Concurrent Rules Machines

Farhad Arbab[1,2] and Carolyn Talcott[3]
December 14, 2024

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

Abstract. Cyber-Physical systems interact with their environment in complex ways. In addition to exchanging information with other systems, they affect and are affected by their physical / natural environment. Their components run concurrently and may be physically distributed, participating in both synchronous and asynchronous interactions. Existing formal models typically model closed systems, or limited interactions among system components. In this paper we introduce the Concurrent Rules Machine (CRM), a formal model of system behavior that makes explicit interaction with the environment. We define an algebra of operations for CRM composition and decomposition along with equational properties. We illustrate our ideas with a simple cyber-physical system example.

Dedication. This paper is dedicated to Marjan Sirjani in celebration of her technical and leadership contributions to advancing computer science.

1 Introduction

Cyber-Physical Systems (CPSs) are increasingly ubiquitous. They come in many sizes ranging from autonomous vehicles to tiny medical devices. They may be low risk, or highly safety critical. CPSs interact with, affect, and are affected by their environment, which is often unpredictable. The components of a CPS are inherently open; they execute concurrently, interact both synchronously and asynchronously, and they may be geographically distributed.

A foundational formal framework for developing CPS models should support representation and reasoning about:

- open systems, including interaction with unknown/unpredictable environments
- discrete and continuous change (cyber and physical)
- diverse interactions among components including synchronous and asynchronous discrete interactions, as well as continuous interactions such as flow of a physical quantity among components (e.g., force, energy, liquid, etc.)
- composition operations that model these different interactions
- concurrent and distributed execution
- methods to support scaling in time and space such as symbolic reasoning and abstraction; and compositional design and reasoning

Existing models cover different features. Hybrid system models [30,29,33,7,13] address the combination of discrete and continuous behavior elements (control and plant). Their focus, however, is generally on closed systems. Automata and process algebra models support concurrent or distributed execution and composition with a mix of synchronous and asynchronous interaction, typically as action labels on transitions. [16,10,1,24,15,34,3,4,5,6] The usual composition results in a blowup of model size. The environment, if considered explicitly, is viewed as *other components*, a deficient model of the physical world.

Rewriting logic (RWL) is an expressive formalism for specifying concurrent and distributed systems [25,12]. There is a rich algebra of modules (theories), that can be lifted to composition of specific models (terms). Basic rewriting operates on closed systems. Extensions to symbolic rewriting provide an important step towards specifying and reasoning about open system [31]. RWL has no notion of initial state or specific system of interest. We can think of a ground term *tm* as representing some entity with a (possibly trivial) behavior, given by the rewrite rules. The rewriting process generates one possible behavior (or multiple possible behaviors if the strategy is not deterministic). Unconstrained search generates all possible behaviors. Since there is no pre-determined notion of system or component, RWL has no explicit mechanism for system/component composition beyond basic term formation using constructors. By localness of rule application, term formation induces a corresponding composition of behaviors of subterms and may add behavior due to rules for the top level constructor.

In this paper, we introduce *Concurrent Rules Machines* (CRMs) as a formal model primarily intended for specification and analysis of open, distributed, cyber-physical systems. A CRM is an abstract machine with distributed state and concurrent rule-based transitions that makes the interaction between a system and its environment explicit. The concurrent semantics of CRMs makes them practically compositional, avoiding exponential explosion of their representation under composition. Intuitively, the basic product of two CRMs produces a resulting CRM whose behavior consists of all permissible interleavings and parallel behaviors of the two CRMs.

Contributions. The main contributions of this paper are the following:

- We introduce the Concurrent Rules Machine (CRM) model of computation, defining the structure of a CRM and its operational semantics. A more abstract observational semantics is derived from the operational semantics.
- In the CRM model, interaction with the environment is explicit. This enables reasoning about open systems and supports multiple notions of composition reflecting different forms of component interaction.
- Multiple models of time can be represented in CRMs as well as behaviors in which time is implicit.
- An algebra of CRM components is defined, including several composition operations, and a division relation for reasoning about decomposition. Structural properties of these operations are also presented.
- Importantly, composition scales in the sense that the size of a composed CRM is roughly the sum of the sizes of its component CRMs.
- A simple cyberphysical system is used to illustrate CRM concepts and several algebraic operations.

1.1 Running Example

To motivate and illustrate key elements of CRMs we borrow and adapt the water system example of [22,23]. There are two kinds of component, (i) a water tank with water entering at an inflow port, `fin`, and draining via an outflow port, `fout`; and (ii) control components that adjust flow rates: inflow or outflow, or both. We start by considering an inflow controller and assume that the outflow is either fixed at a constant rate, or controlled by some unknown environment entity.

The water tank interface has 2 imports (`fin`, `fout`) and one export (`wl`). Variables `fin`, `fout` are rates of water flowing in/out of the tank. Variable `wl` is the water level. In order to compute the water level at a given step the water tank uses the flows assigned in the previous step. Variables `fin`, `fout` are constrained to values in the real interval [0,1] and `wl` ranges from `WLmin` to `WLmax`. `WLmin` is the lowest level reachable by draining, could be 0 or positive. `WLmax` is the overflow point. For simplicity we assume units chosen so that $\texttt{wl}' = \texttt{fin} - \texttt{fout}$, where $\texttt{wl}' = \texttt{dwl/dt}$. Thus if `fin` and `fout` are constant over the interval `[t0,t]` then:

$\texttt{wl}(\texttt{tx}) = \texttt{wl}(\texttt{t0}) + (\texttt{fin} - \texttt{fout}) \times (\texttt{tx} - \texttt{t0})$ for `tx` in `[t0,t]`.

The inflow controller imports a water level reading, `wl`, and exports the inflow rate, `fin`. We assume the controller reads `wl` at time `t` and sets `fin`, which will determine the change in water level until time $\texttt{t1} > \texttt{t}$ when it repeats this action. Note that the inflow controller does not have access to the outflow rate beyond what it might infer from the inflow and change in `wl` over time. As a concrete example, assume the inflow controller has two parameters `WLminSafe` and `WLmaxSafe` with $0 \leq \texttt{WLmin} \leq \texttt{WLminSafe}$ and $\texttt{WLminSafe} < \texttt{WLmaxSafe} \leq \texttt{WLmax}$. When $\texttt{wl} \geq \texttt{WLmaxSafe}$ it sets `fin` to 0; and when $\texttt{wl} \leq \texttt{WLminSafe}$ it sets `fin` to 1; otherwise it leaves `fin` unchanged.

Intuitively the water tank component models a physical component while the controller models a cyber component. They interact by the controller reading sensor informant produced by the water tank and the water tank behavior being dependent on flow rates set by a controller.

Plan. The remainder of the paper is organized as follows. In Section 2 we introduce notation and concepts used. Sections 3,4, and 5 introduce key elements of a CRM: Guards, Actions, and Rules. The CRM structure is defined in Section 6 and its semantics is given in Section 7. Section 8 presents the operations of an algebra of CRMs and Section 9 discusses the representation of time. Related work is discussed in Section 10. We conclude with a summary and discussion of future work in Section 11.

2 Basic concepts

2.1 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 *new*(**Name**) to obtain a fresh new variable name in **Name**.

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.

2.2 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 v_1 with a new one, v_2, excluding the latter's value bindings for the names in $N \in \mathcal{P}(\mathbf{Name})$, such that:

$$revise(v_1, v_2, N)(x) = \begin{cases} v_2(x) \text{ if } x \in Dom(v_2) \setminus N \\ v_1(x) \text{ otherwise} \end{cases} \tag{1}$$

Observe that *revise* has the property of idempotency:

$$revise(revise(v_1, v_2, N), v_2, N) = revise(v_1, v_2, N)$$

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 $x \in X$.

$$fold(f, x, S) = \begin{cases} x & \text{if } S = \emptyset \\ fold(f, f(x, s), S') & \text{otherwise, where } s \in S \text{ and } S' = S \setminus \{s\} \end{cases} \tag{2}$$

[4] We allow $\mathbf{D} \supset \mathbf{Name}$, to support *indirection* (i.e., the value of a variable may be the name of another variable) and *dereferencing*.

[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 all elements of S through 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 $y \in X$.

We use the function $foldalleles : (X \times Y \to X) \times X \times \mathcal{P}(Y) \to \mathcal{P}(X)$ to denote the set of all alleles of the pseudo function $fold : (X \times Y \to X) \times X \times \mathcal{P}(Y) \rightsquigarrow X$:

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

Observe that by definition, *foldalleles* has the following properties:

1. $|foldalleles(f, x, S)| \geq 1$
2. $|foldalleles(corevise(v), v, A')| \leq |foldalleles(corevise(v), v, A)|$ for action sets $A' \subseteq A$ and valuation functions v.

Intuitively, $fold(f, x, S)$ merely picks a nondeterministically chosen element out of the set $foldalleles(f, x, S)$. We formalize this relationship in Lemma 1.

Lemma 1. *$fold(f, x, S) \in foldalleles(f, x, S)$; and $\forall x \in foldalleles(f, x, S)$, x is an image allele of $fold(f, x, S)$.*

Proof. We outline a proof. Let $S^!$ be the set of all permutations of all elements in S, with $n = |S|$. Denote ordered sequences in $S^!$ as $S^k = [s_1^k, s_2^k, ..., s_n^k]$, $1 \leq k \leq n!$. Structurally, $fold(f, x, S)$ recursively traverses one permutation $S^k \in S^!$ for a nondeterministic value of $1 \leq k \leq n!$ and returns $f(...f(f(x, s_1^k), s_2^k)..., s_n^k)$. On the other hand, $foldalleles(f, x, S)$ traverses all $S^k \in S^!$, $1 \leq k \leq n!$ and returns $\bigcup_{1 \leq k \leq n!} \{f(...f(f(x, s_1^k), s_2^k)..., s_n^k)\}$.

