbolic interpretation of actions.

We use the notation of Section 3 with some added notation below to handle symbolic aspects. We use $\overline{n}$, $\overline{m}$ to denote lists of names, $m, n \in \mathbf{Name}$, and $\overline{s}$ to denote a list of symbols, $s \in \mathbf{D}_{Sym}$.

Symbolic valuations. A symbolic valuation is a pair (W, b) where $W \in \mathbf{V}$ and b is a boolean expression (over $\mathbf{D}$) constraining symbols in the range of W.

Substitutions, σ, are finite maps from symbols, $\mathbf{D}_{Sym}$, to $\mathbf{D}$. By $e[\sigma]$ we denote the usual application of a substitution σ to an expression e replacing each symbol, s, in its domain by its image, $\sigma(s)$. Applying a substitution to a valuation map $W[\sigma]$ is effectively the composition of σ and W

$$(W[\sigma])(n) = W(n)[\sigma]$$

A substitution, σ, is an instance of a symbolic valuation (W, b) if $b[\sigma]$ is true, in which case its corresponding instance valuation is $W[\sigma]$. The meaning of a symbolic valuation is the set of its instance valuations.

For a CRM to be symbolically executable automatically, we require an algorithm, such as an SMT solver, that can (partially) decide if there exists a substitution σ such that $b[\sigma]$ is true for b appearing in execution constraints ($isSat(b)$).

Symbolic guards. Let $g = (\overline{n}, P)$ be a CRM guard, where P is a boolean function. The meaning of g in the context of a symbolic valuation (W, b), written $g[[(W, b)]]$ is the symbolic (boolean) expression $P(W(\overline{n}))$. We say that (W, b) *is consistent with* g if there

is an instance σ of (W,b) such that $g[[(W,b)]][\sigma]$ is true. (W,b) *uniformly* satisfies (or just satisfies) g if $g[[(W,b)]][\sigma]$ is true for all instances σ of (W,b).

Symbolic actions. Let $a = (\overline{n}, u)$ be a CRM action where u is an update function whose range is a valuation interpreted as a valuation update. The interpretation of an action, a, in the context of a symbolic valuation, (W,b) is a symbolic valuation

$$a[[(W,b)]] = u(W(\overline{n}))$$

We require that the set of write variables of an action is independent of the valuation context. Thus we can rewrite $u(W(\overline{n}))$ as (Z^a, b^a) where $Dom(Z^a)$ is the set of write variables of a, the range, $\overline{s}$, consists of fresh symbols and b^a is equivalent to the conjunction of $s_i = u(W(\overline{n}))(n_i)$ for $s_i \in \overline{s}$.

In a CRM execution step, sets of actions are performed, not just single actions. These sets are required to be non-conflicting in the context of the valuation they are acting on (see Section 3). To apply an action set A in the context of a symbolic valuation, we require A to be non-conflicting in the context of each valuation instance. In this case, we can form the joint symbolic update of A, as follows. Let $\overline{n}_a$ be the write names of $a \in A$, and let $\overline{s}_a$ be its corresponding list of fresh symbols. Pick an order on A and let $\overline{n}$ be the concatenation of the $\overline{n}_a$ for all $a \in A$ in the chosen order, omitting names from $\overline{n}_a$ that are in $\overline{n}_{a'}$ for earlier a'. Let $\overline{s}$ be the corresponding concatenation of the $\overline{s}_a$ (omitting symbols whose names have been omitted). Then $A[[W]] = (Z^A, b^A)$ where $Z^A(\overline{n}) = \overline{s}$ and b^A is the conjunction of the constraints b^a for $a \in A$ together with equations $s_{a,i} = s_{a',j}$ if $\overline{n}_a(i) = \overline{n}'_a(j)$.

