A Rewriting Framework for Interacting Cyber-Physical Agents

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

[1] Leiden University, Leiden, The Netherlands lion@cwi.nl
[2] CWI, Amsterdam, The Netherlands arbarb@cwi.nl
[3] SRI International, CA, USA talcott@gmail.com

Abstract. The analysis of cyber-physical systems (CPS) is challenging due to the large state space and the continuous changes occurring in their constituent parts. Design practices favor modularity to help reducing this complexity. In a previous work, we proposed a discrete semantic model for CPS that captures both cyber and physical aspects as streams of discrete observations, which ultimately form the behavior of a component. This semantic model is denotational and compositional, where each composition operator algebraically models an interaction between a pair of components.

In this paper, we propose a specification of components as rewrite systems. The specification is operational and executable, and we study conditions for its semantics as components to be compositional. We demonstrate our framework by modeling a coordination of robots moving on a shared field. We show that our system of robots can be coordinated by a protocol in order to exhibit a desired emerging behavior. We use an implementation of our framework in Maude to give practical results.

1 Introduction

Cyber-physical systems are inherently concurrent. From a cyber point of view, the timing of a decision to sense or act on its physical environment impacts the resulting outcome. Moreover, several cyber entities may share the same physical environment, leading to race conditions. From a physical point of view, the ordering of events is not always possible, as some events may be independent. Moreover, two observers of the same physical phenomenon may order events differently. A concurrency protocol encapsulates the orderings of events acceptable to an application, and expressing protocols as separate, concrete modules (as in exogenous coordination [1]) helps to reduce the complexity in the design of cyber-physical systems.

More specifically, in this context, each part of a cyber-physical system (e.g., a car, a road, a battery, etc.) is represented as a module, and the system captures the concurrent and interactive execution of each module. We list the following benefits of such approach. First, it makes concurrency explicit at the level of modules, amenable to exogenous coordination, which provides the opportunity to reason about concurrency protocols directly as first-class objects (e.g., how much a move of a robot consumes energy, can two robots move ‘simultaneously’, etc.). Then, the representation of a system remains small. Often, a modular design allows composing constituent components statically to analyze the resulting system, or dynamically at runtime to keep the state space small for, e.g., simulating some runs. Finally, a component comes with a notion of an interface, that specifies what is visible and what is hidden from other components. This way, both discrete and continuous aspects of components have the same type of interface, containing the set of observations over time.

In [12] we present a model of components that captures timed-event sequences (TESs) as instances of their behavior. An observation is a set of events with a unique time stamp. A component has an interface that defines which events are observable, and a behavior that denotes all possible sequences of its observations (i.e., a set of TESs). Our component model is equipped with a family of operators parametrized with an interaction signature. Thus, cyber-physical systems are defined modularly, where each product of two components models the interaction occurring between the two components. The strength, as well as practical limitation, of our semantic model is its abstraction: there is no fixed machine specification that generates the behavior of a component. We give in this paper an operational description of components as rewrite systems.

Rewriting logic is a powerful framework to model concurrent systems [14, 15]. Moreover, implementations, such as Maude [3], make system specifications both executable and analyzable. Rewriting logic is suitable for specifying cyber-physical systems, as the underlying equational theory can represent both discrete and continuous changes. We give an operational specification for components as rewriting systems, and show its compositionality under some assumptions.

Finally, we apply our work to an example that considers two energy sensitive robots moving on a shared field. Each of the two robots aims at reaching the other robot’s initial position which, by symmetry, may eventually lead to a crossing situation. The crossing of the two robots is the source of a livelock behavior which can lead to failure (i.e., no energy left in the battery). We show how, an exogenous coordination imposed by a protocol can coordinate the moves of the two robots to avoid the livelock situation. We demonstrate the result using our implementation of our framework in Maude.

We present the following contributions:

- an operational specification of components as rewrite systems;
- some conditions for the rewrite system’s semantics to be compositional;
- an incremental, runtime implementation of composition;
- illustration of how a composed Maude specification can be used to incrementally analyze a system design using a case study involving the behavior of two coordinated robot agents roaming on a field.

The remainder of the paper is organized as follows. In Section 2, we recall some results on the algebra of components defined in [12], and give as examples the component version of a robot, a battery, and their product. In Section 3, we give an operational specification, using rewriting logic, of a product of components as a system of agents. We show compositionality: the component of a system of agents is equal to the product of each agent component. In Section 4, we detail the implementation in Maude of the operational specification given in Section 3 and analyse a system consisting of two robots, two private batteries, and a shared field.

2 Semantic model: algebra of components

The design of complex systems becomes simpler if such systems can be decomposed into smaller sub-systems that interact with each other. In order to simplify the design of cyber-physical systems, we introduced in [12] a semantic model that abstracts from the internal details of both cyber and physical processes. As first class entities in this model, a component encapsulates a behavior (set of TESs) and an interface (set of events). We recall basic definitions and properties in this section. See [10] for additional examples.

2.1 Components

Preliminaries A timed-event stream, TES, σ over a set of events E is an infinite sequence of *observations*, where its i^{th} observation $\sigma(i) = (O, t)$, $i \in \mathbb{N}$, consists of a pair of a subset of events in $O \subseteq E$, called the *observable*, and a positive real number $t \in \mathbb{R}_+$ as time stamp. A timed-event stream (TES) has the additional properties that its consecutive time stamps are monotonically increasing and non-Zeno, i.e., if $\sigma(i) = (O_i, t_i)$ is the i^{th} element of TES σ, then (1) $t_i < t_{i+1}$, and (2) for any time $t \in \mathbb{R}_+$, there exists an element $\sigma(i) = (O_i, t_i)$ in σ such that $t < t_i$. We use $\sigma^{(k)}$ to denote the k-th derivative of the stream σ, such that $\sigma^{(k)}(i) = \sigma(i + k)$ for all $i \in \mathbb{N}$. We refer to the stream of observables of σ as its first projection $\text{pr}_1(\sigma) \in \mathcal{P}(E)^\omega$, and the stream of time stamps as its second projection $\text{pr}_2(\sigma) \in \mathbb{R}_+^\omega$. We write $(O, t) \in \sigma$ if there exists $i \in \mathbb{N}$ such that $\sigma(i) = (O, t)$.