The significance of Lemma 1 is that in the special case where $foldalleles(f, x, S)$ is a singleton, then the image of $fold(f, x, S)$ is the same unique element in this singleton, regardless of the sequence of nondeterministic choices that $fold(f, x, S)$ makes.

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. For example, let $g_1 = p(\texttt{wl}) = \texttt{wl} \leq \mathit{WLmin}$, where $\texttt{wl} \in \mathbf{Name}$ and $\mathit{WLmin} \in \mathbf{D}$ represents some (symbolic) value. Then $g_1(v) = \{\texttt{wl}\}$.

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. For example, if $v(\mathit{pvwl}) = 20$ and the constant WLmin is instantiated to 35 then $v \models g_1$ is true.

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* $\lambda(p_1, p_2, ..., p_k)\mathit{lambexp}$. We denote the set of all actions as $\mathbf{A}$.

For $a = (\langle n_1, n_2, ..., n_k \rangle, u)$ and $v \in \mathbf{V}$, we use $a^R(v)$ to denote the set $\{m_1, m_2, ..., m_k\}$ of its *arguments* or *read* variables $m_i \in \mathbf{Name}, \forall 1 \leq i \leq k$ of a in the context of v; $a^U(v)$ to denote the instance of u, the update function of a, in the context of v; and $a^W(v)$ to denote the set of *write variables* of a in the context of v, defined as $\mathit{Dom}(a^U(v)(v(n_1), v(n_2), ..., v(n_k)))$.

We refer to $a^N(v) = a^R(v) \cup a^W(v)$ as the set of *variables* of a in the context of v. Also, we extend our $a^U(v), a^N(v), a^R(v), a^W(v), a^N, a^R$, and a^W notation from individual actions to sets of actions in the obvious way, e.g., the write variables of a set of actions $A \subseteq \mathbf{A}$ is $A^W(v) = \bigcup_{a \in A} a^W(v)$, etc.

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

Example 2. Consider action $a_2 = (\langle \texttt{wl}, \texttt{fin0}, \texttt{fout0}, \texttt{wl0}, \texttt{t0}, \texttt{t} \rangle, \lambda(\mathit{wl}, \mathit{fin}_0, \mathit{fout}_0, \mathit{wl}_0, t_0, t)(\mathit{wl} := \mathit{wl}_0 + (\mathit{fin}_0 - \mathit{fout}_0) \times (t - t_0)))$ where $\texttt{fin0}, \texttt{fout0}, \texttt{wl0}, \texttt{t0}, \texttt{t} \in \mathbf{Name}$. Then for all $v \in \mathbf{V}$ 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 $a^N(v) \subseteq \mathit{Dom}(v)$, applies the update function u on n_i's to yield a new valuation function, v', where generally $\mathit{Dom}(v') \neq \mathit{Dom}(v)$ and the intersection $\mathit{Dom}(v') \cap \mathit{Dom}(v)$ may or may not be empty. We use the function $\mathit{perform} : \mathbf{V} \times \mathbf{A} \to \mathbf{V}$ to obtain v' by performing an action $(\langle n_1, n_2, ..., n_k \rangle, u) \in \mathbf{A}$ in the context of $v \in \mathbf{V}$:

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

For convenience, we gloss over formal details of lambda expressions and use the typical syntax of imperative programming languages for assignment statements and expressions, if-then-else and loop constructs, and sequential composition to represent actions.

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*() (Equation 1) to incorporate the image of a *perform*() into another valuation function. See the CRM semantics defined in Table 1 in Section 7.

Example 3. Consider action $a = (\langle c, d, b\rangle, u)$ where $u = \lambda(x, y, z)(x := y * z)$, with $v(b) = 6$, and $v(d) = 3$. Since $(\langle c, d, b\rangle, \lambda(x, y, z)(x := y * z))^W = \{c\}$, we have:

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

Definition 1 (Equivalent actions). *Two actions $a_1, a_2 \in \mathbf{A}$ are equivalent, denoted as $a_1 \equiv a_2$, if and only if for all valuation functions $v \in \mathbf{V}$ we have $\mathit{perform}(v, a_1) = \mathit{perform}(v, a_2)$.*

4.1 Non-conflicting actions

Consider two actions $a_1, a_2 \in \mathbf{A}$ and the execution of their sequential composition $a_1; a_2$ in the context of a pre valuation function $v \in \mathcal{V}$. In this case, first a_1 executes in the context of v and its execution produces an "update" valuation function $u_1 = \mathit{perform}(v, a_1)$. Next, a_2 executes in the context of the valuation function $u' = \mathit{revise}(v, u_1, \emptyset)$ and produces the "update" valuation function $u_2 = \mathit{perform}(u', a_2)$. Finally, revising u' using u_2 yields the post valuation function $u'' = \mathit{revise}(u', u_2, \emptyset)$.

Conceptually, in contrast, concurrent execution of a_1 and a_2 in the context of v starts at exactly the same time when both actions "observe" the exact same pre valuation function v. Once both executions complete, they yield two "update" valuation functions $v_i = \mathit{perform}(v, a_i), i \in \{1, 2\}$. Now depending on the order in which we apply v_1 and v_2 to cumulatively revise v, we obtain one of the two post valuation functions $v'_i = \mathit{revise}(\mathit{revise}(v, v_i, \emptyset), v_{3-i}, \emptyset)$,where $v'_1 = v'_2$ may or may not hold. We say actions a_1 and a_2 conflict in the context of a valuation function v if $v'_1 \neq v'_2$, and say they are non-conflicting in the context of v, otherwise.

The significance of non-conflicting actions is that not only they can be executed concurrently, regardless of the order in which their respective resulting "update" valuation functions are used to revise their common pre valuation function, we obtain the same unique post valuation function. We use $\mathit{nonconflicting} : \mathbf{A} \times \mathbf{A} \times \mathbf{V} \to \mathbb{B}$ to denote whether or not two actions conflict in the context of a specific valuation function, where:

$$\begin{aligned} &\mathit{nonconflicting}(a_1, a_2, v) \iff \\ &\quad \mathit{revise}(\mathit{revise}(v, v_1, \emptyset), v_2, \emptyset) = \mathit{revise}(\mathit{revise}(v, v_2, \emptyset), v_1, \emptyset) \\ &\text{where: } v_i = \mathit{perform}(v, a_i), i \in \{1, 2\} \end{aligned} \tag{5}$$

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.

$$\mathit{neverconflicting}(a_1, a_2) \iff (v \in \mathbf{V} \implies \mathit{nonconflicting}(a_1, a_2, v)) \tag{6}$$

4.2 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 $corevise : \mathbf{V} \to (\mathbf{V} \times \mathbf{A} \to \mathbf{V})$ maps a valuation function $z \in \mathbf{V}$ to another function $corevise(z) \in \mathbf{V} \times \mathbf{A} \to \mathbf{V}$ where:

$$(corevise(z))(v, a) = revise(v, perform(z, a), \emptyset) \tag{7}$$

See Equation 1 for *revise*.

We use $nonconflicting(A, v)$ to extend the notion of non-conflicting pairs of actions in Equation 5 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 starting with the pre valuation function v, applying the "update" valuation functions that result from performing all actions in A to cumulatively revise v in any sequential order yields the same post valuation function.

Formally, the set of actions in A are non-conflicting in the context of a valuation function $v \in \mathbf{V}$ if we have:

$$nonconflicting(A, v) \iff |foldalleles(corevise(v), v, A)| = 1 \tag{8}$$

Analogously, we define $conflicting(A, v)$ to denote:

$$conflicting(A, v) \iff |foldalleles(corevise(v), v, A)| > 1 \tag{9}$$

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

$$neverconflicting(A) \iff (v \in \mathcal{V} \implies nonconflicting(A, v)) \tag{10}$$

Lemma 2. *For $v \in \mathcal{V}$ and $B \subseteq A \subseteq \mathbf{A}$, $nonconflicting(A, v) \implies nonconflicting(B, v)$.*

Proof. From property 2 of *foldalleles* in Equation 3 we have

$$B \subseteq A \implies |foldalleles(corevise(v), v, B)| \leq |foldalleles(corevise(v), v, A)|$$

By Equation 8 we have $|foldalleles(corevise(v), v, A)| = 1$. Thus, $|foldalleles(corevise(v), v, B)| \leq 1$. From property 1 of *foldalleles* in Equation 3 we have $|foldalleles(corevise(v), v, B)| \geq 1$. Therefore $|foldalleles(corevise(v), v, B)| = 1$.

Using *corevise*, we now extend the definition of *perform* from performing an 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:

$$perform(v, A) = fold(corevise(v), v, A) \tag{11}$$

Strictly speaking, the image of $perform(v, A)$ in Equation 11 is one nondeterministically selected alternative out of a set of the image alleles of the pseudo function *fold* (Equation 2). However, in the special case where $nonconflicting(A, v)$ holds, by Lemma 1 and Equation 8 $perform(v, A)$ maps to a unique image, which justifies treating the pseudo function $perform(v, A)$ as if it were a true function analogous to $perform(v, a)$.[6]

Lemma 3. *For $z, v \in \mathbf{V}$, and $a, a_1, a_2, a_3 \in A \subseteq \mathbf{A}$ we have:*

1. *idempotency:*

$$(corevise(z))((corevise(z))(v, a), a) = (corevise(z))(v, a) = v$$

and moreover, if $nonconflicting(A, v)$, then:

2. *associativity:*

$$(corevise(v))((corevise(v))((corevise(v))(v, a_1), a_2), a_3) = \\ (corevise(v))((corevise(v))((corevise(v))(v, a_2), a_3), a_1)$$

3. *commutativity:*

$$(corevise(v))((corevise(v))(v, a_1), a_2) = (corevise(v))((corevise(v))(v, a_2), a_1)$$

Proof. We sketch a proof as follows.