Symbolic Execution. Recall that a CRM, C, has the form (I, A_0, Rs, T) where A_0 is the set of initialization actions, $T$is the set of termination actions, and Rs is the set of rules. Symbolic Execution of C generalizes the step relation on valuation functions of a CRM to a relation on symbolic valuation functions:

$$(W,b) \Rightarrow (W',b')$$

As for concrete CRM execution, every symbolic step consists of two substeps: the environment and the CRM (sub)steps. The CRM step has two substeps: application of the actions of a candidate rule set, R, followed by application of the set of terminal actions, T.

$$\begin{aligned}(W,b) &\rightarrow_E (W^E, b \wedge b^E)\\ &\rightarrow_R (W^R, b \wedge b^E \wedge b^R)\\ &\rightarrow_T (W^T, b \wedge b^E \wedge b^R \wedge b^T)\end{aligned}$$

These symbolic substeps are explained below. We use the convention that $W^E, W^E_j, \ldots$ each represents a valuation variable denoting a valuation resulting from an environment action, and $b^E, b^E_j, \ldots$ each represents the constraint for that environment action. Similarly the superscript R or A indicates the result of actions of a rule, and the superscript T indicates the result of the set of terminal actions.

The initial step starts with an empty valuation ($W_{-1} = \emptyset$) and true constraint ($b_{-1} = \top$). We let $\overline{e}$ denote the set of names the environment is required to write. In the case of timed systems this includes the special name `time`. The environment is constrained uniformly by a constraint specification *envB* which has the form $(\overline{n}, cs)$ where cs is a set of assignments and boolean expressions over $\overline{n}$. *envB* constrains the action of an environment, thus the meaning depends on the starting valuation and the updated valuation: $envB[[W(\overline{n}), W^E(\overline{n})]]$ Where W^E is W updated by fresh symbols for $\overline{e}$.

A sequence $[(W_j, b_j), R_j \mid -1 \leq j]$ is a symbolic execution of CRM , C, (with respect to the partition of **D** and an environment constraint *envB*) if $isSat(b_{j+1})$ holds and:

$(W_j, b_j) \rightarrow_E (W_j^E, b_j \wedge b_j^E)$

$\rightarrow_{R_j} (W_j^A, b_j \wedge b_j^E \wedge b_j^A)$

$\rightarrow_T (W_j^T, b_j \wedge b_j^E \wedge b_j^A \wedge b_j^T) = (W_{j+1}, b_{j+1})$

where

$\overline{s}_j$is the j^{th} fresh symbols for $\overline{e}$

$W_{-1}(\overline{e}) = \overline{s}_{-1}$

$b_{-1}^E = envB[\overline{s}_{-1}, \overline{s}_{-1}]$

$b_j^E = envB[W_j(\overline{e}), \overline{s}_j]$

$A = if\ j < 0\ then\ A_0\ else\ acts(R_j)\ fi$

$(Z_j^A, b_j^A) = A[[W_j^E]]$

$W_j^A = W_j^E \triangleleft Z_j^A$

$(Z_j^T, b_j^T) = T[[W_j^A]]$

$W_j^T = W_j^A \triangleleft Z_j^T = W_{j+1}$

$b_j^E = envB[[W_j(\overline{n}), W_j^E(\overline{n})]]$

and R_j is a candidate rule set for $(W_j^E, b_j \wedge b_j^E)$, that is $A = acts(R_j)$ is non-conflicting and the guard of each $r \in R_j$ is true in every instance of (W_j, b_j).

We note that to simplify our presentation we have tacitly assumed the shared name *envstep* is true in all reachable valuations, thus the environment acts at every step. It is straight-forward to remove that assumption in practice.

5 Water Tank Input Controller CRM: symbolic execution

In the following we use the water tank input controller CRM, iCtl, to illustrate symbolic reasoning enabled by symbolic execution, After defining the CRM, we work out some steps of symbolic execution, and then consider some properties of the controller in an environment that obeys the physical water-level law (Section 1.1), but can change the outflow valve arbitrarily (between 0 and 1).

In the definition we use the convention that unquoted names (such as `t`, `fin0` or `WL.R`), represent (global or local) *mathematical* variables or constants, whereas quoted names (such as `'fin`, `'time`, or`'wl`), represent *CRM variable names* which will be replaced by their respective values that the CRM semantics (see Section 3.10) finds through look-up in its valuation function at run time.