We write $\sigma(t) = O$ if there exists $i \in \mathbb{N}$ such that $\sigma(i) = (O, t)$, and $\sigma(t) = \emptyset$ otherwise. We use $dom(\sigma)$ to refer to the set of observable time stamps, i.e., the set $dom(\sigma) = \{t \in \mathbb{R}_+ \mid \exists i.\text{pr}_2(\sigma)(i) = t\}$. Moreover, we use $\sigma \cup \tau$ to denote the stream such that, for all $t \in \mathbb{R}_+$, $(\sigma \cup \tau)(t) = \sigma(t) \cup \tau(t)$ and $dom(\sigma \cup \tau) = dom(\sigma) \cup dom(\tau)$

A component denotes *what* observables are possible, over time, given a fixed set of events. We give three examples of components, which capture some cyber-physical aspects of concurrent systems.

Definition 1 (Component). *A component $C = (E, L)$ is a pair of a set of events E, called its* interface*, and a* behavior *$L \subseteq TES(E)$.*

Given component $A = (E_A, L_A)$, we write $\sigma : A$ for a TES $\sigma \in L_A$.

Example 2 (Battery). A battery component is a pair $(E_B(C), L_B(C))$ with events $read(l) \in E_B$ for $0\% \leq l \leq 100\%$, $charge(\mu) \in E_B$, and $discharge(\mu) \in E_B$ with μ a (dis)charging coefficient in % per seconds, and C a constant capacity in *mAH*. The battery displays its capacity with the event *capacity(C)*. The behavior L_B is a set of sequences $\sigma \in L_B$ such that there exists a piecewise linear function $f : \mathbb{R}_+ \to \mathcal{P}(E_B)$ with, for $\sigma(i) = (O_i, t_i)$,

– for $\sigma(0) = (O_0, t_0)$, $f([0; t_0]) = 100\%$, i.e., the battery is initially fully charged;
– if $O_i = \{read(l)\}$, then $f(t_i) = l$ and the derivation $f'_{[t_{i-1}, t_{i+1}]}$ of f is constant in $[t_{i-1}, t_{i+1}]$, i.e., the observation does not change the slope of f at time t_i;
– if $O_i = \{discharge(\mu)\}$, then $f_{[t_i, t_{i+1}]}(t) = \max(f(t_i) - (t - t_i)\mu, 0)$;
– if $O_i = \{charge(\mu)\}$, then $f_{[t_i, t_{i+1}]}(t) = \min(f(t_i) + (t - t_i)\mu, 100)$;

where $f_{[t_1; t_2]}$ is the restriction of function f on the interval $[t_1; t_2]$. There is *a priori* no restrictions on the time interval between two observations, as long as the sequence of timestamps is increasing and non-Zeno. □

Example 3 (Robot). A robot with identifier i is a component $R(i, T) = (E_R, L_R(T))$ with events $read(i, l) \in E_R$ for $0\% \leq l \leq 100\%$, $d(i, p) \in E_R$ with p the power requested by the robot for the move and d the direction, and T a period in seconds. For instance, the event $N(i, p)$ represents robot i moving North with power p. The robot reads the capacity of its battery with the event *getCapacity(i,C)* $\in E_R$, with C in *mAH*. Once the robot knows the capacity of the battery, the values read in percent can be converted to remaining power.

The behavior $L_R(T)$ contains any sequence of observations at fix period T, such that $\sigma \in L_R(T)$ if and only if $\sigma(i) = (O_i, t_i)$ implies $t_i = kT$ with $k \in \mathbb{N}$ and $O_i \subseteq E_R$ with $|O_i| = 1$. We assume that the robot does one action at a time: either a read of its sensors, or a move in some direction. □

2.2 Product and division

Components describe which observations occur over time. When run concurrently, observable events from a component may relate to observable events of another component. This relation defines what kind of interaction occurs between the two components, as it may enforce two events to occur within the same observable at the same time (e.g., actuation of a wheel and changes of location of the robot), or it may prevent two events to occur simultaneously (e.g., two robots moving to the same physical location). Interaction constraints are therefore captured by an algebraic operator that acts on components. The result of forming the product of two components is a new component, whose behavior contains the composition of every pair of TESs, one from each product operand, that satisfies the underlying constraints imposed by that specific operator.

Let $A = (E_A, L_A)$ and $B = (E_B, L_B)$ be two components. We use the relation $R(E_A, E_B) \subseteq TES(E_A) \times TES(E_B)$ and the function $\oplus : TES(E) \times TES(E) \to TES(E)$, with $E = E_A \cup E_B$, to range over composability relations and composition functions, respectively. We use Σ to range over interaction signatures, i.e., pairs of a composability relation and a composition function.

Definition 4 (Product). *The* product *of components A and B under interaction signature $\Sigma = (R, \oplus)$ is the component $C = A \times_{\Sigma} B = (E_A \cup E_B, L)$ where*

$$L = \{\sigma \oplus \tau \mid \sigma \in L_A, \tau \in L_B,\ (\sigma, \tau) \in R(E_A, E_B)\}$$

For simplicity, we write $\times$ as a general product when the specific Σ is irrelevant.

Example 5. We define $\Sigma_{RB} = ([\kappa_{RB}], \cup)$ where $\cup$ unions two TESs as defined in the preliminaries, and $[\kappa_{RB}]$ specifies co-inductively (see [12] for details of the construction), from a relation on observations κ_{RB}, how event occurrences relate in the robot and the battery components of capacity C. More specifically, κ_{RB} is the smallest symmetric relation over observations such that $((O_1, t_1), (O_2, t_2)) \in \kappa_{RB}$ implies that $t_1 = t_2$ and

- the discharge event in the battery coincides with a move of the robot, i.e., $d(i, p) \in O_1$ if and only if $discharge(\mu) \in O_2$. Moreover, the interaction signature imposes a relation between the discharge coefficient μ and the required power p, i.e., $\mu = p/C$;
- the read value of the robot sensor coincides with a value from the battery component, i.e., $read(i, l) \in O_1$ if and only if $read(l) \in O_2$;
- the robot reads the capacity value that corresponds to the battery capacity, i.e., *getCapacity(i,c)* $\in O_1$ if and only if *capacity(c)* $\in O_2$.

