# Real-time Rewriting Logic Semantics for Spatial Concurrent Constraint Programming

Sergio Ramírez, Miguel Romero, Camilo Rocha, Frank Valencia

# Real-time Rewriting Logic Semantics for Spatial Concurrent Constraint Programming

Sergio Ramírez[1], Miguel Romero[1], Camilo Rocha[1], and Frank Valencia[1],[2]

[1] Pontificia Universidad Javeriana de Cali, Colombia
[2] CNRS, LIX École Polytechnique de Paris, France

**Abstract.** Process calculi provide a language in which the structure of *terms* represents the structure of processes together with an *operational semantics* to represent computational steps. This paper uses rewriting logic for specifying and analyzing a process calculus for *concurrent constraint programming* (ccp), combining spatial and real-time behavior. In these systems, agents can run processes in different computational spaces (e.g., containers) while subject to real-time requirements (e.g., upper bounds in the execution time of a given operation), which can be specified with both discrete and dense linear time. The real-time rewriting logic semantics is fully executable in Maude with the help of rewriting modulo SMT: partial information (i.e., constraints) in the specification is represented by quantifier-free formulas on the shared variables of the system that are under the control of SMT decision procedures. The approach is used to symbolically analyze existential real-time reachability properties of process calculi in the presence of spatial hierarchies for sharing information and knowledge.

## 1 Introduction

*Concurrent constraint programming* (ccp) [26] is a well-established process model for concurrency based upon the shared-variables communication model. Its basic intuitions arise mostly from logic; in fact, ccp processes can be interpreted both as concurrent computational entities and logic specifications (e.g., process composition can be seen as parallel execution and as conjunction). In ccp, agents can interact by *posting* (or *telling*) partial information in a medium such as a centralized *store*. Partial information is represented by constraints (e.g., $x > 42$) on the shared variables of the system. The other way in which agents can interact is by *querying* (or *asking*) about partial information entailed by the store. This provides the synchronization mechanism of the model: asking agents are suspended until there is enough information in the store to answer their query. As other mature models of concurrency, ccp has been extended to capture aspects such as mobility [4,9,12], stochastic behavior [10], and —most prominently— temporal behavior [6,11,16,23,25] for timed and reactive computations, where processes can be constrained also by unit delays and time-out conditions.

However, due to their centralised notion of store, all the previously-mentioned extensions are unsuitable for today's systems where information and processes can be spatially distributed among certain groups of agents. Examples of these systems include agents posting and querying information in the presence of *spatial hierarchies*

*for sharing information and knowledge*, such as friend circles and shared albums in social networks, or shared folders in cloud storage. Recently, the authors of [13] enhanced and generalized the theory of ccp for systems with spatial distribution of information in the novel *spatial concurrent constraint programming* (sccp), where computational hierarchical spaces can be assigned to belong to agents. In sccp, each space may have ccp processes and other (sub) spaces, processes can post and query information in their given space (i.e., locally), and may as well move from one space to another.

As an example, consider the tree-like structures depicted in Figure 1. They correspond to hierarchical computational spaces of, e.g., virtual containerization (i.e., virtual machines inside other virtual machines). Each one of these spaces is endowed with an agent identifier (either root or a natural number) and a local store (i.e., a constraint), and the processes can be executed and spawned concurrently inside any space, with the potential to traverse the structure, querying and posting information locally, and even creating new spaces. The sccp calculus enables the formal modeling of such scenarios and of transitions that can lead from an initial system state (e.g., Figure 1a) to a final state (e.g., Figure 1b) by means of an operational semantics [13].
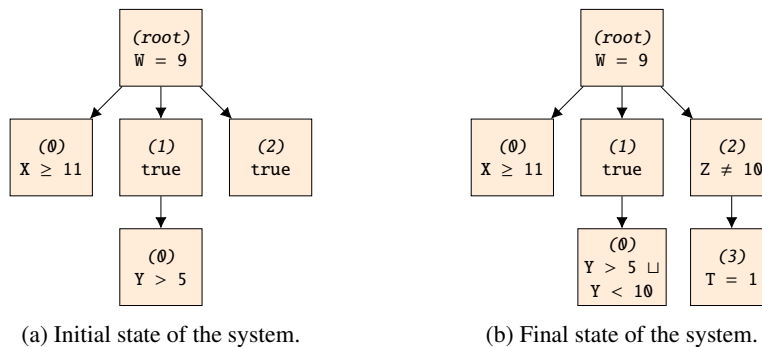


(a) Initial state of the system.     (b) Final state of the system.