Definition 2 (iCtl CRM.).

```
  iCtl = (iCtl.I, iCtl.A0, iCtl.R, iCtl.T)
where
  iCtl.I = ({'wl,'time}, {'fin}, {})
      --- the interface with imports ('wl 'time),
      --- exports ('fin)
      --- no shared variables ({}).
  iCtl.A0 = {(< >, \lambda (). 'fin := finInit)}
     --- the initialization action parameterized by finInit
  iCtl.R = { (true, {ic.fin})}
     --- the rule updating the input flow
  iCtl.T = (< 'fin >, \lambda f. 'fin0 := f)
    --- the termination action that caches value of 'fin
 where
  ic.fin= (< 'fin0, 'wl >,
           \lambda (f0,  wl).
               'fin := (if wl > WL.smx
                        then 0
                        else (if wl < WL.smn
                             then 1
                             else 1/2 fi) fi) )
```

For illustration, we fix the values of the water tank constants as follows (to keep examples small).

```
      wlInit = 3     finInit = 1
      WL.mx = 5      WL.mn = 3
      WL.smx = 9/2   WL.smn  = 7/2
```

A valuation is represented as a set of bindings (NAME := VALUE). `mt` denotes the empty valuation. The initialization and first concrete execution step of the iCtl CRM are shown below. We use => to represent the full step relation, while `-e>`, `-r>`, and `-t>` indicate the environment, rule, and terminal action substeps, respectively.

```
mt
  -e> ('time := 0) ('fout := 0) ('wl := 3)
  -r> ('time := 0) ('fout := 0) ('wl := 3) ('fin := 1)
  -t> ('time := 0) ('fout := 0) ('wl := 3) ('fin := 1) ('fin0 := 1)
=>
 -e> ('time := 1) ('fout := 1/2) ('wl := 4)
 -r> ('time := 1) ('fout := 1/2) ('wl := 4) ('fin := 1/2)
 -t> ('time := 1) ('fout := 1/2) ('wl := 4) ('fin := 1/2) ('fin0 := 1)
```

For symbolic execution of iCtl, we add a set of symbols, $\mathbf{D}_{Sym}$, to the constants of the concrete execution, and define the environment constraint $envB = (\overline{n}, cs)$. $\mathbf{D}_{Sym}$ has elements $\mathtt{v}(n, j)$ for $n \in \mathbf{Name}$ and $j \in \mathbb{N}_0$. Intuitively $\mathtt{v}(n, j)$ is the j^{th} value assigned to the name n (i.e., at the j^{th} step). The environment constraint is

```
envB =
(<'wl 'fin 'fout 'time 't0 >,
  WL.mn <= 'wl and 'wl <= WL.mx and
'wl := 'wl + ('fin - 'fout) * ('time - 't0)
)
```

Recall that *envB* is interpreted in a pair of valuation maps (W_j, W_{j+1}). The Boolean constraints are interpreted in W_{j+1} while assignment right-hand-sides are interpreted in W_j and equated to the value of the name in the left-hand-side interpreted in W_{j+1} Valuation functions are represented as in the concrete case where the range now includes symbols in addition to constants (here, real numbers).

The first two steps of symbolic execution follow.

Symbolic initialization step

```
(mtVMap , true) ==>
  -e>
 (('wl := v('wl,0)) ('time := 0) ('t0 := 0),Cstr0e)
  -r>
 (('fin := v('fim,0)) ('wl := v('wl,0)) ('time := 0) ('t0 := 0),
  Cstr0e and Cstr0r)
  -t>
  (('fin0 := v('fin,0) ('fin := v('fim,0))
   ('wl := v('wl,0)) ('time := 0) ('t0 := 0),
    Cstr0e and Cstr0r
    =  (W_0,Cstr0) )
where
Cstr0e = (v('wl,0) >= 3) and (v('wl,0) <= 5) and (v('wl,0) === initWL)
Cstr0r =  (0 <= v('fin,0)  and (v('fin,0) <= 1)
```