The product $B \times_{\Sigma_{RB}} R(T, i)$ of a robot and a battery component, under the interaction signature Σ_{RB}, restricts the behavior of the battery to match the periodic behavior of the robot, and restricts the behavior of the robot to match the sensor values delivered by the battery.
As a result, the behavior of the product component $B \times_{\Sigma_{RB}} R(T, i)$ contains all observations that the robot performs in interaction with its battery. Note that trace properties, such as *all energy sensor values observed by the robot are within a safety interval*, does not necessarily entail safety of the system: some unobserved energy values may fall outside of the safety interval. Moreover, the frequency by which the robot samples may *reveal* some new observations, and such robot can safely sample at period T if, for any period $T' \leq T$, the product $B \times_{\Sigma_{RB}} R(T', i)$ satisfies the safety property. $\Box$

3 System of agents and compositional semantics

Components in Section 2 are declarative. Their behavior consists of all the TESs that satisfy some internal constraints. The abstraction of internal states in components makes the specification of observables and their interaction easier. The downside of such declarative specification lies in the difficulty of generating an element from the behavior, and ultimately verifying properties on a product expression.

An operational specification of a component provides a mechanism to construct elements in its behavior. An *agent* is the operational specification that produces finite sequences of observations that, in the limit, determine the behavior of a component. An agent is stateful, and has transitions between states, each labeled by an observation, i.e., a set of events with a time-stamp. We consider a finite specification of an agent as a rewrite theory, where finite applications of the agent's rewrite rules generate a sequence of observables that form a prefix of some elements in the behavior of its corresponding component. We restrict the current work to integer time labeled observations. While in the cyber-physical world, time is a real quantity, we consider in our fragment a countable infinite domain for time, i.e., natural numbers. The time interval between two tics is therefore the same for all agents, and may be interpreted as, e.g., seconds, milliseconds, femtoseconds, etc. We show how an agent may synchronize with a local clock that forbids actions at some time values, thus modeling different execution speeds.

An operational specification of a composite component provides a mechanism to construct elements in the behavior of a product expression. The product on components is parametrized by an interaction signature that tells *which* TESs can compose, and *how* they compose to a new TES. We consider, in the operational fragment of this section, interaction signatures each of whose composability relation is co-inductively defined from a relation on observations κ. Intuitively, such restriction enables a step-by-step operation to check that the head of each sequence is valid, i.e., extends the sequence to be a prefix of some elements in the composite component. Moreover, we require κ to be such that the product on component $\times_{([\kappa],\cup)}$ is commutative and associative (see [12]). By *system* we mean a set of agents that compose under some interaction signature $\Sigma = ([\kappa], \cup)$. A system is stateful, where each state is formed from the states of its component agents, and has transitions between states, each labeled by an observation, formed from the component agent observations. We consider a finite specification of a system as the composition of a set of rewriting theories (one for each agent), and a system rewrite rule that produces a composite observation complying with the relation κ. We prove compositionality: the system component is equal to the product under the interaction signature $\Sigma = ([\kappa], \cup)$ of every one of its constituent agent components.

In order to give to the agent a semantics as components, we recall some results and notations about TES transition systems $T = (Q, E, \rightarrow)$ (see [11] for more results on TES transition systems) where Q is a set of states, E a set of events, and $\rightarrow \subseteq Q \times (\mathcal{P}(E) \times \mathbb{R}_+) \times Q$ a set of transitions.

We write $q \xrightarrow{u} p$ for the sequence of transitions $q \xrightarrow{u(0)} q_1 \xrightarrow{u(1)} q_2 ... \xrightarrow{u(n-1)} p$, where $u = \langle u(0), ..., u(n-1)\rangle \in (\mathcal{P}(E) \times \mathbb{R}_+)^n$. We write $|u|$ for the size of the sequence u.

We use $\mathcal{L}^{\text{fin}}(T, q)$ to denote the set of finite sequences of observables labeling a finite path in T starting from state q, such that

$$\mathcal{L}^{\text{fin}}(T, q) = \{u \mid \exists q'.q \xrightarrow{u} q', \forall i < |u| - 1.u(i) = (O_i, t_i) \wedge t_i < t_{i+1}\}$$

Additionally, the set $\mathcal{L}^{\text{fin}*}(T, q)$ is the set of sequences from $\mathcal{L}^{\text{fin}}(T, q)$ postfixed with empty observations, i.e., the set

$$\mathcal{L}^{\text{fin}*}(T, q) = \{u\tau \in \mathit{TES}(E) \mid u \in \mathcal{L}^{\text{fin}}(T, q) \text{ and } \tau \in \mathit{TES}(\emptyset)\}$$

We use $\mathcal{L}^{\text{inf}}(T, q)$ to denote the set of TESs labeling infinite paths in T starting from state q, such that

$$\mathcal{L}^{\text{inf}}(T, q) = \{\sigma \in \mathit{TES}(E) \mid \forall n.\sigma[n] \in \mathcal{L}^{\text{fin}}(T, q)\}$$

where, as introduced in Section 2, $\sigma[n]$ is the prefix of size n of σ.

Let $X \subseteq \mathit{TES}(E)$, we use $cl(X)$ to denote the set that contains the continuation with empty observations of any prefix of an element in X, i.e., $cl(X) = \{u\tau \in \mathit{TES}(E) \mid \tau \in \mathit{TES}(\emptyset) \text{ and } \exists\sigma.\exists i.\sigma \in X \wedge \sigma[i] = u\}$. Given a component $C = (E, L)$, we write $cl(C)$ for the new component $(E, cl(L))$.

3.1 Action, agent, and system

We give the operational counterparts of an observation, a component, and a product of components as, respectively, an action, an agent, and a system of agents. See [10] for proof sketches.