Fig. 1: A containerization example.

This paper presents the *real-time sccp* calculus (rtsccp), a generalization of sccp with timing constraints. The intended models of rtsccp are spatially-distributed multi-agent reactive systems that may have different computing capabilities (e.g., virtual containers with heterogeneous bandwidth and main memory configurations) and be subject to real-time requirements (e.g., upper bounds in the execution time of a given operation). The formal semantics of rtsccp is given in the form of a real-time rewriting logic semantics that is executable in the Maude system [5]. As such, the real-time rewriting logic specification can be subject to automatic reachability analysis and LTL model-checking, and thus enables the formal analysis of timing behavior for agents distributed in hierarchical spaces, such as fault-tolerance and consistency.

In the rtsccp real-time rewriting logic semantics, flat configurations of object-like terms encode the hierarchical structure of spaces, and equational and rewrite rules both axiomatize the concurrent computational steps of processes. Time attributes are associ-

ated to process-store interaction, as well as to process mobility in the space structure, by means of maps from agents to non-negative real quantities; these choices can be interpreted to denote, as previously mentioned, upper bounds in the execution time of the given operations. The underlying constraint system of sccp is materialized with the help of the rewriting modulo SMT [21] approach, with constraints being quantifier-free formulas over Boolean and integer shared variables, and information entailment queried as semantic inference and automatically delivered by the SMT-based decision procedures. The main contribution of this work can also be seen as yet another interesting use of rewriting logic as a semantic framework: the support in rewriting logic for real-time systems [18] and open systems [21], make of the rtsccp real-time rewriting logic semantics a symbolic and fully executable specification in Maude, that is both sound and complete (relative to initial semantics) for reachability analysis in spatial constraint systems with discrete and dense linear timing constraints.

*Outline.* Section 2 summarizes some preliminaries on constraint systems and rewriting logic. The real-time rewriting logic semantics for rtsccp is presented in Section 3. The use of the semantics for symbolic reachability analysis is illustrated with some examples in Section 4. Additional related work and concluding remarks are included in Section 5. The rewriting logic specification for rtsccp and the examples are available at

$$\texttt{http://escher.javerianacali.edu.co/rtsccpe/index.html}$$

## 2 Preliminaries

This section briefly explains the basics of constraint and spatial constraint systems with extrusion. It also presents a summary of order-sorted rewriting logic [14], a semantic framework that unifies a wide range of models of concurrency.

A *constraint system* (cs) $\mathbf{C}$ is a complete algebraic lattice $\mathbf{C} = (Con, \sqsubseteq)$, where the elements in *Con* are called *constraints* and the order $\sqsubseteq$ *entailment of information*: $d \sqsubseteq c$ (or, $c \sqsupseteq d$) asserts that the constraint $c$ contains at least as much information as the constraint $d$. The symbols $\sqcup$, *true*, and *false* denote the least upper bound (*lub*) operation, the bottom, and the top element of $\mathbf{C}$, respectively. An $n$-agent *spatial constraint system* (*n*-scs) [13] $\mathbf{C}$ is a cs $(Con, \sqsubseteq)$ equipped with $n$ self-maps $[\cdot]_1, \ldots, [\cdot]_n$ over its set of constraints *Con* satisfying, for each function $[\cdot]_i : Con \rightarrow Con$: $[true]_i = true$ and $[c \sqcup d]_i = [c]_i \sqcup [d]_i$, for each $c, d \in Con$. An *n-agent spatial constraint system with extrusion (n-scse)* [12] is an *n*-scs $\mathbf{C}$ equipped with $n$ self-maps $\uparrow_1, \ldots, \uparrow_n$ over *Con*, written $(\mathbf{C}, \uparrow_1, \ldots, \uparrow_n)$, such that each $\uparrow_i$ is the right inverse of $[\cdot]_i$.