1. idempotency: Follows from Equation 7 and the idempotency property of *revise* in Equation 1.
2. associativity: Let $v_l = (corevise(v))((corevise(v))((corevise(v))(v, a_1), a_2), a_3)$ and $v_r = (corevise(v))((corevise(v))((corevise(v))(v, a_2), a_3), a_1)$. Observe that $\{v_l, v_r\} \subseteq foldalleles(corevise(v), v, B)$ (see Equation 3). We have $B = \{a_1, a_2, a_3\} \subseteq A$ and $nonconflicting(A, v)$. By Lemma 2, then, we have $nonconflicting(B, v)$, which by Equation 8 means $|foldalleles(corevise(v), v, B)| = 1$. Therefore $v_l = v_r$.
3. commutativity: Similar to the argument for the case of associativity, above.

Recall that generally $perform(v, a)$ is neither associative nor commutative nor idempotent over a set of actions A (see Equation 4). By its definition (Equation 7), then, $(corevise(z))(v, A)$ is also neither associative nor commutative over A. Lemma 3, however, establishes that $(corevise(z))(v, A)$ is idempotent; and if $nonconflicting(A, v)$ holds then it is also associative and commutative.

[6] Observe that whereas $v' = perform(v, a)$ for action $a \in \mathbf{A}$ evaluates to an "update change" to v, the evaluation of $v'' = perform(v, A)$ for $A \subseteq \mathbf{A}$ yields a "replacement" for v. Conceptually, an update change to v differs from a replacement for v in that on the one hand, generally, the relation $Dom(v) \subseteq Dom(v')$ may (or may not) hold. On the other hand, $Dom(v) \subseteq Dom(v'')$ always holds because *revise* is monotonic, i.e., for $u' = revise(u, z, N)$ we have $u' \supseteq u$. Thus, a replacement can also harmlessly be considered an update change, i.e., just as for $w = revise(v, v', \emptyset)$ we have $w = revise(v, w, \emptyset)$, we also have $v'' = revise(v, v'', \emptyset)$. Therefore, we ignore their subtle distinction and treat both $perform(v, a)$ and $perform(v, A)$ as expressions that evaluate to an update change for v.

4.3 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 4.1 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 11 to define the valuation function that results from performing the action $A_1; A_2$ in the context of a valuation function v as:

$$\mathit{perform}(v, A_1; A_2) = \mathit{perform}(\mathit{perform}(v, A_1), A_2) \tag{12}$$

Observe that whereas the image of $\mathit{perform}(v, a_1)$ is an "update" valuation function for revising v, the image of $\mathit{perform}(v, A_1)$ is a "replacement" valuation function for v, and therefore $\mathit{revise}(v, \mathit{perform}(v, A_1)) = \mathit{perform}(v, A_1)$. See Footnote 6.

For convenience, we also define $\mathit{perform}(v, A; a) = \mathit{perform}(\mathit{perform}(v, A), \{a\})$ and $\mathit{perform}(v, a; A) = \mathit{perform}(\mathit{perform}(v, \{a\}), A)$, where $a \in \mathbf{A}$ and $A \in \mathcal{P}(\mathbf{A})$.

5 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. For $v \in \mathbf{V}$, we call $r^N(v) = (r^A)^N(v), r^R = (r^A)^R$, and $r^W(v) = (r^A)^W(v)$ the *variables*, the *read-only variables*, and the *write variables* of r, respectively. We call $r^S(v) = r^N(v) \cup r^W(v) \cup (r^G)^N$ the *scope* of a rule r, and $r^S = \bigcup_{v \in \mathbf{V}} r^S(v)$, the *superset of the scope* of r. Moreover, we extend our $r^G, r^A, r^N(v), r^R, r^W(v), r^S(v)$ notation to sets of rules in the obvious way, e.g., the scope and the superset of the scope of a set of rules $R \subseteq \mathbf{R}$ are $R^S(v) = \bigcup_{r \in R} r^S(v)$ and $R^S = \bigcup_{r \in R} r^S$, respectively.

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 $\mathit{nonconflicting}(r^A, v)$.

Example 4. An inflow controller rule for the case of low water level has a guard that checks that the level is below the safe minimun. When the guard is true, the rule is enabled and `fin` is set to 1.

```
((<'wl>,\lambda wl. wl <= WLminSafe), {(< >,\lambda (). 'fin := 1)}))
```

Definition 2 (Equivalent rules). *Two rules $r_i = (g_i, A_i), i \in \{1, 2\}$ are equivalent, denoted as $r_1 \equiv r_2$, if and only if $g_1 \iff g_2 \wedge A_1 \equiv A_2$.*

We also extend the notion of equivalence from individual rules to sets of rules. We say two sets of rules $R_i, i \in \{1, 2\}$ are equivalent, denoted as $R_1 \equiv R_2$, if and only if $r_i \in R_i \implies \exists r_{3-i} \in R_{3-i}$ such that $r_1 \equiv r_2$.

6 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.

Definition 3 (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 7.

We denote the set of all concurrent rules machines by **CRM** and use $\emptyset_{\mathbf{CRM}}$ to denote the empty concurrent rules machine $((\emptyset, \emptyset, \emptyset), \emptyset, \emptyset, \emptyset)$. 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. We call $C^{W_0} = (A_0)^W$ the *initialized variables*, $C^S = (C^R)^S \cup (A_0)^R \cup C^{W_0} \cup T^W \cup C^{Imp} \cup C^{Sh}$ the *scope* of C, $C^{Rd} = (C^R)^R \cup (A_0)^R \cup C^{Imp} \cup C^{Sh}$ the *read* variables of C, $C^W = C^{W_0} \cup (C^R)^W \cup T^W$ the *write* variables of C, and $C^{Priv} = C^S \setminus (C^{Exp} \cup C^{Imp} \cup C^{Sh})$ the *private* or local variables of C. We use C^{Ro} to denote the set of *read-only* variables of C, defined as $C^{Ro} = C^S \setminus C^W$.

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}.

Definition 3 restricts the scope of the rules in C^R to the set of interface variables in C^I and the (local) variables initialized by the initialization actions in C^{A_0}. 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}.

Example 5 (Water tank CRM). The water tank component of our running example (Section 1.1) can be modeled as a CRM, WL (water level), in the pseudo-code below. In this (and other pieces of) pseudo code, we use the convention that unquoted names (such as `t`, `fin0` or `WL.R`), represent (global or local) *mathematical variables*, whereas quoted names (such as `'fin`, `'time`, or`'wl`), represent *CRM program variables (variable names)* which will be replaced by their respective values that the CRM semantics (see Section 7) finds through look-up in its valuation function at run time.

To be concrete, we fix the global parameters as follows: `WLmin` = 1, `WLminSafe` = 2, `WLmaxSafe` = 3, and `WLmax` = 4. We assume that initially the water level is at `WLminSafe`. The water level CRM computes the level at the current time (the value of `'time`), given the water level at the previous time, `t0`, and the control settings at `t0`. [7] When the water level rule executes, the environment has already overwritten `'time`, and the control variables. Thus at each step these values need to be cached for the next step. To do so, we introduce private variables `'t0` (to store the previous time), to know how much time has passed, and `'fin0` and `'fout0` (to store the values of `'fin` and `'fout` at the previous time). These variables are set by termination actions. Note that there is no need to cache the value of `'wl` as it will have the value at the previous time when the water level rule executes.

```
  WL = (WL.I, WL.A0, WL.R, WL.T)
where
  WL.I = ({'fin, 'fout, 'time}, {'wl},{})
  --- the interface with imports ('fin 'fout 'time), exports ('wl)
      and no shared variables ({}).
  WL.A0 = {(< >, \lambda (). 'wl := WCminSafe)}
     --- the initialization action
  WL.R = { (true, {a.wl})}
     --- the rule updating the water level
  WL.T = {(< 'time, 'fin, 'fout >,
           \lambda (t, fin, fout).
               {'t0 := t,'fin0 := fin, 'fout0 := fout} )}
          --- the termination action that caches the
              values of 'fin 'fout and 'time
 and
  a.wl = (< 'fin0,'fout0,'t0,'time , 'wl >,
            \lambda (fin0, fout0, t0, t, wl0).
                'wl := wl0 + (fin0 - fout0) * (t - t0) )
```

7 CRM 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

[7] In cyber physical models, we assume the environment provides a notion of time (see ***time*** in Section 9). The imported variable `'time` in our pseudo-code represents ***time*** in Section 9.

are non-conflicting, and atomically performs those actions to revise its current valuation function for its next round. 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)[8].

Pick $\mathcal{V}^{-1} \in \mathbf{V}$ such that $RequiredVars \subseteq Dom(\mathcal{V}^{-1})$; see Equation 13 for *RequiredVars*.

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

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

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

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

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

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

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

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

7.1 CRM Operational Semantics

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:

$$RequiredVars = C^{Imp} \cup C^{Sh} \setminus (A_0^W \cup T^W) \tag{13}$$

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

[8] Observe that by Definition 3, $envstep \in C^{Sh}$ for every CRM C.

the variables in C^I, the interface of C.[9] 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).

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

Example 6 (Sample WL execution). The listing below gives the sequence of valuation maps of a possible execution of WL when the environment provides time in unit increments (and values of `'fin,'fout`). First the environment writes initial values for `time` (= 0), `'fin` = 1, and `'fout` = 0. The the initialization actions `WL.A0` of WL are executed (`'wl` = 2.0) followed by the termination actions `WL.T` that cache the values of `'time,'fin,'fout` as `'t0'fin,'fout`. This is followed by alternations between the environment action and execution of the rule `WL.R`.

```
time fin  fout  wl   t0  fin0  fout0
  0   1    .5                        Env initiates
                2.0   0    1      .5   WL A0;T
  1   0    1                         Env writes
                2.5   1    0      1    WL,R ; WL.T
  2   1    0                         Env writes
                1.5   2    1      0    WL,R ; WL.T
  3   1    0                         Env writes
                2.5   3     1     0    WL,R ; WL.T
  .....
```

We can also execute symbolically. Suppose at time 0 the water level is wl_0 and we fix `fout` to be 0.75. Let fin_j be the value of `'fin` at time j . Then wl_j, the water level at time j is given by

$$wl_1 = wl_0 + (fin_0 - 0.75)$$

$$wl_2 = wl_1 + (fin_1 - 0.75) = wl_0 + fin_0 + fin_1 - 1.5$$

$$\ldots$$

$$wl_{j+1} = wl_0 + \Sigma_{i\in[0,j]} fin_i + (j+1) \times 0.75$$

Initialization. Initially as round 0 begins, the environment of C must pick an arbitrary valuation function $\mathcal{V}^{-1}$ to give values to at least the set of variables in *RequiredVars* defined in Equation 13.