Symbolic step 1

```
(W_0,Cstr0)
  -e>
 (('wl := v('wl,1) ('time := 1) ('t0 := 0)
 ('fin0 := v('fin,0)) ('fin := v('fim,0)),
   Cstr0 and Cstr1e )
  -r>
 (('fin0 := v('fin,0) ('fin := v('fim,1))
  ('wl := v('wl,1) ('time := 1) ('t0 := 0),
   Cstr0 and Cstr1e and Cstr1r )
 -t>
 (('fin0 := v('fin,1) ('fin := v('fim,1))
```

```
  ('wl := v('wl,1) ('time := 1) ('t0 := 0),
   Cstr0e and Cstr0r and Cstr1e and Cstr1r )
where
Cstr1e ~~
  ( v('wl,1) >= 3 and v('wl,1) <= 5 and
    v('wl,1) === v('wl,0) + (v('fin,0) - v('fout,0)) * 1 )
Cstr1r =
  (vvWLCi.fin#1:Real === (0/1).Real
  or
   vvWLCi.wl#1:Real <= (7/2).Real) and
   vvWLCi.fin#1:Real === (1/1).Real
  or
   vvWLCi.wl#1:Real < (9/2).Real and
   (7/2).Real < vvWLCi.wl#1:Real and
     vvWLCi.fin#1:Real === (1/2).Real)
```

Symbolic Reasoning Two simple examples of the questions one can ask of iCtrl using symbolic execution are: Starting with water level 3 (minimum):

(1) Can the water level go above `WL.smx`?
(2) Can the water level reach `WL.mx`?

The answer to the first question is yes. Here is one concrete solution:

```
(v('fin,0) := 31/32) (v('fout,0) := 1/32) (v('wl,0) := 3)
(v('fin,1) := 1/2)   (v('fout,1) := 1/16) (v('wl,1) := 63/16)
(v('fin,2) := 1/2) (v('fout,2) := 1/8) (v('wl,2) := 35/8)
(v('fin,3) := 0)   (v('fout,3) := 1/2) (v('wl,3) := 19/4)
```

The answer to the second question is up to time 12, no. However if we change the time step to be increments of 2, then the answer is yes. Here is a witness

```
(v('fin,0) := 1)  (v('fout,0) := 0 (v('wl,0) := 3
(v('fin,1) := 0)  (v('fout,1) := 1/2) (v('wl,1) := 5
```

This emphasizes that the choice of time interval matters. To investigate effects of the choice of time interval we can turn that interval into a symbol, `'dt`. If we constrain `'dt` to be in the interval $[1/2, 2/1]$ and ask question 2 again, there are several solutions depending on the upper-bound on number of steps. Here is one witness where the step upper-bound is 6.

```
v('dt,0) := 7/4
v('fin,0) := 1/2   v('fout,0) := 1/4   v('wl,0) := 3
v('fin,)1 := 1  v('fout,1) := 3/28  v('wl,1) := 55/16
v('fin,2) := 0  v('fout,2) := 1/2  v('wl,2)  := 5
```

If we ask question 2 with `'dt` constrained to be in the interval $[1/2, 1/1]$ there is no solution up to 12 steps. An interesting question for future work is: can we find k such that no solution up to k steps implies there is no solution. The idea is to find a constraint on states such that after k steps that constraint holds again, yielding an induction condition. See [25] for an example of such reasoning.

6 Soundness and Partial Completeness

For symbolic execution to be useful as a reasoning method, it is important that it gives accurate information about CRM behavior. For example, is an instance of a symbolic trace a trace of the CRM according to the operational semantics (soundness)? Conversely, if there is no symbolic trace instance violating a given property, might there be a trace of the operational semantics that violates that property (completeness)?

In the following we prove soundness for all CRMs that admit symbolic execution. We also prove completeness for CRMs that satisfy a simple additional condition.