Action Actions are terms of sort `Action`. An action has a name of sort `AName` and some parameters. We distinguish two typical actions, the idle action $\star$ and the ending action `end`. A term of sort `Action` corresponds to an observable, i.e., a set of events. The idle action $\star$ and the ending action `end` both map to the empty set of events. An example of an action is `move(R1,d)` or `read(R1, position, l)` that, respectively, moves agent `R1` in direction `d` or reads the value `l` from the position sensor of `R1`. The semantics of action `move(R1, d)` consists of all singleton event of the form {`move(R1, d)`} with d a constant direction value. We use the operation $\cdot$: `Action Action` $\rightarrow$ `Action` to construct a composite action `a1` $\cdot$ `a2` out of two actions `a1` and `a2`.

Agent An agent operationally specifies a component in rewriting logic. We give the specification of an agent as a rewrite theory, and provide the semantics of an agent as a component. An agent is a four tuple $(\Lambda, \Omega, \mathcal{E}, \Rightarrow)$, each of whose elements we introduce as follow.

The set of sorts Λ contains the `State` sort and the `Action` sort, respectively for state and action terms. A pair of a state and a set of actions is called a configuration. The set of function symbols Ω contains ϕ : `State` $\times$ `Action` $\rightarrow$ `State`, that takes a pair of a state and an action term to produce a new state. The (Λ, Ω)-equational theory $\mathcal{E}$ specifies the update function ϕ. The set of equations that specify the function ϕ can make ϕ both a continuous or discrete function.

The rule pattern in (1) updates a configuration with an empty set to a new configuration, i.e.,

$$(s, \emptyset) \Rightarrow (s', \mathit{acts}) \tag{1}$$

with *acts* a non-empty set of action terms, and s' a new state. We call an agent *productive* if, for any state s : `State`, there exists a state s' with $(s, \emptyset) \Rightarrow (s', acts)$ and *acts* non empty set. Such agent may eventually do the idling action $\star$.

We give a semantics of an agent as a component by considering the limit application of the agent rewrite rules. We construct a TES transition system $\mathcal{T}_{\mathcal{A}} = (Q, E, \rightarrow)$ as an intermediate representation for agent $\mathcal{A} = (\Lambda, \Omega, \mathcal{E}, \Rightarrow)$. The set of states $Q = \texttt{State} \times \mathbb{N}$ is the set of pairs of a state of $\mathcal{A}$ and a time-stamp natural number. We use the notation $[s, t]$ for states in Q where $t \in \mathbb{N}$. The set of events E is the union of all observables labeling the transition relation $\rightarrow \subseteq Q \times (\mathcal{P}(E) \times \mathbb{N}) \times Q$, defined as the smallest set such that, for $t \in \mathbb{N}$:

$$\frac{(s, \emptyset) \Rightarrow (s', acts) \qquad a \in acts \qquad \phi(s', a) =_{\mathcal{E}} s''}{[s, t] \xrightarrow{(a, t+1)} [s'', t+1]} \tag{2}$$

An agent that performs a rewrite moves the global time from one unit forward. All agents share the same time semantically, and we show some mechanisms at the system level to artificially run some agents *faster* than others.

Let $\mathcal{A} = (\Lambda, \Omega, \mathcal{E}, \Rightarrow)$ be an agent initially in state $s_0 \in S$ at time $t_0 \in \mathbb{N}$. The finite, respectively infinite, component semantics of $\mathcal{A}$ is the component $[\![\mathcal{A}([s_0, t_0])]\!]^* = (E, \mathcal{L}^{\text{fin}*}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$, respectively the component $[\![\mathcal{A}([s_0, t_0])]\!] = (E, \mathcal{L}^{\text{inf}}(\mathcal{T}_{\mathcal{A}}, [s_0, t_0]))$, with $E = \bigcup_{a \in \texttt{Action}} a$.

Lemma 6 (Closure). *Let $\mathcal{A}$ be a productive agent initially in state $[s_0, t_0]$. Then $[\![A([s_0, t_0])]\!]^* = cl([\![A([s_0, t_0])]\!])$.*

Lemma 6 gives a condition under which a step by step execution of the agent is sound with respect to generating prefixes of elements in the component semantics. More precisely, if an agent $\mathcal{A}$ is productive, Lemma 6 ensures that finite sequences of rewrite rule applications generate finite sequences of observations each of which is a prefix of an element in the behavior of the component corresponding to $\mathcal{A}$. Alternatively, if $\mathcal{A}$ is not productive, a finite sequence of rule application may lead to a state for which no rule applies anymore. In such a case, there may not be any corresponding element in the agent component for which such finite sequence is a prefix.

System A system gives an operational specification of a product of a set of components under $\Sigma = ([\kappa], \cup)$. The composability relation κ is fixed to be symmetric, so that the product $\times_{\Sigma}$ is commutative. We define $[\kappa]$ co-inductively, as in [11,12]. Formally, a system consists of a set of agents with additional sorts, operations, and rewrite rules. A system is a tuple $(\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ where $\mathcal{A}$ is a set of agents. We use $(\Lambda_i, \Omega_i, \mathcal{E}_i, \Rightarrow_i)$ to refer to agent $\mathcal{A}_i \in \mathcal{A}$.

The set of sorts Λ contains a sort `Action` $\in \Lambda$ which is a super sort of each sort $\texttt{Action}_i$ for $\mathcal{A}_i \in \mathcal{A}$. The set Ω contains the function symbol `comp` : `Action` $\times$ `Action` $\rightarrow$ `Bool`, which relates pairs of action terms. Given two actions `a1,a2:Action`, `comp(a1, a2)` = `True` when the two actions `a1` and `a2` are *composable*. The set of equations $\mathcal{E}$ specifies the composability relation `comp`.