For $i = 0$ we get $\mathcal{E}^0 = revise(\mathcal{V}^{-1}, [envstep \to \top], \emptyset)$, where $[envstep \to \top] \in \mathbf{V}$ is a function that maps the distinguished variable *envstep* to $\top$[10] (see Equation 1) Thus we

[9] Of course, in every turn, C may change its own private variables, as well. However, these changes are not directly visible outside of C; the environment will indirectly observe the impact of changes to local variables of C as/if they alter the values of the export and shared interface variables of C.

[10] The default value of $\top$ for *envstep* in $\mathcal{V}^0$ means that by default, Table 1 ensures that successive steps of the behavior of C are interleaved with updates of its interface variables to reflect the

have $\mathcal{E}^0(envstep) = \top$ and for all $x \in Dom(v) \setminus \{envstep\}$ we have $\mathcal{E}^0(x) = \mathcal{V}^{-1}(x)$. Then through $perform(\mathcal{E}^i, S^i; T)$ in $\mathcal{V}^i = perform(\mathcal{E}^i, S^i; T), C$ performs for $i = 0$ the set $S^0 = E_R^0 = A_0$ of its initialization actions, sequentially followed by the set of termination actions T, within the context of the valuation function $\mathcal{E}^0$ to yield the valuation function for the next round (see Equation 12). This ends round $i = 0$.

Observe that for $i > 0$, depending on the nondeterministic evaluation of *Toss*(), the function $pickedacts(E_R^i, \emptyset, \mathcal{E}^i, i > 0)$ may evaluate to $\emptyset$ regardless of whether or not $E_R^i = \emptyset$. This observation means that a CRM can always refrain from performing the actions of any of its enabled rules in a round $i > 0$. However, the evaluation of $S^0 = pickedacts(E_R^0, \emptyset, \mathcal{E}^0, 0 > 0)$ shows that $S^0 = \emptyset$ only if $E_R^0 = A_0 = \emptyset$. In other words, a CRM cannot "skip" performing its initialization actions in round $i = 0$.

Multi-steps. In Table 1, the initial valuation function $\mathcal{E}^0$ sets the value of the distinguished private variable *envstep* to $\top$. Therefore, unless an action in A_0 changes the value of *envstep*, we have $\mathcal{V}^0(envstep) = \mathcal{E}^0(envstep) = \top$.

For $i > 0$, the valuation function $\mathcal{E}^i$ is obtained through (possible) interaction with the environment, formalized as function $interact : \mathbf{V} \times \mathcal{P}(\mathbf{Name}) \rightarrow \mathbf{V}$ in Table 1. The value of $\mathcal{V}^{i-1}(envstep)$ determines whether (1) *interact* yields $\mathcal{V}^{i-1}$ (if $\mathcal{V}^{i-1}(envstep) = \bot$), in which case C takes another turn without interference by the environment, in the context of the valuation function $\mathcal{E}^i = \mathcal{V}^{i-1}$, or (2) the environment takes a turn (if $\mathcal{V}^{i-1}(envstep) = \top$). Thus, by setting the value of the distinguished private variable *envstep*, C can decide to take one or more steps between every two successive steps by the environment.

Interaction with the environment. If $\mathcal{V}^{i-1}(envstep) = \top$ for some $i > 0$, the environment takes its turn to incrementally modify the values of some of the interface variables in C^I, an activity that we represent by the pseudo function $xchange : \mathbf{V} \rightsquigarrow \mathbf{V}$.[11] We use the function *revise* (see Equation 1) to incorporate these incremental modifications by the environment into $\mathcal{V}^{i-1}$ to produce $\mathcal{E}^i$, the valuation function in the context of which C must determine and perform its set of enabled actions S^i in its next iteration for i.

Generally, nothing prevents the environment from changing the values of the export variables of C (which C offers as read-only to the environment) or the local private variables of C (which only C must be able to modify). To account for this undesirable fact, we allow the valuation function that is the image of $xchange(\mathcal{V}^{i-1})$ to possibly map some of the variable names in $C^{Exp} \cup C^{Priv}$ to new values. In order to preserve the integrity of the *protected* variables in $C^{Exp} \cup C^{Priv}$, however, the definition of $\mathcal{E}^i$ in Table 1 excludes their modification through the second argument of *interact* (which serves as

changes in the environment. Of course, C can change this default by changing the value of *envstep* in its initialization actions or by the actions in its rules.

[11] Observe that having *xchange* as a pseudo function imposes no restriction on how the environment may behave; the behavior of the environment may neither be computable, nor expressible as a function. Regardless of how/why nature/physics changes certain properties of the environment, intuitively, *xchange* serves as a *sensor reading function* that modifies the values of some of the interface variables of a CRM, based on its snapshot of, e.g., the energy level of a battery, the velocity of a moving object, the water-level in a tank, ambient humidity, temperature, etc.

the third argument of *revise* in the definition of *interact*). This exclusion guarantees that for all $i > 0$ and $x \in C^{Exp} \cup C^{Priv}$ we have $\mathcal{E}^i(x) = \mathcal{V}^{i-1}(x)$.

Enabled actions. For $i \in \mathbb{N}_0$, the set S^i contains the enabled actions that C decides to perform in its i^{th} round of recursion. The set S^i is recursively constructed by mutual recursion of functions *pickedacts* and *annex*, which together pick a nondeterministic subset of E_R^i. The set E_R^i contains the sets of actions of the enabled rules in round i.

For $i = 0$, E_R^0 contains the set of initialization actions of the CRM C, i.e., $E_R^0 = \{A_0\}$, and we have $S^0 = \mathit{pickedacts}(\{A_0\}, \emptyset, \mathcal{E}^0, \bot)$. Since the condition $\{A_0\} = \emptyset \vee \bot \wedge \mathit{Toss}()$ in the definition of *pickedacts* evaluates to $\bot$ and $A \in \{A_0\} \implies A = A_0$, we have $\mathit{pickedacts}(\{A_0\}, \emptyset, \mathcal{E}^0, \bot) = \mathit{annex}(A_0, \{A_0\} \setminus \{A_0\}, \emptyset, \mathcal{E}^0)$. By definition, $\mathit{neverconflicting}(A_0) = \top$. Thus $\mathit{annex}(A_0, \{A_0\} \setminus \{A_0\}, \emptyset, \mathcal{E}^0) = \mathit{pickedacts}(\emptyset, A_0, E_R^0, \top)$. Consequently, the condition $\emptyset = \emptyset \vee (\top \wedge \mathit{Toss}())$ in the definition of *pickedacts* evaluates to $\top$ and we have $\mathit{pickedacts}(\emptyset, A_0, E_R^0, \top) = A_0$. Therefore, $S^0 = A_0$.

For $i = 0$, as we saw above, what *Toss*() evaluates to is irrelevant. For $i > 0$, the nondeterministic image of *Toss*() together with the nondeterminism inherent in the choice of $A \in F$ steer the mutual recursion of *pickedacts* and *annex* to nondeterministically pick a "flattened" subset of the action sets in E_R^i that are non-conflicting in the context of the valuation function $\mathcal{E}^i$, as the set S^i of the enabled actions to be performed in round i. Note that the actions in S^i may belong to different rules whose guards evaluate to true. In other words, C does *not* select one enabled rule to perform its actions; it selects a subset of its enabled rules and performs the actions in their action sets. Thus, depending on whether or not C selects one of its enabled rules r, C performs either all or none of the actions in the actions set of r. Note also that C may "idle" in a round if for some $i > 0$, we have $E_R^i = \emptyset$ or if evaluating $\mathit{pickedacts}(E_R^i, \emptyset, \mathcal{E}^i, i > 0)$, *Toss*() nondeterministically evaluates to $\top$.

Observe that in Table 1 if $\mathcal{V}^{i-1}(\mathit{envstep}) = \top$ for $i > 0$, then we have $\mathcal{E}^i = \mathit{revise}(\mathcal{V}^{i-1}, \mathit{xchange}(\mathcal{V}^{i-1}), C^{Exp} \cup C^{Priv})$. This means that even if $S^i = \emptyset$, through $\mathit{xchange}(\mathcal{V}^{i-1})$, the environment may change the name-value bindings in $\mathcal{V}^{i-1}$, from which we obtain $\mathcal{V}^i$. In other words, interaction with the environment may allow C to progress in round i regardless of whether or not C idles in round $i - 1$.