6.1 Soundness

Soundness says that given a symbolic trace of a CRM with environment constrained by *envB* any instance of (a finite prefix of) this trace is a concrete trace of that CRM.

An instance of a symbolic trace

$$[(W_j, b_j) \rightarrow_{E_j,R_j} (W_{j+1}, b_{j+1}) \mid -1 <= j < m]$$

is a sequence

$$[V_j \rightarrow_{F_j,R_j} V_{j+1} | -1 <= j < m]$$

given by an instance σ of b_m where

$$[V_j = W_j[\sigma] \mid -1 <= j < m]$$

and F_j given by the equations of b_j^E and and σ. [8]

Note that by the definition of instance and the monotonic nature of the constraints b_j, if σ is an instance of b_m then σ is an instance of b_j for $j <= m$.

Theorem 1. *Each instance of a symbolic trace of a CRM is a (concrete) trace of that CRM.*

Proof. Fix a symbolic trace as above and instance substitution σ. Consider step j

$$(W_j, b_j) \rightarrow_{E_j,R_j} (W_{j+1}, b_{j+1})$$

of the symbolic trace for some $j < m$. Note that σ is an instance of b_j and of b_{j+1}. We show that the corresponding instance step is a valid (concrete) CRM step.

By definition, the rule set of the j^{th} step, R_j, is a valid choice of concurrent rule set for a CRM step. We show that the instance of each substep of step j is a valid substep.

Finally, as noted above $isSat(b_{j+1})$ holds with witness σ.

[8] The expression $(W_j, b_j) \rightarrow_{E_j,R_j} (W_{j+1}, b_{j+1})$ represents the result of the three parts of a CRM step, parameterized by the environment update E_j, a rule set R_j, and the terminal actions T, left implicit as it is the same actions for each step.

Environment substep

$$(W_j, b_j) \rightarrow_{E_j} (W_j^E, b_j \wedge b_j^E)$$

where $Z_j^E(\overline{e}) = \overline{s}_j$(fresh symbols), $W_j^E = W_j \triangleleft Z_j^E$, and $b_j^E = envB[[W_j^E]]$.

The instance substep is

$$V_j \rightarrow_{F_j} V_j^E$$

where F_j is given by the equations of b_j^E (corresponding to the assignments of *envB*) and σ; $V_j = W_j[\sigma]$ and $V_j^E = W_j \triangleleft Z_j^E[\sigma] = V_j \triangleleft F_j$. A valid environment step.

Rule action substep

$$(W_j^E, b_j \wedge b_j^E) \rightarrow_{R_j} (W_j^R, b_j \wedge b_j^E \wedge b_j^R)$$

where $W_j^R = W_j^E \triangleleft Z_j^R$, A_j is the union of action sets of R_j $(Z_j^R, b_j^R) = A_j[[W_j^E]]$

At the concrete level, $V_j = W_j[\sigma]$ and by definition of symbolic trace step, the guards of rules in R_j are true in V_j and the joint action set is non-conflicting in the context of V_j. Also letting $V_j^R = V_j^E \triangleleft Z_{A_j}$, we have $V_j^R = W_j^R[\sigma] = W_j^E \triangleleft Z_j^R$.

Termination action substep The termination step is

$$(W_j^R, b_j \wedge b_j^E \wedge b_j^R) \rightarrow_T (W_j^T, b_j \wedge b_j^E \wedge b_j^R \wedge b_j^T) = (W_{j+1}, b_{j+1})$$

The argument for termination action set case is the same as for rule action set case.

6.2 Restricted Completeness

Completeness of symbolic execution holds under under some conditions on the rules of a CRM. For example CRMs where rule guards depend only on names that are always bound to constants, the write variables of a rule is independent of valuation context, and no two rules write the same variable. These conditions hold for many classes of CRM.

Definition 3 (Symbolically complete CRM).