A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E \uplus B, R)$ with: (i) $(\Sigma, E \uplus B)$ an order-sorted equational theory with signature $\Sigma$, $E$ a set of equations over $T_\Sigma$, and $B$ a set of structural axioms – disjoint from the set of equations $E$ – over $T_\Sigma$ for which there is a finitary matching algorithm (e.g., associativity, commutativity, and identity, or combinations of them); and (ii) $R$ a finite set of (possibly conditional) rewrite rules over $T_\Sigma(X)$. Intuitively, $\mathcal{R}$ specifies a concurrent system whose states are elements of the set $T_{\Sigma/E\uplus B}$ of $\Sigma$-terms modulo $E \uplus B$ and whose concurrent transitions are axiomatized by the rules $R$ according to the inference rules of rewriting logic [3]. In particular, for $t, u \in T_\Sigma$ representing states of the concurrent system described by $\mathcal{R}$, a transition from $t$ to $u$ is

captured by a formula of the form $t \longrightarrow_{\mathcal{R}} u$; the symbol $\longrightarrow_{\mathcal{R}}$ denotes the binary rewrite relation induced by $R$ over $T_{\Sigma/E \uplus B}$ and $\mathcal{T}_{\mathcal{R}} = (T_{\Sigma/E \uplus B}, \longrightarrow_{\mathcal{R}})$ denotes the *initial reachability model* of $\mathcal{R}$.

The rewriting logic semantics of a language $\mathcal{L}$ is a rewrite theory $\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}} \uplus B_{\mathcal{L}}, R_{\mathcal{L}})$ where $\longrightarrow_{\mathcal{R}_{\mathcal{L}}}$ provides a step-by-step formal description of $\mathcal{L}$'s *observable* run-to-completion mechanisms. The conceptual distinction between equations and rules in $\mathcal{R}_{\mathcal{L}}$ has important consequences that are captured by rewriting logic's *abstraction dial* [15]. Setting the level of abstraction in which all the interleaving behavior of evaluations in $\mathcal{L}$ is observable, corresponds to the special case in which the dial is turned down to its minimum position by having $E_{\mathcal{L}} \uplus B_{\mathcal{L}} = \emptyset$. The abstraction dial can also be turned up to its maximal position as the special case in which $R_{\mathcal{L}} = \emptyset$, thus obtaining an equational semantics of $\mathcal{L}$ without observable transitions. The rewriting logic semantics presented in this paper is *faithful* in the sense that such an abstraction dial is set at a position that exactly captures the interleaving behavior of the concurrency model.

The real-time rewrite theory presented in this work is *time-robust*, namely: (i) in any given state, time can advance either any amount up to a specific instant in time or not at all; and (ii) instantaneous rules (i.e., those that are not tick rules and are supposed to take zero time) can only be applied when the system has advanced the maximal possible amount of time before any timed action can become enabled. Under these two assumptions and by using the maximal time sampling strategy, unbounded and time-bounded search and model checking are sound and complete with respect to *timed fair paths* [19]. They exclude paths with an infinite sequence of tick steps where, at each step, time could have advanced to time $r$ (the duration of the first step in a path) or beyond, but with a total path duration less than $r$. Also are excluded those 'unfair' paths containing an infinite and consecutive sequence of 0-time ticks over a state on which an instantaneous rule can be applied. Note that a time-robust system may have Zeno paths, where the sum of the durations of an infinite number of tick steps is bounded. By restricting the computations to time-bounded prefixes only a finite set of states can be reached from an initial state, so that the target real-time specification does not exhibit any Zeno behavior and temporal properties can be model checked.