First, we impose `comp` to be symmetric, i.e., for all actions `a1,a2:Action`, `comp(a1,a2)` = `comp(a2,a1)`. Second, we assume that $\texttt{comp}(\texttt{a1}\cdot\texttt{a2},\ \texttt{a3})$ and `comp(a1, a2)` hold if and only if `comp(a2, a3)` and $\texttt{comp}(\texttt{a1},\ \texttt{a2}\cdot\texttt{a3})$ hold, for any actions `a1, a2, a3` from disjoint agents. Given a set `actions` of actions, we use the notation `comp(actions)` for the predicate that is `True` if all pairs of actions in `actions` are composable, i.e., for all `a1, a2` in `actions`, `comp(a1,a2)` is `True` and for all agent $\mathcal{A}_i$ such that there is no $\texttt{a3} : \texttt{Action}_\texttt{i} \in \texttt{actions}$, then $\texttt{comp}(\texttt{a1}, \star_\texttt{i})$ is `True`. We call a set `actions` of actions for which `comp(actions)` holds, a *clique*. The conditions for a set of actions to form a clique models the fact that each action in the clique is independent from agent $\mathcal{A}_i$ with no action in that clique (see Section 4.1 for an instance of `comp`), and therefore composable with the silent action $\star_i$. The relation `comp` can be graphically modelled as an undirected graph relating actions, where a clique is a connected component.

The rewrite rule pattern in (3) selects a set of actions, at most one from each agent, checks that the set of actions forms a clique with respect to `comp`, and applies the update accordingly. For $\{k_1, ..., k_j\} \subseteq \{1, ..., n\}$:

$$\{(s_{k_1}, acts_{k_1}), ..., (s_{k_j}, acts_{k_j})\} \Rightarrow_S \{(\phi_{k_1}(s_{k_1}, a_{k_1}), \emptyset), ..., (\phi_{k_j}(s_{k_j}, a_{k_j}), \emptyset)\} \quad (3)$$

if $\texttt{comp}(\bigcup_{i\in[1,j]}\{a_{k_i}\})$. As we show later, a system does not necessarily update all agents in lock steps, and an agent not doing an action may stay in the configuration $(s, \emptyset)$. As multiple cliques may be possible, there is non-determinism at the system level. Different strategies may therefore choose different cliques as, for instance, taking the largest clique.

We define the transition system for $\mathcal{S} = (\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ as the TES transition system $\mathcal{T}_\mathcal{S} = (Q, E, \rightarrow)$ with $Q = \texttt{StateSet} \times \mathbb{N}$ the set of states, E the union of all observables labeling the transition relation $\rightarrow \subseteq Q \times (\mathcal{P}(E) \times \mathbb{N}) \times Q$, which is the smallest transition relation such that, for $\{k_1, ..., k_j\} \subseteq \{1, ..., n\}$:

$$\frac{\{(s_{k_i}, acts_{k_i})\}_{i\in[1,j]} \Rightarrow_S \{(\phi_{k_i}(s_{k_i}, a_{k_i}), \emptyset)\}_{i\in[1,j]} \quad \bigwedge_{i\in[1,j]} \phi_{k_i}(s_{k_i}, a_{k_i}) =_{\mathcal{E}_i} s''_{k_i}}{[\{s_i\}_{i\in[1,n]}, t] \xrightarrow{(\bigcup_{i\in[1,j]} a_{k_i}, t+1)} [\{s_1, ..., s''_{k_1}, ..., s''_{k_j}, ..., s_n\}, t+1]} \quad (4)$$

for $t \in \mathbb{N}$ and where we use the notation $\{x_i\}_{i\in[1,n]}$ for the set $\{x_1, ..., x_n\}$.

Remark 7. The top left part of the rule is a rewrite transition at the system level. As defined earlier, the condition for such rewrite to apply is the formation of a clique by all of the actions in the update. The states and labels of the TES transition system (bottom of the rule) are sets of states and sets of labels from the TES transition system of every agent in the system.

Let $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_n\}$ be a set of agents, and let $\mathcal{S} = (\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ be a system initially in state $\{(s_{0i}, \emptyset)\}_{i\in[1,n]}$ at time t_0 such that, for all $i \in [1, n]$, $\mathcal{A}_i$ is initially in state s_{0i} at time t_0. The finite, respectively infinite, semantics of initialized system $\mathcal{S}([s_0, t_0])$, is the component $[\![\mathcal{S}([s_0, t_0])]\!]^* = (E, \mathcal{L}^{\text{fin}*}(\mathcal{T}_\mathcal{S}, [s_0, t_0]))$, respectively $[\![\mathcal{S}([s_0, t_0])]\!] = (E, \mathcal{L}^{\text{inf}}(\mathcal{T}_\mathcal{S}, [s_0, t_0]))$, where $E = \bigcup_{i\in[1,n]} E_i$ with E_i the set of events for the agent component $[\![\mathcal{A}([s_{0i}, t_0])]\!]$.

Given a composability relation `comp`, we define the interaction signature $\Sigma = ([\kappa_{\texttt{comp}}], \cup)$, with $\kappa_{\texttt{comp}}(E_1, E_2) \subseteq (\mathcal{P}(E_1) \times \mathbb{N}) \times (\mathcal{P}(E_2) \times \mathbb{N})$ to be such that, for $\texttt{ai} : \texttt{Action}_\texttt{i}$ and $\texttt{aj} : \texttt{Action}_\texttt{j}$:

- if $\texttt{comp}(\texttt{ai}, \texttt{aj})$, then $((a_i, n), (a_j, n)) \in \kappa_{\texttt{comp}}(E_i, E_j)$ for all $n \in \mathbb{N}$, i.e., two composable actions occur at the same time;
- if $\texttt{comp}(\texttt{ai}, \star_\texttt{j})$, then $((a_i, n), (a, k)) \in \kappa_{\texttt{comp}}(E_i, E_j)$ for all $(a, k) \in \mathcal{P}(E_j) \times \mathbb{N}$ with $k \geq n$, i.e., $\mathcal{A}_j$ may have an action at arbitrary future time.

with E_i the set of events of agent $\mathcal{A}_i$.

Lemma 8 (Composability). *If* $\texttt{Action}_\texttt{i} \cap \texttt{Action}_\texttt{j} = \emptyset$ *for all disjoint agents* i *and* j*, then the product* $\times_{([\kappa_{comp}], \cup)}$ *is commutative and associative.*

Theorem 9 (Compositional semantics). *Let* $\mathcal{S} = (\mathcal{A}, \Lambda, \Omega, \mathcal{E}, \Rightarrow_S)$ *be a system of* n *agents with disjoint actions and* $[\{s_{01}, ..., s_{0n}\}, t_0]$ *as initial state. We fix* $\Sigma = ([\kappa_{comp}], \cup)$*. Then,* $[\![\mathcal{S}([s_0, t_0])]\!] = \times_\Sigma \{[\![\mathcal{A}_i([s_{0i}, t_0])]\!]\}_{i \in [1,n]}$ *and* $[\![\mathcal{S}([s_0, t_0])]\!]^* = \times_\Sigma \{[\![\mathcal{A}_i([s_{0i}, t_0])]\!]^*\}_{i \in [1,n]}$.