A CRM is symbolically complete if given any (reachable) symbolic valuation (W, b)*, and substitution,* σ*, that satisfies* b*; if rule set* R *is a candidate for execution in the context of* $W[\sigma]$ *(all guards are true and the union of their action sets is non-conflicting) then* R *is a candidate for execution in the context of* $W[\sigma']$ *for any* σ' *satisfying* b*.*

A CRM in which rule guards depend on variables whose values are always constants (not symbolic) and the write variables of actions in action sets within or across rules are independent of valuation and disjoint, is symbolically complete. This is true for a large class of CRMs modeling CPSs.

Lemma 1 (One (sub)step completeness).

If CRM, C, is symbolically complete, (W, b) *a reachable symbolic valuation,* σ *an instance of* (W, b) *(*σ *satisfies* b*),* $V = W[\sigma]$ *the corresponding valuation instance, and* A *a nonconflicting action set.* [9] *then*

$$V \rightarrow_R V' \textit{ implies } (W, b) \rightarrow_A (W', b')$$

[9] A could be an environment action set, the initialization actions A_0, the actions of a rule set R of rules enabled in V, or the set of termination actions T.

where V' is an instance of (W', b').

Proof. The idea is the following. Let $\overline{m} = A^W$ (the write variables of A), $U = A[V]$ the joint update function of actions of A in V, and $Z, b^a = A[[W]]$, the joint symbolic update function of A in W. Let σ' be defined by

$$\sigma'(W'(\overline{m})) = V'(\overline{m}) = U(\overline{m})$$

then $b' = b \wedge b^a$,

$$b^a = \wedge_{m_i \in \overline{m}}(W'(m_i) = U(m_i))$$

and $\sigma + \sigma'$ satisfies b' is the desired instance mapping (W', b') to V'

Theorem 2 (Restricted Completeness).
If CRM, C, is symbolically complete, and

$$Tr = V_j \rightarrow_{F_j, R_j} V_{j+1} \mid -1 \leq j < m$$

is a concrete trace of C such that F_j satisfies envB for $-1 \leq j < m$, then there is a symbolic trace, Tr^*, of C with environment constraint envB such that Tr is an instance of Tr^*.

Proof. Given *Tr*, we show that there is a symbolic trace Tr^* firing the same sequence of rule sets as *Tr*, and *Tr* is an instance of Tr^*.

We first show that the initialization step ($j = -1$) of *Tr* is an instance of the initialization step of C, which is the same for any symbolic trace (constrained by *envB*). Then we show that for any $j > 0$, assuming by induction that we have Tr^*_j with *Tr* up to j an instance, then Tr^*_j can be extended using R_j, so that *Tr* is an instance up to $j + 1$.

Initialization step. We show that the initialization step of *Tr* is an instance of the initialization step of Tr^*. *Tr* initialization has 3 substeps.

$$\begin{aligned}
&\rightarrow_{F_0} V^E = F_0 \\
&\quad \rightarrow_{A_0} V^A = V^E \triangleleft U^A \\
&\quad \rightarrow_T V^T = V^A \triangleleft U^T = V_1
\end{aligned}$$

where F_0 is an update valuation with domain the environment required variables $U^A = A_0[[V^E]]$, and $U^T = T[[V^A]]$.

The symbolic initialization has three symbolic substeps

$$\begin{aligned}
&\rightarrow_{E_0} (W^E, b^E) \\
&\quad \rightarrow_{A_0} (W^E \triangleleft Z^A, b^E \wedge b^A) = (W^A, b^{EA}) \\
&\quad \rightarrow_T (W^A \triangleleft Z^T, b^{EA} \wedge b^T = (W_0, b_0)
\end{aligned}$$

where $(Z^A, b^A) = A_0[[W^E]$, and] $(Z^T, b^T) = T[[W^A]]$.

Define the substitution, σ_0, making *Tr* initialization an instance of Tr^* initialization as follows.

For $n \in dom(E_0) = Dom(F_0)$, $\sigma_0(E_0[n]) = F_0[n]$. For $n \in dom(Z^A)$, $\sigma_0(Z^A[n]) = V^A[n]$. For $n \in dom(Z^T)$, $\sigma_0(Z^T[n]) = V^T[n]$. Note that the sets $dom(E_0)$, $dom(Z^a)$, $dom(Z^T)$ are disjoint due to freshness of symbols at each substep.