7.2 CRM behavior

Observables. Initially, the environment may set the values of all variables in $C^{Imp} \cup C^{Sh}$. The recursive definitions in Table 1 state that in round $i > 0$ the environment can change the values of the variables in the domain of the valuation function that is the image of $\mathit{xchange}(\mathcal{V}^{i-1})$. These changes affect the behavior of C only if they involve the interface variables in $C^{Imp} \cup C^{Sh}$. So, the set of relevant variables whose values the environment changes in rounds $i > 0$ is $(C^{Imp} \cup C^{Sh}) \cap \mathit{Dom}(\mathit{xchange}(\mathcal{V}^{i-1}))$. Combining the two cases, we conclude that the set of C variables whose values the environment changes in the i^{th} round is:

$$M_E^i = (C^{Imp} \cup C^{Sh}) \cap (i = 0 \; ? \; \mathbf{Name} : \mathit{Dom}(\mathit{xchange}(\mathcal{V}^{i-1}))), i \in \mathbb{N}_0$$

From Table 1 we derive the set of variables whose values C may change in the i^{th} round as the write-variables of the set of actions that C performs in round i:

$$M_C^i = T^W \cup \bigcup_{A \in S^i} A^W(v), i \in \mathbb{N}_0 \tag{14}$$

Therefore, the set of all C variables whose values may change (are written) between the ends of rounds $i-1$ and i is $M^i = M_E^i \cup M_C^i$ or:

$$M^i = ((C^{Imp} \cup C^{Sh}) \cap (i = 0 \; ? \; \mathbf{Name} : Dom(xchange(\mathcal{V}^{i-1})))) \cup M_C^i, i \in \mathbb{N}_0 \tag{15}$$

Binding values to the variables in M^i constitute the difference between the mappings of $\mathcal{V}^{i-1}$ and $\mathcal{V}^i$. Observe that M^i can be partitioned into four disjoint sets of variables: C^{Exp}, C^{Imp}, $C^{Sh} \setminus M_C^i$, and $M_C^i \setminus (C^{Exp} \cup C^{Sh})$. Only C can change the values of its internal variables in $M_C^i \setminus (C^{Exp} \cup C^{Sh})$ and those in C^{Exp}, and only the environment can change the values of the variables in C^{Imp} and $C^{Sh} \setminus M_C^i$. But both C and the environment may change the values of the variables in $M_C^i \cap C^{Sh}$ in the i^{th} round.

Nevertheless, there is no race condition between C and the environment over their shared variables in $M_C^i \cap C^{Sh}$. Table 1 states that C performs its actions that make changes to the values of the variables in M_C^i (through $fold(corevise(\mathcal{E}^i), \mathcal{E}^i, S^i)$) only *after* the environment has made its possible changes to the values of the variables in C^{Sh} to produce $\mathcal{E}^i$. Therefore in the i^{th} round, (1) the values that C reads for the shared variables in $M_C^i \cap C^{Sh}$ are what the environment has set for them in $\mathcal{E}^i$, and (2) in $\mathcal{V}^i$, any change that C makes to the values of the shared variables in $M_C^i \cap C^{Sh}$ overrides the values that the environment may have bound to these variables. Moreover, by Equation 8, regardless of the nondeterministic choices made by *fold* (see Equations 11 and 12), the image of $perform(\mathcal{E}^i, S^i; T)$ is the same unique $\mathcal{V}^i$.

We use O^i to denote the *observable* of the i^{th} round, which we define as the set of name-value pairs $(n, v) \in \mathbf{Name} \times \mathbf{D}$ for every variable in M^i that acquires a new value through actions of either the environment or C in round i:

$$O^i = \bigcup_{n \in M^i} \{(n, \mathcal{V}^i(n))\}, i \in \mathbb{N}_0 \tag{16}$$

Behavior. We identify a *run* of a CRM C as an infinite sequence $\langle O^0, O^1, O^2, ... \rangle \in (\mathcal{P}(\mathbf{Name} \times \mathbf{D}))^\omega$ of observables O^i. The *CRM behavior* of C, denoted as $\mathcal{B}(C)$, consists of the set of all of its runs.

Lemma 4. *For* $C^i = ((E^i, I^i, S^i), A_0^i, R^i, T^i) \in \mathbf{CRM}, i \in \{1, 2\}$*:*

$$\begin{pmatrix} E^1 = E^2 \wedge I^1 = I^2 \wedge S^1 = S^2 \\ \wedge \\ A_0^1 \equiv A_0^2 \wedge R^1 \equiv R^2 \wedge T^1 \equiv T^2 \end{pmatrix} \implies \mathcal{B}(C^1) = \mathcal{B}(C^2)$$

8 An algebra of CRMs

8.1 CRM containment, union, and intersection

For $C_1, C_2 \in \mathbf{CRM}$, we write $C_1 \subseteq C_2$ if and only if C_2 component-wise contains C_1, that is, if and only if $(C_1^{Exp} \subseteq C_2^{Exp}) \wedge (C_1^{Imp} \subseteq C_2^{Imp}) \wedge (C_1^{Sh} \subseteq C_2^{Sh}) \wedge (C_1^{A_0} \subseteq C_2^{A_0}) \wedge (C_1^{R} \subseteq C_2^{R}) \wedge (C_1^{T} \subseteq C_2^{T})$.

Analogously, we define $C_1 \cup C_2$ and $C_1 \cap C_2$ for the structures that result from, respectively, the union and the intersection of the corresponding structural components of the tuples that form C_1 and C_2.

Properties of containment, union, and intersection.

- For all $C_1, C_2 \in \mathbf{CRM}$, their intersection $C = C_1 \cap C_2 \in \mathbf{CRM}$.
- For all $C_1, C_2 \in \mathbf{CRM}$, their union $C = C_1 \cup C_2 \in \mathbf{CRM}$ if and only if:
  - The three sets $C_1^{Exp} \cup C_2^{Exp}$, $C_1^{Imp} \cup C_2^{Imp}$, and $C_1^{Sh} \cup C_2^{Sh}$ are mutually disjoint;
  - $(C_1^{A_0} \cup C_2^{A_0})^W \cap (C_1^{Imp} \cup C_2^{Imp}) = \emptyset$;
  - $\mathit{neverconflicting}(C_1^{A_0} \cup C_2^{A_0})$; and
  - $\mathit{neverconflicting}(C_1^{T} \cup C_2^{T})$.
- CRM intersection is associative, commutative, and idempotent.
- CRM union is associative, commutative, and idempotent.
- For $C \in \mathbf{CRM}$:
  - $C \cup \emptyset_{\mathbf{CRM}} = C$
  - $C \cap \emptyset_{\mathbf{CRM}} = \emptyset_{\mathbf{CRM}}$

8.2 CRM product

Informally, the product of two CRMs is a CRM whose behavior is the same as the behavior of the concurrent execution of the two CRMs. As such, the product of two CRMs is very much like their union, except that the union of two CRMs is not always a CRM (see Paragraph 8.1), whereas the product of every two CRMs is always a CRM (see properties of product, below).

Definition 4 (CRM product). *Let $C_1, C_2 \in \mathbf{CRM}$ such that their initialization and termination actions do not conflict, and they share no private variables:*

- $\mathit{neverconflicting}(C_1^{A_0} \cup C_2^{A_0})$.
- $\mathit{neverconflicting}(C_1^{T} \cup C_2^{T})$.
- $C_1^{Priv} \cap C_2^{Priv} = \emptyset$.

The product of C_1 and C_2, denoted as $C_1 \times C_2$ is $C = (C^I, C_1^{A_0} \cup C_2^{A_0}, C_1^{R} \cup C_2^{R}, C_1^{T} \cup C_2^{T})$ where:

$$
\begin{aligned}
C^I &= (C^{Exp}, C^{Imp}, C^{Sh}) \\
C^{Exp} &= C_1^{Exp} \cup C_2^{Exp} \\
C^{Imp} &= (C_1^{Imp} \cup C_2^{Imp}) \setminus (C^{Exp} \cup C^{Sh}) \\
C^{Sh} &= (C_1^{Sh} \cup C_2^{Sh}) \setminus C^{Exp}
\end{aligned}
$$

Intuitively, the product of two CRMs $C_i \in \mathbf{CRM}, i \in \{1,2\}$, consists of their union after their conflicting elements are symmetrically removed. Conflicts of interface variables are resolved following the policy that export takes precedence over shared which in turn takes precedence over import. Specifically, any $x \in C_i^{Exp}$ that C_i exports remains an export interface variable in $C_1 \times C_2$. If C_i exports a variable $x \in C_i^{Exp} \cap (C_{3-i}^{Imp} \cup C_{3-i}^{Sh})$ that C_{3-i} either imports or shares in its interface, then x remains an export interface variable in the product CRM. If C_i shares a variable $x \in C_i^{Sh} \cap C_{3-i}^{Imp}$ that C_{3-i} imports in its interface, then x remains a shared interface variable in the product CRM.

Properties of product.

- For all $C_1, C_2 \in \mathbf{CRM}$ their product $C = C_1 \times C_2 \in \mathbf{CRM}$.
- CRM product is associative, commutative, and idempotent.
- For all $C \in \mathbf{CRM}$ we have $C \times \emptyset_{\mathbf{CRM}} = C$.
- For all $C_1, C_2 \in \mathbf{CRM}$ if $C_1 \cup C_2 \in \mathbf{CRM}$ then their product and union have:
  - the same behavior: $\mathcal{B}(C_1 \times C_2) = \mathcal{B}(C_1 \cup C_2)$
  - the same interface: $(C_1 \times C_2)^I = (C_1 \cup C_2)^I$

8.3 Hiding

Hiding is an operation that removes an interface variable of a CRM. An interface variable x of a CRM $C \in \mathbf{CRM}$ can be hidden only if x is among the variables that C initializes, i.e., only if $x \in C^{A_0^W}$. Thus, because $C^{Imp} \cap C^{A_0^W} = \emptyset$, a variable x of C can be hidden only if $x \in C^{Exp}$ or $x \in C^{Sh} \cap C^{A_0^W}$.

Definition 5 (Hiding). *By $\exists[x]C$ we denote hiding a variable $x \in (C^{Exp} \cup C^{Sh}) \cap C^{A_0^W}$ of a $C \in \mathbf{CRM}$, where $\exists[x]C = (I, C^{A_0}, C^R, C^T)$ and $I = (C^{Exp} \setminus \{x\}, C^{Imp}, C^{Sh} \setminus \{x\})$.*

Removing x from the interface of C makes x a private variable of C, but does not affect any of its rules or actions.

Observe that for $x, y, z \in (C^{Exp} \cup C^{Sh}) \cap C^{A_0^W}$, hiding is associative, commutative, and idempotent. Therefore, we extend our notation for hiding from a single variable to a set of variables, i.e., for a set $S = \{x_1, x_2, ..., x_n\}$ of $0 < i \leq n$ variables $x_i \in (C^{Exp} \cup C^{Sh}) \cap C^{A_0^W}$, we define $\exists[S]C = \exists[x_1, x_2, ..., x_n]C = \exists[x_1]\exists[x_2]...\exists[x_n]C$, where the order is irrelevant.