4 Application

We present the Maude implementation of the rewrite theories described in Section 3. We first describe our general framework as currently implemented in Maude, separating the agent modules, from the system module, and the composability relation. The framework is instantiated for a system consisting of two robot agents, each interacting with a (shared) field and a (private) battery agent (more details can be found in [10]). Finally, we run some analysis on the system using the Maude reachability search engine. The implementation of the framework in Maude can be found in [9].

4.1 General framework

Actions An action is a pair that contains the name of the action, and the set of agent identifiers on which the action applies. An agent action is identified by the source agent identifier, and is a triple `(id, (a; ids))` where `id` is the agent doing the action with name `a` onto the set of agents `ids`, that we call resources of agent `id` for action named `a`.

```
fmod ACTION is
    inc STRING . inc BOOL . inc SET{Id} . ...
    sort AName Action AgentAction .
    op (_;_) : AName Set{Id} -> Action [ctor] .
    op (_,_) : Id Action -> AgentAction [ctor] .
    op mta : -> AgentAction .
endfm
```

Agent The AGENT module in Listing 1.1 defines the theories on which an agent relies, the Agent sort, and operations that an agent instance must implement. The module is parametrized with a CSEMIRING theory, that is used to rank actions of an agent. Additionally, the AGENT includes modules that define state and action terms. A term of sort IdStates is a pair of an identifier and a map of sort MapKD.

A term of sort Agent is a tuple [id: C| state; ready?; softaction]. The identifier id is unique for each agent of the same class C. The state state of an agent is a map from keys to values. For instance, the state of a robot has three keys, position, energy, and lastAction, with values in Location, Status, and Bool. The flag ready? is of sort Bool and is True when the agent has submitted a possibly empty list of actions, and False otherwise. The pending actions softaction is a set of actions valued in the parametrized CSEMIRING. The use of a constraint semiring as a structure for action valuations enables various kinds of reasoning about preferences at the agent and system levels. We use the two operations of the csemiring, sum $+$ and product $\times$, as respectively modeling the choice and the compromise of two alternatives. See [6, 20, 21] for more details.

An agent instance implements four operations: computeActions, getOutput, getPostState, and internalUpdate. The operation computeActions, given a state:MapKD of agent id of class C, returns a set of valued actions in the parametrized CSEMIRING. The operation internalUpdate, given a state:MapKD of agent id of class C, returns a new state state':MapKD. For instance, an agent may record in its state, as an internal update, the outcome of computeActions and change the value that the key lastAction maps to. The getOutput operation, given an action name a:Name from agent identified by id2 applied to an agent id of class C in a state state, returns a collection of outputs outputs = getOutput(id, C, id2, an, state). The outputs generated by getOutput are of sort MapKD and therefore structured as a mapping from keys to values. For instance, the output of the action named read applied on a field agent has a key position that maps to the position value of the agent doing the read action. The operation getPostState, given an action name a:AName with inputs input:IdStates from agent identified by id2 applied on an agent id1 of class C in a state state, returns a new state state' = getPostState(id1, C, id2, an, input, state). The input input:IdStates is a collection of key to value mappings that results from collecting the outputs, i.e., with getOutput, of an action (id, an, ids) on all its resources in ids.

Listing 1.1. Extract from the AGENT Maude module.

```
fmod AGENT{X :: CSEMIRING} is
 inc IDSTATE .  inc ACTION .
 sort Agent .
 op [_:_|_;_;_] : Id Class MapKD Bool X$Elt -> Agent [ctor].
 op computeActions : Id Class MapKD -> X$Elt .
 op internalUpdate : Id Class MapKD -> MapKD .
 op getPostState : Id Class Id AName IdStates MapKD -> MapKD
 op getOutput : Id Class Id AName MapKD -> MapKD .
endfm
```

The agent's dynamics are given by the rewrite rule in Listing 1.2, that updates the pending action to select one atomic action from the set of valued actions:

Listing 1.2. Conditional rewrite rule applying on agent terms.

```
crl[agent] : [sys [id : ac | state  ; false ; null]] =>
      [sys [id : ac | state' ; true  ; softaction]]
    if softaction + sactions := computeActions(id, ac, state)
       /\ state' := internalUpdate(id, ac, state) .
```

The rewrite rule in Listing 1.2 implements the abstract rule of Equation 2. After application of the rewrite rule, the `ready?` flag of the agent is set to `True`. The agent may, as well, perform an internal update independent of the success of the selected action.

System The `SYSTEM` module in Listing 1.3 defines the sorts and operations that apply on a set of agents. The sort `Sys` contains set of `Agent` terms, and the term `Global` designates top level terms on which the system rewrite rule applies (as shown in Listing 1.4). The `SYSTEM` module includes the `Agent` theory parametrized with a fixed semiring `ASemiring`. The theory `ASemiring` defines valued actions as pairs of an action and a semiring value. While we assume that all agents share the same valuation structure, we can also define systems in which such a preference structure differs for each agent. The `SYSTEM` module defines three operations: `outputFromAction`, `updateSystemFromAction`, and `updateSystem`. The operation `outputFromAction` returns, given an agent action `(id, (an, ids))` applied on a system `sys`, a collection of identified outputs `idOutputs = outputFromAction((id, (an, ids)), sys)` given by the union of `getOutput` from all agents in `ids`. The operation `updatedSystemFromAction` returns, given an agent action `(id, (an, ids))` applied on a system `sys`, an updated system `sys' = updatedSystemFromAction((id, (an, ids)), sys)`. The updated system may raise an error if the action is not allowed by some of the resource agents in `ids` (see the battery-field-robot example in [10]). The updated system, otherwise, updates *synchronously* all agents with identifiers in `ids` by using the `getPostState` operation. The operation `updateSystem` returns, given a list of agent actions `agentActions` and a system term `sys`, a new system `updateSystem(sys, agentActions)` that performs a sequential update of `sys` with every action in `agentActions` using `updatedSystemFromAction`. The list `agentActions` ends with a delimiter action `end` performed on every agent, which may trigger an error if some expected action does not occur (see `PROTOCOL` in [10]).