By definition, $V_0 = W_0[\sigma_0]$. It remains to show that σ_0 satisfies b_0. σ_0 satisfies b^E due the requirement that environment actions satisfy *envB*. σ_0 satisfies b^A and σ_0 satisfies b^T by Lemma 1.

Step $j \geq 0$. Assume the jth stop of Tr is

$$\begin{aligned} V_j &\rightarrow_{F_j} V_j \triangleleft F_j = V_j^E \\ &\rightarrow_{R_j} V_j^E \triangleleft Y_A = V^A \\ &\rightarrow_T V_j^A \triangleleft Y_T = V_j^T = V_{j+1} \end{aligned}$$

where the F_j substep satisfies *envB*.

Assume by induction that (W_j, b_j) is the corresponding symbolic state and σ_j the substitution instance such that σ_j satisfies b_j and $W_j[\sigma_j] = V_j$.

Then by symbolic completeness R_j is enabled in (W_j, b_j) so

$$\begin{aligned} (W_j, b_j) &\rightarrow_{E_j} (W_j \triangleleft E_j, b_j \wedge b_j^E) = (W_j^E, b_j \wedge b_j^E) \\ &\rightarrow_{R_j} (W_j^E \triangleleft Z^A, b_j \wedge b_j^E \wedge b_j^A) = (W_j^A, b_j^{EA}) \\ &\rightarrow_T (W_j^A \triangleleft Z^T, b_j^{EA} \wedge b_j^T) = (W_{j+1}, b_{j+1}) \end{aligned}$$

where $dom(E_j) = dom(F_j)$, $rng(E_j)$ is a set of fresh symbols, $b_j^E = envB[W_j, W_j^E]$, A_j is the joint actions of R_j, $(Z_j^A, b^A) = A_j[[W_j^E]]$, and $(Z_j^T j, b^T) = T[[W_j^A]]$.

By Lemma 1 V_{j+1} is an instance of $(W_{j+1} b i_{j+1})$ using an extension of σ_j to new symbols.

7 Conclusion

In this paper we defined symbolic execution of Concurrent Rules Machines, a model of open distributed systems with explicit interaction with the environment. We show that symbolic execution is sound and it is complete for a large class of CRMs. We illustrated the utility of symbolic execution by verification of properties of interaction of a simple water level controller with the environment.

This is just the beginning. There are several directions for future work.

Compositionality of CRMs and their view of the environment suggest that symbolic execution of CRM models is also a promising approach to compositionally reason about the properties of larger systems through property-preserving symbolic composition of their constituent subsystems.

The reasoning examples in this paper were worked out using a first version of an implementation of symbolically executable CRMs using Rewriting modulo SMT. Once a robust implementation is available, we intend to carry out substantial case studies.

An alterntative to reasoning symbolically about interactions with the environment, is to assign probability distributions to effects of actions (exported variables) and to

observations by a CRM (import variables), and use mechanisms such a probablistic or statistical model checking to reason about system behavior and effects on the environment. This is also an important approach to investigate. To the best of our knowledge there is no theory supporting a sound and computable combination of symbolic and probablisstic/statistical reasoning. However, the formal patterns/theory transformation approach used for analyzeing Maude executable models allows one to understand analysis results in the context of a common base model [21,16].