8.4 Sequential composition of CRMs

In this section we introduce two CRM sequential composition operators: the cyclic (;) and the sum (+) composition. Each of these compositions combines two CRMs into a composite CRM whose behavior consists of sequences of *macro-steps*, in each of which the first and the second component CRM takes its turn to perform one round of its rule-execution as a *micro-step*. The component CRMs no longer interact with the environment at the end of their turns (i.e., the end of each of its micro-steps); instead, the composite CRM interacts with the environment only at upon the completion of each macro-step.

Definition 6 (CRM cyclic composition). *Let $C_1, C_2 \in$ **CRM** such that they share no private variables (i.e., $C_1^{Priv} \cap C_2^{Priv} = \emptyset$). The cyclic composition of C_1 and C_2, denoted as $C_1; C_2$ is $C = (C^I, A_0, R, T)$ where:*

$$
\begin{aligned}
&C^I = (C^{Exp}, C^{Imp}, C^{Sh})\\
&C^{Exp} = C_1^{Exp} \cup C_2^{Exp}\\
&C^{Imp} = (C_1^{Imp} \cup C_2^{Imp}) \setminus (C^{Exp} \cup C^{Sh}))\\
&C^{Sh} = (C_1^{Sh} \cup C_2^{Sh}) \setminus C^{Exp}
\end{aligned}
$$

$$A^0 = \{\texttt{turn} := 0, \mathit{envstep} := \bot; C_1^{A^0}; C_1^T; \mathit{envstep} := \bot; C_2^{A^0}; C_2^T; \mathit{envstep} := \bot\}$$

$$R = \{(\texttt{turn} = 1 \wedge g, a) \mid (g, a) \in C_1^R\} \cup \{(\texttt{turn} = 2 \wedge g, a) \mid (g, a) \in C_2^R\}$$

$$
\begin{aligned}
T = \{&(\{\mathit{if}\ (\texttt{turn} = 1)\ \mathit{then}\ a\ \mathit{fi} \mid a \in C_1^T\} \cup \{\mathit{if}\ (\texttt{turn} = 2)\ \mathit{then}\ a\ \mathit{fi} \mid a \in C_2^T\});\\
&\mathit{if}\ ((\texttt{turn}\ \texttt{mod}\ 2) = 0)\ \mathit{then}\ \mathit{envstep} := \top\ \mathit{else}\ \mathit{envstep} := \bot\ \mathit{fi};\\
&\texttt{turn} := (\texttt{turn}\ \texttt{mod}\ 2) + 1\}
\end{aligned}
$$

where $\texttt{turn} \in$ **Name** *represents a unique fresh new name obtained through new*(**Name**) *(see Section 2.1 for new*(**Name**)*).*

Intuitively, $C = C_1; C_2$ is a CRM that lets C_1 take a turn to perform C's first micro-step wherein C_1 performs its own choice of its enabled rules once, then lets C_2 perform its respective micro-step to complete a macro-step of C, at the end of which the environment gets to interact with C. The interface of $C_1; C_2$ is the same as the interface of $C_1 \times C_2$.

The initialization actions of C consist of a sequential composition of the initialization and termination actions of its component CRMs (see Equation 12), and initializing the fresh unique new variable $\texttt{turn} = 0$. Because A^0 sets *envstep* to $\bot$ before every $C_i^{A^0}$, the initialization actions of each component CRM (in $C_i^{A^0}$) can detect whether their respective CRM runs interacting directly with the environment (*envstep* $= \top$), or as a micro-step in some sequential composition (*envstep* $= \bot$), and initialize their respective CRMs accordingly (see initialization of *envstep* in Table 1).

The combination of the initialization action $\texttt{turn} := 0$ and the termination action $\texttt{turn} := (\texttt{turn}\ \texttt{mod}\ 2) + 1$ yield the following sequence of values for turn: $0, 1, 2, 1, 2, 1, 2, \ldots$. The conjunctions $\texttt{turn} = i \wedge g$ for $i \in \{1, 2\}$ with the guards g of rules $(g, a) \in C_i^R$ restrict C to run only the selected actions of the enabled rules of its component CRM C_i.

The termination action if $((\texttt{turn}\ \texttt{mod}\ 2) = 0)$ then *envstep* $:= \top$ else *envstep* $:= \bot$ fi permits C to interact with the environment only after its (micro-step) rounds where turn has the value of either 0 or 2, i.e., only upon completion of every macro-step.

Observe that the initialization actions in $C_1^{A^0}$ and $C_2^{A^0}$, and the termination actions in C_1^T and C_2^T, run concurrently in $C_1 \times C_2$ as in $C_1^{A^0} \cup C_2^{A^0}$ and $C_1^T \cup C_2^T$, respectively. As such, *neverconflicting*$(C_1^{A_0} \cup C_2^{A_0})$ and *neverconflicting*$(C_1^T \cup C_2^T)$ are prerequisites in Definition 4 for $C_1 \times C_2$ to be well-defined. In contrast, in cyclic composition (Definition 6) and sum composition (Definition 7), the initialization actions in $C_1^{A^0}$ run before those in $C_2^{A^0}$, and the termination actions in C_1^T run before those in C_2^T. As such,

neverconflicting($C_1^{A_0} \cup C_2^{A_0}$) and *neverconflicting*($C_1^T \cup C_2^T$) are not required for cyclic and sum compositions to be well-defined.

The cyclic and sum compositions differ only in when they perform the termination actions of their component CRMs. In cyclic composition (Definition 6), the composite CRM performs the termination actions of its component CRMs at the end of each of their respective micro-steps (including their micro-initialization-steps). In sum composition (Definition 7) the termination actions of component CRMs are performed not at the end of their respective micro-steps, but only collectively at the end of each macro-step.

Definition 7 (CRM sum composition). *Let* $C_1, C_2 \in$ **CRM** *such that they share no private variables (i.e.,* $C_1^{Priv} \cap C_2^{Priv} = \emptyset$*). The sum composition of* C_1 *and* C_2*, denoted as* $C_1 + C_2$ *is* $C = (C^I, A_0, R, T)$ *where:*

$$
\begin{array}{l}
C^I = (C^{Exp}, C^{Imp}, C^{Sh}) \\
C^{Exp} = C_1^{Exp} \cup C_2^{Exp} \\
C^{Imp} = (C_1^{Imp} \cup C_2^{Imp}) \setminus (C^{Exp} \cup C^{Sh})) \\
C^{Sh} = (C_1^{Sh} \cup C_2^{Sh}) \setminus C^{Exp}
\end{array}
$$

$$
A^0 = \{\texttt{turn} := 0, \mathit{envstep} := \bot; C_1^{A^0}; \mathit{envstep} := \bot; C_2^{A^0}; \mathit{envstep} := \bot\}
$$

$$
\begin{array}{l}
R = \{(\texttt{turn} = 1 \wedge g, a) \mid (g, a) \in C_1^R\} \cup \{(\texttt{turn} = 2 \wedge g, a) \mid (g, a) \in C_2^R\} \cup \\
\quad \{((\texttt{turn mod } 3) = 0, \{C_1^T; C_2^T\})\}
\end{array}
$$

$$
\begin{array}{l}
T = \{\mathit{if}\ ((\texttt{turn mod } 3) = 0)\ \mathit{then\ envstep} := \top\ \mathit{else\ envstep} := \bot\ \mathit{fi}; \\
\quad \texttt{turn} := (\texttt{turn mod } 3) + 1\}
\end{array}
$$

and $\texttt{turn} \in$ **Name** *represents a fresh new name obtained from new*(**Name**).

The combination of the initialization action $\texttt{turn} := 0$ and the termination action $\texttt{turn} := (\texttt{turn mod } 3) + 1$ yield the following sequence of values for turn: $0, 1, 2, 3, 1, 2, 3, 1, 2, 3, \ldots$. The conjunctions $\texttt{turn} = i \wedge g$ for $i \in \{1, 2\}$ with the guards g of rules $(g, a) \in C_i^R$ restrict C to run only the selected actions of the enabled rules of its component CRM C_i. The tailor-made rule $((\texttt{turn mod } 3) = 0, \{C_1^T; C_2^T\})$ then performs the composite action $C_1^T; C_2^T$ as an extra micro-step to end each macro-step.

The termination action if $((\texttt{turn mod } 3) = 0)$ then $\mathit{envstep} := \top$ else $\mathit{envstep} := \bot$ fi permits C to interact with the environment only after its (micro-step) rounds where turn has the value of either 0 or 3, i.e., only at the end of every macro-step (including initialization).

Example 7 (Interactive composition of WL and WC). The controller CRM, WC, formalizes the input controller described in Section 1.1.

```
 WC = (WC.I, WC.A0, WC.R, WC.T)
where
  WC.I = ({'wl 'time}, {'fin}, {})
  WC.A0 = {(< >,\lambda ().'fin := WLminSafe0)}
  WC.R = {wc.r1, wc.r2}
  WC.T = {}
```

where wc.r1 is the rule described in Example 4 and wc.r2 is the analogous rule for the case when the value of 'wl is greater or equal to WLmaxSafe.