Satisfiability Modulo Theories (SMT) studies methods for checking satisfiability of first-order formulas in specific models. In this work, the representation of the constraint system is based on SMT solving technology. Given an many-sorted equational theory $\mathcal{E}_0 = (\Sigma_0, E_0)$ and a set of variables $X_0 \subseteq X$ over the sorts in $\Sigma_0$, the formulas under consideration are in the set $QF_{\Sigma_0}(X_0)$ of quantifier-free $\Sigma_0$-formulas: each formula being a Boolean combination of $\Sigma_0$-equation with variables in $X_0$ (i.e., atoms). The terms in $T_{\Sigma_0/E_0}(X)$ are called *built-ins* and represent the portion of the specification that will be handled by the SMT solver (i.e., they are semantic data types). In this setting, an SMT instance is a formula $\phi \in QF_{\Sigma_0}(X_0)$ and the initial algebra $\mathcal{T}_{\mathcal{E}_0^+}$, where $\mathcal{E}_0^+$ is a *decidable extension* of $\mathcal{E}_0$ such that $\phi$ is satisfiable in $\mathcal{T}_{\mathcal{E}_0^+}$ iff there exists $\sigma : X_0 \longrightarrow T_{\Sigma_0}$ such that $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$.

Maude [5] is a language and tool supporting the formal specification, execution, and analysis of concurrent systems specified as rewrite theories, including those with real-time semantics (see [19]) and those with built-ins as proposed in the rewriting modulo SMT approach (see [21]).

# 3 Rewriting Logic Semantics

This section introduces the rtsccp real-time rewriting logic semantics in the form of a real-time rewrite theory $\mathcal{R}$, detailing some aspects of to its syntax and transitions.

Figure 2 depicts the module structure of $\mathcal{R}$, where a triple-line arrow ($\Rightarrow$) represents module importation by protecting and a single-line arrow ($\rightarrow$) module importation by inclusion. The difference between these two importing modes is that the former allows surjectivity (*junk*) and injectivity (*confusion*) [5].
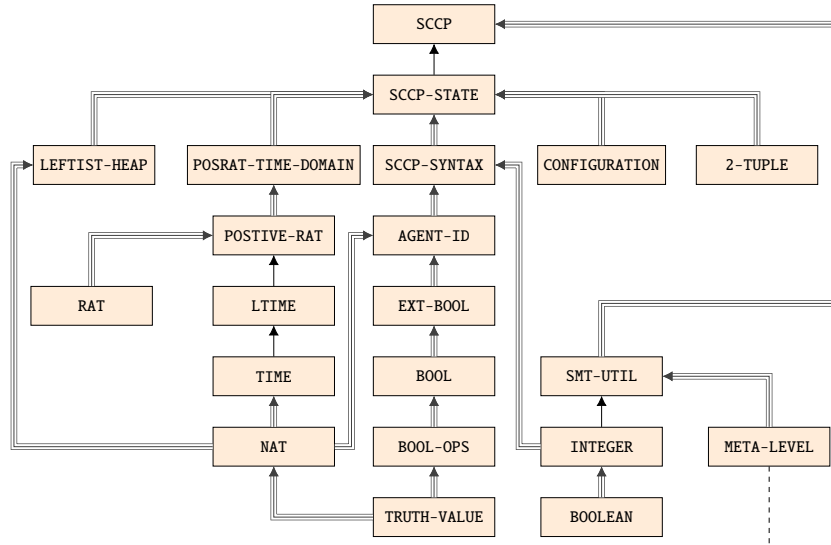


Fig. 2: Module hierarchy of rtsccp

## 3.1 System States

The tree-like structure of the hierarchical spaces is represented as a flat configuration of object-like terms encoding the state of execution of the agents. The hierarchical relationships among spaces are specified by common prefixes as part of an agent's name. In an observable state, each agent's space is represented by a set of object-like terms: some encoding the state of execution of all its processes and exactly one object representing its local store. The object-based system is represented using Maude's predefined module `CONFIGURATION` imported in `SCCP-STATE`. The object and class identifiers are:

```
subsorts Nat Aid < Oid .
ops agent process Simulation :    -> Cid .
op {_}          : Configuration   -> Sys .
```

The system states are represented by the topsort `Sys` with argument a configuration of objects containing the setup of each one of the agents in the system. A `Configuration`

is a multiset of objects with set union denoted by juxtaposition and identity `none`. There are two types of object identifiers: agent identifiers (`Aid`) for identifying agents and their hierarchical structure, and natural numbers (`Nat`) for some additional identification used internally in $\mathcal{R}$. There are three types of class identifiers, namely, for agents, processes, and a simulation object. A simulation object specifies the attributes required for the real-time simulation of the system, such as the global time and the scheduler.