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
2. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
3. Farhad Arbab and Carolyn Talcott. Concurrent rules machines. In *Rebeca for Actor Analysis in Action: Essays in the Honour of Marjan Sirjani*, volume 15560 of *LNCS Festschrift*. Springer, 2025.
4. J Arias, K. Bae, C. Olarte, P. C. Ölveczky amd L. Petrucci, and F Rømming. Symbolic analysis and parameter synthesis for time petri nets using maude and smt solving,. *arXive*, 2023. arXiv:2303.08929.
5. J Arias, K. Bae, C. Olarte, P. C. Ölveczky amd L. Petrucci, and F Rømming. Symbolic analysis and parameter synthesis for networks of parametric timed automata with global variables using maude and smt solving. *Science of Computer Programming*, 2024.
6. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
7. Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn L. Talcott. Equational unification and matching, and symbolic reachability analysis in maude 3.2 (system description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 529–540. Springer, 2022.
8. S. Escobar, R. López-Rueda, and J. Sapiña. Symbolic analysis by using folding narrowing with irreducibility and smt constraints. In *9th ACM SIGPLAN International Workshop on Formal Techniques for Safety- Critical Systems (FTSCS 2023)*, pages 14–25. ACM, 2023.
9. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
10. E. A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE, 2008.
11. E. A. Lee. Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, 2016.
12. J. Lee, K. Bae, S. Kim, M. Kang, and P. C. Ölveczky. Modeling and formal analysis of virtually synchronous cyber-physical systems in aadl. *International Journal on Software Tools for Technology Transfer*, 24(6):911–948, 2022.
13. J. Lee, K. Bae, and P. C. Ölveczky. An extension of hybridsynchaadl and its application to collaborating au- tonomous uavs. In *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning (ISoLA 2022)*, LNCS, page 47–64. Springer, 2022.

14. J. Lee, S. Kim, and K. Bae. Bounded model checking of plc st programs using rewriting modulo smt. In *8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2022)*, pages 56–67. ACM, 2022.
15. J. Lee, S. Kim, K. Bae, and P. C. Ölveczky. Hybridsynchaadl: Modeling and formal analysis of virtually synchronous cpss in aadl. In *Computer Aided Verification*, LNCS, page 491–504. Springer, 2021.
16. Si Liu, José Meseguer, Peter Csaba Ölveczky, Min Zhang, and David A. Basin. Bridging the semantic gap between qualitative and quantitative models of distributed systems. *Proc. ACM Program. Lang.*, 6(OOPSLA2):315–344, 2022.
17. Simon Lunel, Benoît Boyer, and Jean-Pierre Talpin. Compositional proofs in differential dynamic logic dl. In *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*, pages 19–28, 2017.
18. Simon Lunel, Stefan Mitsch, Benoit Boyer, and Jean-Pierre Talpin. Parallel composition and modular verification of computer controlled systems in differential dynamic logic. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 354–370, Cham, 2019. Springer International Publishing.
19. J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
20. José Meseguer. Twenty years of rewriting logic. *J. Log. Algebraic Methods Program.*, 81(7-8):721–781, 2012.
21. José Meseguer. Taming distributed system complexity through formal patterns. *Sci. Comput. Program.*, 83:3–34, 2014.
22. José Meseguer. Generalized rewrite theories, coherence completion, and symbolic methods. *J. Log. Algebraic Methods Program.*, 110, 2020.
23. José Meseguer. Generalized rewrite theories, coherence completion, and symbolic methods. *J. Log. Algebraic Methods Program.*, 110, 2020.
24. V. Nigam and C. Talcott. Automating safety proofs about cyber-physical systems using rewriting modulo smt. In *Rewriting Logic and Its Applications*, LNCS, page 212–229. Springer, 2022.
25. Vivek Nigam and Carolyn L. Talcott. Automating recoverability proofs for cyber-physical systems with runtime assurance architectures. In Cristina David and Meng Sun, editors, *Theoretical Aspects of Software Engineering - 17th International Symposium, Proceedings*, volume 13931 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2023.
26. Corina S Pasareanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20:391–425, 2013.
27. Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *Int J Software Tools Technology Transfer*, 18:67–91, 2016.
28. Camilo Rocha, José Meseguer, and César A. Muñoz. Rewriting modulo SMT and open system analysis. *J. Log. Algebraic Methods Program.*, 86(1):269–297, 2017.
29. Ying Zhang, Peng Li, Yu Ding, Lingxiang Wang, Dan Williams, and Na Meng. Broadly enabling klee to effortlessly find unrecoverable errors in rust. In *ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice*. ACM/IEEE, 2024.