Listing 1.3. Extract from the `SYSTEM` Maude module.

```
fmod SYS is
  inc AGENT{ASemiring} . sort Sys  Global .
  subsort Agent < Sys . op [_] : Sys -> Global [ctor] .
  op __ :  Sys Sys -> Sys [ctor assoc comm id: mt] .   ...
  op outputFromAction : AgentAction Sys -> IdStates .
  op updatedSystemFromAction : AgentAction Sys -> Sys .
  op updateSystem : Sys List{AgentAction} -> Sys .
endfm
```

The rewrite rule in Listing 1.4 applies on terms of sort `Global` and updates each agent of the system synchronously, given that their actions are composable. The rewrite rule in Listing 1.4 implements the abstract rule of Equation 4. The rewrite rule is conditional on essentially two predicates: `agentsReady?` and `kbestActions`. The predicate `agentsReady?` is `True` if every agent has its `ready?` flag set to `True`, i.e., the agent rewrite rule has already been applied. The operation `kbestActions` returns a ranked set of cliques (i.e., composable lists of actions), each paired with the updated system. The element of the ranked set are lists of actions containing at most one action for each agent, and paired with the system resulting from the application of `updateSystem`. If the updated system has reached a `notAllowed` state, then the list of actions is not composable and is discarded. The operations `getSysSoftActions` and `buildComposite` form the set of lists of composite actions, from the agent's set of ranked actions, by composing actions and joining their preferences.

Listing 1.4. Conditional rewrite rule applying on system terms.

```
crl[transition] : [sys]  => [sys']
 if agentsReady?(sys)  /\ saAtom := getSysSoftActions(sys) /\
  saComp := buildComposite(saAtom , sizeOfSum(saAtom)) /\
  p(actseq, sys') ; actseqs := kbestActions(saComp, k, sys) .
```

Composability relation The term `saComp` defines a set of valued lists of actions. Each element of `saComp` possibly defines a clique. The operation `kbestActions` specifies which, from the set `saComp`, are cliques. We describe below the implementation of `kbestActions`, given the structure of action terms.
An action is a triple `(id, (an, ids))`, where `id` is the identifier of the agent performing the action `an` on resource agents `ids`. Each resource agent in `ids` reacts to the action `(id, (an, ids))` by producing an output `(id', an, O)` (i.e., the result of `getOutput`). Therefore, $\texttt{comp}((\texttt{id}, (\texttt{an}, \texttt{ids})), \texttt{a}_\texttt{i})$ holds, with $\texttt{a}_\texttt{i} : \texttt{Action}_\texttt{i}$ and $i \in \texttt{ids}$, only if $\texttt{a}_\texttt{i}$ is a list that contains an output `(i, an, O)`, i.e., an output to the action. If one of the resources outputs the value `(i,notAllowed(an))`, the set is discarded as the actions are not pairwise composable. Conceptually, there are as many action names `an` as possible outputs from the resources, and the system rule (2) selects the clique for which the action name and the outputs have the same value. In practice, the list of outputs from the resources get passed to the agent performing the action.

4.2 Analysis in Maude

We analyze in Maude two scenarios. In one, each robot has as strategy to take the shortest path to reach its goal. As a consequence, a robot reads its position, computes the shortest path, and submits a set of optimal actions. A robot can sense an obstacle on its direct next location, which then allows for sub-optimal lateral moves (e.g., if the obstacle is in the direct next position in the West direction, the robot may go either North or South). In the other scenario, we add a protocol that swaps the two robots if robot `id(0)` is on the direct next location on the west of robot `id(1)`. The swapping is a sequence of moves that ends in an exchange of positions of the two robots. See [10] for details on the `TROLL`, `FIELD`, `BATTERY`, and `PROTOCOL` agents specified in Maude, and for the specification of the `init` term for both scenarios.

In the two scenarios, we analyze the behavior of the resulting system with two queries. The first query asks if the system can reach a state in which the energy level of the two batteries is 0, which means that its robot can no longer move:

```
search [1] init =>* [sys::Sys
       [ bat(1) : Battery | k(level) |-> 0 ; true ; null],
       [ bat(2) : Battery | k(level) |-> 0 ; true ; null]] .
```

The second query asks if the system can reach a state in which the two robots successfully reached their goals, and end in the expected locations:

```
search [1] init =>* [sys::Sys  [ field : Field | k(( 5 ; 5 ))
  |-> d(id(0)), k(( 0 ; 5 )) |-> d(id(1)) ; true ; null]] .
```

As a result, when the protocol is absent, the two robots can enter in a livelock behavior and eventually fail with an empty battery:

```
Solution 1 (state 80)
states: 81  rw: 223566 in 73ms cpu (74ms real) (3053554 rw/s)
```

Alternatively, when the protocol is used, the livelock is removed using exogenous coordination. The two robots therefore successfully reach their end locations, and stop before running out of battery:

```
No solution. states: 102
rewrites: 720235 in 146ms cpu (145ms real) (4920041 rw/s)
```

In both cases, the second query succeeds, as there exists a path for both scenarios where the two robots reach their end goal locations. The results can be reproduced by downloading the archive at [9].

5 Related work

Real-time Maude Real-Time Maude is implemented in Maude as an extension of Full Maude [18], and is used in applications such as in [8]. There are two ways to interpret a real-time rewrite theory, called the pointwise semantics and the continuous semantics. Our approach to model time is similar to the pointwise semantics for real-time Maude, as we fix a global time stamp interval before execution. The addition of a composability relation, that may discard actions to occur within the same rewrite step, differs from the real-time Maude framework.