Each agent has one attribute, namely, its a store, and each process has two attributes: an universal identifier (used internally for execution purposes) and the command (i.e., process) it is executing:

```
op store :_    : Boolean        -> Attribute [ctor] .
op Uid :_      : Nat            -> Attribute [ctor] .
op command :_  : SCCPCmd        -> Attribute [ctor] .
```

The syntax of commands is presented in Section 3.2. Section 3.3 explains how quantifier free formulas are used to represent constraints (as `Boolean`) and the entailment relation is encoded with the help of SMT-based decisions procedures.

Finally, the attributes of the simulation object include the global time (attribute `gtime`); the priority queue of the system commands to be processed, ordered by time-to-execution (attribute `pqueue`); the collection of pending commands, i.e., ask commands that are waiting for its guarding constraint to become activate (attribute `pend`); the counter for assigning the next internal identifier when spawning a new process (attribute `nextID`); a flag that is on whenever a tick rule needs to be applied (attribute `flg`); and a collection of maps containing the time it takes to process certain commands relative to the space where they are executed (attributes `MAsk`, `MTell`, `MSp`, and `MExt`). The sort `Time`, as it is often the case in Real-time Maude [18], can be used to represent either discrete or dense linear time, while `Ttime` is the name of the Maude view that is used to instantiate parameterized sorts with time:

```
op gtime :_    : Time           -> Attribute [ctor] .
op pqueue :_   : Heap{Tuple}    -> Attribute [ctor] .
op pend :_     : Heap{Tuple}    -> Attribute [ctor] .
op nextID :_   : Nat            -> Attribute [ctor] .
op flg :_      : Bool           -> Attribute [ctor] .
op MAsk :_     : Map{Aid, Ttime}  -> Attribute [ctor] .
op MTell :_    : Map{Aid, Ttime}  -> Attribute [ctor] .
op MSP :_      : Map{Aid, Ttime}  -> Attribute [ctor] .
op MExt :_     : Map{Aid, Ttime}  -> Attribute [ctor] .
```

As mentioned before, the qualified identifiers of agents are used to encode the hierarchical structure of spaces (sort `Aid`). The root of any tree is denoted by constant `root` and any other qualified name corresponds to a dot-separated list natural numbers (sort `Nat`), organized from left to right:

```
op root : -> Aid .
op _._  : Nat Aid -> Aid .
```

*Example 1.* In this syntax, the container system depicted in Figure 1a can be specified as follows:

```
< root : agent | store : (W:Integer === (9).Integer) >
< 0 . root : agent | store : (X:Integer >= 11) >
< 0 . 1 . root : agent | store : (Y:Integer > 5) >
< 1 . root : agent | store : true >
< 2 . root : agent | store : true >
```

## 3.2  Commands

The following EBNF-like notation defines the process-like syntax of commands:

$$P ::= \mathbf{0} \mid \mathbf{tell}(c) \mid \mathbf{ask}(c) \to P \mid P \parallel P \mid [P]_i \mid \uparrow_i (P)$$

where $c$ is a constraint and $i$ an agent identifier. The *tell*$(c)$ command posts the constraint $c$ to the local store (once a constraint is added, it cannot be removed from the store so that the store grows monotonically). The command $ask(c) \to P$ queries if $c$ can be deduced from the information in the local store; if so, the agent behaves like $P$, otherwise, it remains blocked until more information is added to the store. A basic ccp process like-language usually adds *parallel composition* $(P \parallel Q)$ combining processes $P$ and $Q$ concurrently. The command $[P]_i$ indicates that command $P$ must be executed inside the agent $i$'s space: any information that $P$ produces is available to other commands that execute within the same space. The command $\uparrow_i (P)$ denotes that $P$ is to be run outside the space of agent $i$ and the information posted by $P$ is going to be stored in the parent of agent $i$. The SCCP-SYNTAX module includes the syntax of commands in rtsccp:

```
op 0      :                     -> SCCPCmd .
op tell   : Boolean             -> SCCPCmd .
op ask_->_ : Boolean SCCPCmd -> SCCPCmd .
op _||_    : SCCPCmd SCCPCmd -> SCCPCmd [assoc comm gather (e E) ] .
op _in_    : SCCPCmd Nat      -> SCCPCmd .
op _out_   : SCCPCmd Nat      -> SCCPCmd .
```