In composition WL and WC interact: the controller reads the water level written by the water tank and the water level of the water tank is determined by the inflow set by the controller and the outflow set by the environment in the previous step. Thus the water tank rules should fire before the controller rules. To preserve the assumption by termination actions that all (original) imports have been written before the termination actions are executed, the termination actions need to be executed after all components have executed. We choose to expose 'wl as an export, but to make 'fin private. These requirements are satisfied by the sum form of sequential composition operation (Section 8.4) and the hiding operation (Section 8.3). Thus we define:

```
WLWC = Exists['fin](WL + WC)
```

Using the definitions of sum and hiding we obtain

```
  WLWC    = [WLWC.I, WLWC.A0, WLWC.R, WLWC.T]
where
  WLWC.I = ['fout 't, 'wl, {}]
  WLWC.A0 = { {turn := 0} ; WL.A0 ; WC.A02 ;
              {envstep := false} }
  WLWC.R = {(turn = 1) a.wl)
            (turn = 2, a.wc)
            (turn = 3, WL.T + WC.T)}
  WLWC.T = {if (turn=3) then envstep := true
                        else envstep := false fi;
            turn := (turn mod 3) + 1 }
```

Note that WL and WC satisfy the conditions for sum to be a CRM: combined initialization / termination actions of WL and WC read and write different variables, thus they are conflict free; and WL and WC have disjoint private variables (WC has none).

We illustrate runs with micro steps, with a sample WLWC execution.

Example 8 (Sample run of WLWC). Fix fout = .5. Recall WLmin = 1, WLminSafe = 2, WLmaxSafe = 3, WLmax = 4, and initially wl = 2.00, fin = 0. Below, we show the valuation maps Vij at each micro step *j* of step *i*. To avoid clutter, we show values only where variables are assigned.

```
     time  fout   wl   fin   t0   fin0  turn  envstep
V0    0    .5    2.00   0     0     0    1    false
v11               2.00                     2
v12                      1                 3
v13                               0     1    1    true
v21   1           2.50                     2    false
v22                      1                 3
v23                               1     1    1    true
v31   2           3.00                     2    false
v32                      0                 3
v33                               2     0    1    true
........
```

8.5 CRM division

Just as product is an essential operation for composition of simpler CRMs into more complex models, its inverse operation, division, is also useful for decomposition of CRMs into simpler modules, e.g., for analysis, modular replacement, or (dynamic) reconfiguration. Definition 8 establishes which CRMs can be divided by other CRMs.

Definition 8 (CRM divisibility). *Let $C_1, C_2 \in \mathbf{CRM}$. We say C_1 is divisible by C_2 if:*

1. C_1^I *subsumes* C_2^I*, i.e.:*
   (a) $C_2^{Exp} \subseteq C_1^{Exp}$
   (b) $(C_1^{Imp} \cap C_2^{Sh} = \emptyset) \wedge (C_2^{Imp} \subseteq C_1^{Imp} \cup C_1^{Exp} \cup C_1^{Sh})$
   (c) $C_2^{Sh} \subseteq C_1^{Exp} \cup C_1^{Sh}$
2. $C_2^{A_0} \subseteq C_1^{A_0}$
3. C_1^R *subsumes* C_2^R*, i.e.,* $r_2 \in C_2^R \implies \exists r_1 \in C_1^R \wedge r_1 \equiv r_2$
4. $C_2^T \subseteq C_1^T$

Lemma 5. *For all $C_1, C_2 \in \mathbf{CRM}$, $C_1 \times C_2$ is divisible by C_2.*

If C_1 is divisible by C_2, then there exists a C such that $C_1 = C_2 \times C$. Generally, however, C is not unique: there may be more than one unique C whose product composition with C_2 yields C_1. Therefore, Definition 9 defines the result of a division as *a set of quotients*, instead of a unique CRM.

Definition 9 (CRM division). *Let $C_1, C_2 \in \mathbf{CRM}$. The division of C_1 by C_2, denoted as C_1/C_2, is the set of quotients:*

$$C_1/C_2 = \begin{cases} Q & \text{if } C_1 \text{ is divisible by } C_2 \\ \{\emptyset_{\mathbf{CRM}}\} & \text{otherwise} \end{cases}$$

where the set of quotients, Q, is:

$$Q = \left\{ ((E, I, S), A_0, R, T) \;\middle|\; \begin{array}{l} E = (C_1^{Exp} \setminus C_2^{Exp}) \cup E^{xs} \\ I = (C_1^{Imp} \setminus C_2^{Imp}) \cup I^{xs} \\ S = (C_1^{Sh} \setminus C_2^{Sh}) \cup S^{xs} \\ A_0 = (C_1^{A_0} \setminus C_2^{A_0}) \cup A_0^{xs} \\ R = (C_1^R \setminus R_{equiv}) \cup R^{xs} \\ T = (C_1^T \setminus C_2^T) \cup T^{xs} \end{array} \right\}$$

and:

- $(R^S \cup A_0^N \cup T^N) \cap (E \cup I \cup S) = (R^S \cup A_0^N \cup T^N) \cap (C_1^{Exp} \cup C_1^{Imp} \cup C_1^{Sh})$
- $E^{xs} \subseteq C_1^{Exp} \cap C_2^{Exp}$
- $I^{xs} \subseteq C_1^{Imp} \cap (C_2^{Exp} \cup C_2^{Imp} \cup C_2^{Sh})$
- $S^{xs} \subseteq C_1^{Sh} \cap (C_2^{Exp} \cup C_2^{Sh})$
- $A_0^{xs} \subseteq C_1^{A_0}$
- $R_{equiv} = \{r_1 \mid r_1 \in C_1^R, r_2 \in C_2^R, r_1 \equiv r_2\}$
- $R^{xs} \subseteq R_{equiv}$
- $T^{xs} \subseteq C_2^T$

Intuitively, C_1/C_2 produces a set of quotient CRMs the product of each of which with C_2 is behaviorally equivalent to C_1.

Basic Properties of Division

- For all $C_1, C_2 \in \mathbf{CRM}$, every quotient $C \in C_1/C_2$ is a CRM.
- Every $C \in \mathbf{CRM}$ is divisible by $\emptyset_{\mathbf{CRM}}$ and $C/\emptyset_{\mathbf{CRM}} = \{C\}$.
- For all $C_1, C_2 \in \mathbf{CRM}$:
  1. if C_1 is divisible by C_2 then $C_1 \in C_1/C_2$.
  2. $C \in C_1/C_2 \implies C \subseteq C_1$.
- For all $C_1, C_2 \in \mathbf{CRM}$, $C_1 \in (C_1 \times C_2)/C_2$.
- For all $C_1, C_2 \in \mathbf{CRM}$, $C_1 \cap C_2 = \emptyset_{\mathbf{CRM}} \implies (C_1 \times C_2)/C_2 = \{C_1\}$.
- For all $C_1, C_2 \in \mathbf{CRM}$, $C \in C_1/C_2 \implies \mathcal{B}(C \times C_2) = \mathcal{B}(C_1)$.

Example 9 (Water system CRM division). To illustrate the utility of division, we start with an in/out controller CRM, WCIO. It has two rule sets: rules in `RI` read the water level and set `fin` (see Example 7); `RO` reads the water level and sets `fout`.

```
  WCIO= (WCIO.I, WCIO.A0, WCIO.R, WCIO.T)
where
  WCIO.I = ({'wl},{'fin, 'fout}, {}) --- imports, exports, shared
  WCIO.A0 = {(< >,\lambda (). 'fin :=  0; 'fout := .5)}
  WCIO.R =
   {  RI: {wc.r1, wc.r2}
      RO: { (true, {a.wo}) }
    }
  WCIO.T = {}
```

Recall that rules `wc.r1,wc.r2` are defined in Example 7. We also define a controller, WCO, for `fout`.

```
  WCO = (WCO.I, WCO.A0, WCO.R, WCO.T)
where
  WCO.I = ({'wl}, {'fout}, {}) -- imports,exports,shared
  WCO.A0 = {(< >,\lambda (). 'fout := .5)}
  WCO.R = { RO: (true,{a.wo}) }
  WCO.T = {}
```

Then we have the following: WCO divides WCIO and WC is in WCIO/WCO.

9 Time in CRM

Since the concept of time is inherent in physics, representation of time is essential in formal models of cyber-physical systems. We base our model of time on the concept of *dense time*, and for simplicity represent time values as non-negative real numbers.

We call a stream of real numbers $\delta \in \mathbb{N}_0 \to \mathbb{R}_0^+$ *non-Zeno progressive* if:

1. for every $i \in \mathbb{N}_0, \delta(i) < \delta(i+1)$, and
2. for every $n \in \mathbb{N}_0$, there exists an $i \in \mathbb{N}_0$ such that $\delta(i) > n$.

We stipulate that:

1. there exists a distinguished variable, ***time*** $\in$ **Name**, whose value ranges over $\mathbb{R}_0^+$ to represent the continuous passage of time.

2. ***time*** is available as a read-only variable for CRMs to import; and
3. change of ***time*** is always reported, i.e., $v \in \mathbf{V} \implies \textbf{\textit{time}} \in Dom(xchange(v))$.

Note that by stipulation 1, above, the values of ***time*** represent the continuous passage of time. Therefore, the successive values of ***time*** form a non-Zeno progressive infinite sequence of real numbers in $\mathbb{R}_0^+$.

The availability of ***time*** provided by the environment does not mean that a CRM is required to import it, neither does it preclude the possibility of a $C \in \mathbf{CRM}$ having its own internal clock manifested as a local, shared, or exported variable ***time***. Because time progresses continuously, the value of ***time*** changes at least as quickly as either C or the environment performs its actions.

We say that a $C \in \mathbf{CRM}$ shows an explicit sense of time in its run σ if $\textbf{\textit{time}} \in C^{Rd}$. In this case, for $i \in \mathbb{N}_0$, by Equation 15 $\textbf{\textit{time}} \in M^i$ and by Equation 16 a name-value pair $(\textbf{\textit{time}}, t)$ exists in every observable O^i in σ. Moreover, by stipulation 1, above, the t values of the $(\textbf{\textit{time}}, t)$ pairs in successive observables $O^i, i \in \mathbb{N}_0$ form a non-Zeno progressive sequence of real numbers in $\mathbb{R}_0^+$.

10 Related Work

We discuss related work in three areas: rewriting logic, labeled transition systems, and composable semantic models.