Models based on rewriting logic In [21], the modeling of cyber-physical systems from an actor perspective is discussed. The notion of event comes as a central concept to model interaction between agents. Softagents [20] is a framework for specifying and analyzing adaptive cyber-physical systems implemented in Maude. It has been used to analyze systems such as vehicle platooning [4] and drone surveillance [13]. In Softagents agents interact by sharing knowledge and resources implemented as part of the system timestep rule.

Softagents only considers compatibility in the sense of reachability of desired or undesired states. Our approach provides more structure enabling static analysis. Our framework allows, for instance, to consider compatibility of a robot with a battery (i.e., changing the battery specification without altering other agents in the system), and coordination of two robots with an exogenous protocol, itself specified as an agent.

Algebra, co-algebra The algebra of components described in this paper is an extension of [12]. Algebra of communicating processes [5] (ACP) achieves similar objectives as decoupling processes from their interaction. For instance, the encapsulation operator in process algebra is a unary operator that restricts which actions may occur, i.e., $\delta_H(t \parallel s)$ prevents t and s to perform actions in H. Moreover, composition of actions is expressed using communication functions, i.e., $\gamma(a, b) = c$ means that actions a and b, if performed together, form the new action c. Different types of coordination over communicating processes are studied in [2].

Discrete Event Systems Our work represents both cyber and physical aspects of systems in a unified model of discrete event systems [1,17]. In [7], the author lists the current challenges in modelling cyber-physical systems in such a way. The author points to the problem of modular control, where even though two modules run without problems in isolation, the same two modules may block when they are used in conjunction. In [19], the authors present procedures to synthesize supervisors that control a set of interacting processes and, in the case of failure, report a diagnosis. An application for large scale controller synthesis is given in [16]. Our framework allows for experiments on modular control, by adding an agent controller among the set of agents to be controlled. The implementation in Maude enables the search of, for instance, blocking configurations.

6 Conclusion

We give an operational specification of the algebra of components defined in [12]. An agent specifies a component as a rewrite theory, and a system specifies a product of components as a set of rewrite theories extended with a composability relation. We show compositionality, i.e., that the system specifies a component that equals to the product, under a suitable interaction signature, of components specified by each agent.

We present an implementation of our framework in Maude, and instantiate a set of components to model two energy sensitive robots roaming on a shared field. We analyze the behavior of the resulting system before and after coordination with a protocol, and show how the protocol can prevent livelock behavior.

The modularity of our operational framework and the interpretation of agents as components in interaction add structure to the design of cyber-physical systems. The structure can therefore be exploited to reason about more general properties of CPSs, such as compatibility, sample period synthesis, etc.

Acknowledgement Talcott was partially supported by the U. S. Office of Naval Research under award numbers N00014-15-1-2202 and N00014-20-1-2644, and NRL grant N0017317-1-G002. Arbab was partially supported by the U. S. Office of Naval Research under award number N00014-20-1-2644.

References

1. 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.
2. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, 1984.
3. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
4. Yuri Gil Dantas, Vivek Nigam, and Carolyn L. Talcott. A formal security assessment framework for cooperative adaptive cruise control. In *IEEE Vehicular Networking Conference*, pages 1–8. IEEE, 2020.
5. Wan J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000.
6. Tobias Kappé, Benjamin Lion, Farhad Arbab, and Carolyn Talcott. Soft component automata: Composition, compilation, logic, and verification. *Science of Computer Programming*, 2019.
7. Stéphane Lafortune. Discrete event systems: Modeling, observation, and control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2(1):141–159, 2019.
8. Jaehun Lee, Sharon Kim, Kyungmin Bae, and Peter Csaba Ölveczky. Hybridsynchaadl: Modeling and formal analysis of virtually synchronous CPSs in AADL. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 491–504. Springer, 2021.
9. Benjamin Lion. Cyber-physical framework in maude. `http://dx.doi.org/10.5281/zenodo.6587173`, May 2022.
10. Benjamin Lion, Farhad Arbab, and Carolyn Talcott. A rewriting framework for interacting cyber-physical agents, 2022.
11. Benjamin Lion, Farhad Arbab, and Carolyn Talcott. Runtime composition of systems of interacting cyber-physical components, 2022.
12. Benjamin Lion, Farhad Arbab, and Carolyn L. Talcott. A semantic model for interacting cyber-physical systems. In Julien Lange, Anastasia Mavridou, Larisa Safina, and Alceste Scalas, editors, *Proceedings 14th Interaction and Concurrency Experience, ICE 2021, Online, 18th June 2021*, volume 347 of *EPTCS*, pages 77–95, 2021.
13. Ian A. Mason, Vivek Nigam, Carolyn L. Talcott, and Alisson Vasconcelos De Brito. A framework for analyzing adaptive autonomous aerial vehicles. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods Collocated*

Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Revised Selected Papers, volume 10729 of *Lecture Notes in Computer Science*, pages 406–422. Springer, 2017.

14. José Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.
15. José Meseguer. Twenty years of rewriting logic. *J. Log. Algebraic Methods Program.*, 81(7-8):721–781, 2012.
16. L. Moormann, J.M. van de Mortel-Fronczak, W.J. Fokkink, P. Maessen, and J.E. Rooda. Supervisory control synthesis for large-scale systems with isomorphisms. *Control Engineering Practice*, 115:104902, 2021.
17. Maurice Nivat. *Behaviors of Processes and Synchronized Systems of Processes*, pages 473–551. Springer Netherlands, Dordrecht, 1982.
18. Peter Csaba Ölveczky. Real-time maude and its applications. In Santiago Escobar, editor, *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 42–79. Springer, 2014.
19. Meera Sampath, Stéphane Lafortune, and Demosthenis Teneketzis. Active diagnosis of discrete-event systems. *IEEE Trans. Autom. Control.*, 43(7):908–929, 1998.
20. Carolyn Talcott, Farhad Arbab, and Maneesh Yadav. Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, volume 8950 of *LNCS*. Springer, 2015.
21. Martin Wirsing, Grit Denker, Carolyn L. Talcott, Andy Poggio, and Linda Briesemeister. A rewriting logic framework for soft constraints. *Electr. Notes Theor. Comput. Sci.*, 176(4):181–197, 2007.