## 3.3  Time Scaffolding

The real-time behavior in $\mathcal{R}$ associates timing behavior to those commands that interact with stores (i.e., tell and ask commands) and to commands that involve mobility among the space structure of the system (i.e., [_]_ and ↑_(_)). More precisely, tell and ask commands can take time when posting and querying, respectively, from a store. Moving the execution of a command inside an agent and extruding from a space can also take up time. Such times are given by the time maps MTell (for *tell*), MAsk (for *ask*), MSp (for [_]_), and MExt (for ↑_(_)), and can be consulted using the getTimeCmd function. For example, MTell(i) denotes the time it takes to execute a tell command inside the agent's $i$ space.

```
op fTime : Map{Aid, Ttime} Aid -> Time .
eq fTime(M:Map{Aid, Ttime}, L1)
 = if $hasMapping(M, L1) then M[L1] else 0 fi .
```

```
op getTimeCmd : Attribute Attribute Attribute Attribute
                                        SCCPCmd Aid -> Time .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, tell(B1), L1)
 = fTime(MT, L1) .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, C1 in I1, L1)
 = fTime(MI, L1) .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, C1 out I1, L1)
 = fTime(MO, L1) .
eq getTimeCmd(MTell: MT, MAsk: MA, MIn: MI, MOut: MO, C1, L1)
 = 0 [owise] .
```

The run-to-completion time of commands is simulated with the help of a leftist heap that keeps track of all the active commands that are waiting for the global timer to advance. One motivation to use leftist heaps is that insertion, removal, and querying are defined without the need of structural axioms, which may result in performance gains during execution. A *leftist heap* [17] is a heap-ordered binary tree that satisfies the *leftist property*: the rank (i.e., the length of its rightmost path to a leaf) of any left child is at least as large as the rank of its right sibling. Each entry in the heap is a pair $(i, t)$ where $i$ is an unique identifier of a process and $t$ the time it needs to start executing. At the beginning, all the processes belong to the heap and they are ordered with respect to their execution time. A process is executed when its execution time is the minimum time of all the processes that are pending to complete their transitions. The leftist heap is implemented as a parameterized container in the functional module LEFTIST-HEAP{X :: STRICT-TOTAL-ORDER}, with admissible parameters only being strict total orders:

```
sort Heap{X} NeHeap{X} .
subsort NeHeap{X} < Heap{X} .
op empty : -> Heap{X} [ctor] .
op T(_,_,_,_) : Nat X$Elt Heap{X} Heap{X} -> NeHeap{X} [ctor] .
op isEmpty : Heap{X} -> Bool .
eq isEmpty(empty)      = true .
eq isEmpty(T(Ra,E,L,R)) = false .
op rank    : Heap{X} -> Nat .
eq rank(empty)       = 0 .
eq rank(T(Ra,E,L,R)) = Ra .
op makeT   : X$Elt Heap{X} Heap{X} -> NeHeap{X} .
eq makeT(E,L,R)
 = if rank(L) >= rank(R)
   then T(rank(R) + 1,E,L,R)
   else T(rank(L) + 1,E,R,L)
   fi .
```