Rewriting Logic Rewriting Logic (RWL) [25,26] 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 a signature, E is a set of equations used to define functions and relations, and R is a set of 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. In RWL system states are represented by terms allowed by the signature, while CRM states are maps from variable names to values. RWL provides a means to formally specify the actions of rules, while CRM actions are mathematical functions. In both CRM and RWL the semantics of a system is based on the one step rewrite relation. In RWL this relation is determined by the rules. In CRM the relation accounts for interaction with the environment; CRM rules control only the exported and shared variables.

CRM and RWL both support expressive composition mechanisms. RWL supports a rich algebra of theories. This combined with the use of equationally defined functions to compose system states provides the ability to define many forms of interaction within a composition.

The rewrite relation has been generalized for large classes of rewrite theories to symbolic rewriting either as rewriting modulo SMT [32] or symbolic reachability (using unification rather than matching for rule application) [27,14]. Symbolic rewriting 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.

An important aspect of modeling CPS is representing change over time. Timed rewrite theories are a special class of rewrite theories where some rules have duration [28]. This can be used to represent many forms of interactions and change over time.

Labeled transition systems A number of existing formalisms have been used for modeling a system's interaction with its environment by labeling transitions, including finite automata models and process calculi The environment of a component is usually considered to be other components rather than including a physical, possibly non-computable, environment.

Automata may be non-deterministic, but they do not really address concurrency. They are composed by a product operation (that suffers blowup). The processes of process algebra are naturally concurrent and are composed using a parallel construction. In either case the mechanism of interaction is synchronizing of labels representing actions. Neither automata nor processes have explicit interfaces. The networks of communicating automata model [10] supports modeling concurrent executions. Again interactions are by synchronization of actions (also called messages).

Interface Automata [1], abstract from details of how the behavior of a component is generated to focus on observable behavior. The interface automata formalism specifies the temporal order of input and output actions of a component as well as its local events. Interface automata accept only some behaviors generated by the environment and specify a component's behavior only under these inputs. The composition of two interfaces is achieved by synchronizing their shared output and input actions. Two components are said to be composable if they satisfy some conditions on action names, and there is an environment in which the product has non-empty behavior. A notion of adapter for interface automata is proposed in [11] Adapters provide a mechanism to express simple interactions beyond basic synchronization. In contrast, a CRM specifies behavior under arbitrary environment conditions.

IO automata [24] model asynchronous concurrent systems. In contrast to interface automata, an IO automata is required to accept input actions at any time. The semantics includes only fair executions. Two or more IO automata component models can be composed under certain conditions. In the composite automata, actions with the same label must happen together. Component automata may operate at arbitrary relative speeds.

Proposed in 1997 as a formal model for synchronous groupware, Team Automata [15,34] extend IO automata by removing some of the restrictions on how actions compose. Formally, a team automaton consists of a synchronous product of an underlying set of its component automata. These component automata act as a conceptually identifiable team with some shared context and goals. Acting together, the component automata in a team automaton collaborate and cooperate through joint actions. A team automaton is a synchronous model wherein every one of its component automata either participates in one instantaneous action, or it remains idle. The precise manner in which component actions synchronize in team automata to form a joint action is determined by a *synchronization type* assigned to each action, which specifies the range of the number of participating actions.

Constraint Automata (CA) [9] were introduced as an operational semantics for the coordination language Reo. Reo [3,4,5,6] is an exogenous coordination language in which interaction protocols are built compositionally out of a set of (user defined) primitive protocols. CA were primarily intended to express the semantics of Reo composite protocols (called circuits) for the purpose of formal verification through model checking [8,18]. Composition of constraint automata allows both synchronous and asynchronous

composition of actions. It uses a strict notion of synchronization that while simpler than synchronization in team automata (by eliminating the need for synchronization types), supports the same expressive power through additional asynchronous components. Moreover, in contrast to team automata (as well as many other automata models), constraint automata support consideration and manipulation of data in specification of the behavior of a system.

Various automata models have been extended to represent time. The monograph *The Theory of Timed I/O Automata* [17] describes the extension of I/O automata to model timed executions. This model allows components to operate on different time scales. Timed constraint automata were introduced for verification of timed interaction protocols specified as timed Reo connector circuits [2,19].

Labeled transition systems are abstract models with limited means to express the structure of state. Formal verification of a component is done by fixing its environment, which often means other components. CRM models are designed to be executed either concretely or symbolically. A concrete execution represents a behavior of a CRM running in a specific environment. A symbolic execution represents a behavior of a CRM running in any environment whose variables satisfy a set of constraints, that may also involve CRM component parameters.

Time is an essential part of hybrid systems models as physical components model change over time for given input parameters. Formalisms based on hybrid systems such as Differential Logic and hybrid programs model physical behavior of components and the interaction with controllers. [29] A composition algebra for hybrid programs is presented in [22,23]. Here interaction between components is based on shared variables and analysis is done by fixing a specific environment.

Composable Semantics The CRM interaction model is inspired by the work on composable semantics for Cyber-Physical System components [20,21], where component behaviors are expressed as traces consisting of sequences of observations, hiding internal details of cyber and physical processes. A composition operation parameterized by interaction relations is defined. Using interaction relations, diverse interaction mechanisms including forms of synchronization and mutual exclusion can be expressed. Conditions for algebraic properties of composition such as associativity and commutativity are identified. CRM semantics can be mapped to this semantic model and many important interaction relations can be realized by CRM coordination components.

11 Conclusions and future work

In this paper we presented the Concurrent Rules Machine (CRM), a model of concurrent (and distributed) computation with explicit representation of interaction of a system with its environment. We defined CRM structure and execution semantics. An algebra of operations on CRMs is defined including different notions of composition (product), hiding, and *division* and key properties of these operations are given. As one intended application of CRMs is modular specification of cyber-physical systems, representation of time is also discussed. The concepts and operations are illustrated by a water tank system example with components representing the water level sensor and the control of input and output flows.

This first step provides the mathematical foundation. Future work includes implementation of a class of CRMs in the Maude rewriting logic system. This will allow modeling and reasoning about complex systems in the context of different environment constraints. The algebra of CRMs can be implemented using Maude's module algebra and reflection. In addition we plan to develop a language for specifying interactive composition in terms of component interfaces. This will simplify developing complete systems and support compositional reasoning. Of course of great interest is developing real world case studies.

References

1. L. Alfaro and T. A. Henzinger. Interface automata. page 109–120, 2001.
2. F. Arbab, C. Baier, F.C. Boer, and J.J.M.M. Rutten. Models and temporal logics for timed component connectors. In *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 198–207. IEEE Society Press, 2004.
3. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Math. Structures Comput. Sci.*, 14(3):329–366, 2004.
4. Farhad Arbab. Puff, the magic protocol. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 169–206. Springer, 2011.
5. Farhad Arbab. *Proper Protocol*, pages 65–87. Springer International Publishing, Cham, 2016.
6. Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In Farhad Arbab and Carolyn Talcott, editors, *Coordination Models and Languages*, pages 22–39, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
7. Guillaume Babin, Yamine Aït Ameur, Neeraj Kumar Singh, and Marc Pantel. A system substitution mechanism for hybrid systems in Event-B. pages 106–121, 2016.
8. Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. Design and verification of systems with exogenous coordination using vereofy. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 97–111, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
9. Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Sci. Comput. Programming*, 61(2):75–113, 2006.
10. Howard Bowman and Rodolfo Gomez. *Concurrency Theory: Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. Springer-Verlag London Limited, 2006.
11. Samir Chouali, Sebti Mouelhi, and Hassan Mountassir. Adapting component behaviours using interface automata. In *36th Euromicro Conference on Software Engineering and Advanced Applications*, 2010.
12. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
13. Guillaume Dupont, Yamine Aït Ameur, Neeraj Kumar Singh, and Marc Pantel. Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, (4):35:1–35:37, 2021.
14. 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.
15. Clarence A. Ellis. Team automata for groupware systems. In *Proceedings of GROUP'97, International Conference on Supporting Group Work: The Integration Challenge, November 16-19, 1997, Embassy Suites Hotel, Phoenix, Arizona, USA*, pages 415–424. ACM, 1997.
16. John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.
17. Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Distributed Computing. Springer, 2 edition, 2011.
18. Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *Sci. Comput. Program.*, 74(9):688–701, 2009.
19. Natallia Kokash, Mohammad Mahdi Jaghoori, and Farhad Arbab. From timed Reo networks to networks of timed automata. *Electronic Notes in Theoretical Computer Science*, 295:11–29, 2013. Proceedings the 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).
20. Benjamin Lion, Farhad Arbab, and Carolyn Talcott. A semantic model for interacting cyber-physical systems. In *Workshop on Interaction and Concurrency Experience*, EPTCS, 2021. to appear.
21. Benjamin Lion, Farhad Arbab, and Carolyn L. Talcott. A semantic model for interacting cyber-physical systems. *CoRR*, abs/2106.15661, 2021.
22. 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.
23. 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.
24. Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.
25. J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
26. José Meseguer. Twenty years of rewriting logic. *J. Log. Algebraic Methods Program.*, 81(7-8):721–781, 2012.
27. José Meseguer. Generalized rewrite theories, coherence completion, and symbolic methods. *J. Log. Algebraic Methods Program.*, 110, 2020.
28. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, 2007.
29. André Platzer. Differential dynamic logic for hybrid systems. *J Autom Reasoning*, pages 143–189, 2008.
30. 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.
31. Camilo Rocha, José Meseguer, and César Muñoz. Rewriting modulo smt and open system analysis. In Santiago Escobar, editor, *Rewriting Logic and Its Applications*, pages 247–262, Cham, 2014. Springer International Publishing.
32. 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.
33. The KeYmaera X team. KeYmaera X: An aXiomatic tactical theorem prover for hybrid systems, 2022. Last accessed Sept 22, 2022.

34. Maurice H. ter Beek, Rolf Hennicker, and José Proença. Team automata: Overview and roadmap. In Ilaria Castellani and Francesco Tiezzi, editors, *Coordination Models and Languages*, pages 161–198, Cham, 2024. Springer Nature Switzerland.