Heaps are constructed from the constant empty and the T(_,_,_,) function symbol: the first argument is the rank of the tree (sort Nat), the second one the label (sort X$Elt), and the third and fourth ones the left and right children (sort Heap{X}), respectively. In the semantics of rtsccp, the sort X$Elt is instantiated with the sort of pairs of the form $(I, T)$, where $I$ is an internal process identifier and $T$ is the run-completion time of such

a process. Auxiliary operations include `isEmpty`, `rank`, and `makeT`, which are used to verify whether a heap is empty, compute the rank of a given heap, and create a heap out of two heaps, respectively. Other key operations on leftist heaps are the merging of two heaps (function `merge`), inserting an element in a heap (function `insert`), removing an element from a heap (function `deleteMin`), and finding the element at the top of a non-empty heap (function `findMin`).

```
op merge     : Heap{X} Heap{X} -> Heap{X} .
eq merge(empty, L) = L .
eq merge(L, empty) = L .
eq merge(T(Ra,E,L,R),T(Ra',E',L',R'))
 = if (E < E' or E == E')
   then makeT(E,L,merge(R,T(Ra',E',L',R')))
   else makeT(E',L',merge(T(Ra,E,L,R),R'))
   fi .
op insert    : X$Elt Heap{X} -> NeHeap{X} .
eq insert(E,L) = merge(T(1,E,empty,empty),L) .
op deleteMin : NeHeap{X} -> Heap{X} .
eq deleteMin(T(Ra,E,L,R)) = merge(L,R) .
op findMin   : NeHeap{X} -> X$Elt .
eq findMin(T(Ra,E,L,R)) = E .
```

### 3.4 The Constraint System

In this rewriting logic semantics, the sort `Boolean` (available in the current version of Maude from the `INTEGER` module) defines the data type used to represent rtsccp's constraints. Terms of sort `Boolean` are quantifier-free formulas built from variables ranging over the Booleans and integers, and the usual function symbols. The current version of Maude is integrated with the CVC4 [2] and Yices2 [8] SMT solvers, which can be queried via the meta-level. In this semantics, queries to the SMT solvers are encapsulated by functions `check-sat` and `check-unsat`:

```
op check-sat : Boolean -> Bool .
eq check-sat(B) = metaCheck(['INTEGER], upTerm(B)) .
op check-unsat : Boolean -> Bool .
eq check-unsat(B) = not(check-sat(B)) .
```

The function invocation `check-sat`($B$) returns true only if $B$ is satisfiable. Otherwise, it returns false if it is unsatisfiable or undefined if the SMT solver cannot decide. Note that function invocation `check-unsat`($B$) returns true only if $B$ is unsatisfiable. Therefore, the rewriting logic semantics of rtsccp instantiates the constraint system $\mathbf{C} = (Con, \sqsubseteq)$ by having quantifier-free formulas, modulo the semantic equivalence in $\mathcal{T}_{\mathcal{E}_0^+}$ (i.e., the model implemented in the SMT solver extending the initial model $\mathcal{T}_{\mathcal{E}_0}$), as the constraints $Con$ and semantic validity relative to $\mathcal{T}_{\mathcal{E}_0^+}$ as the entailment relation $\sqsubseteq$. More precisely, if $\Gamma$ is a finite set of terms of sort `Boolean` and $\phi$ is term of sort `Boolean`, the following equivalence holds: $\Gamma \sqsubseteq \phi$ iff `check-unsat`$\left( \left( \bigwedge_{\gamma \in \Gamma} \gamma \right) \wedge \neg \phi \right)$. In order to make a direct relation between the entailment relation $\sqsubseteq$ and the Maude syntax, the operator `entails` is defined as follows:

```
op entails : Boolean Boolean -> Bool .
eq entails(C1:Boolean, C2:Boolean)
 = check-unsat(C1:Boolean and not(C2:Boolean)) .
```

### 3.5 System Transitions

The tick rule models time elapse in the system [18]:

```
crl [tick] :
  { X < I : Simulation | pqueue : P, gtime : T, flg : true,
            pend : P0, Atts > }
=> { X < I : Simulation | pqueue : merge(delta(deleteMin(P),T0),P0),
            gtime : (T plus T0), flg : false, pend : empty, Atts > }
  if T0 := p2(findMin(P)) .
```

where the the auxiliary operation `delta` reduce `T0` units the execution time of every command in the heap P:

```
op delta : Heap{Tuple} Time -> Heap{Tuple} .
eq delta(empty,T') = empty .
eq delta(T(N,((I,T)),P,P0),T')
 = T(N,((I,T monus T')),delta(P,T'),delta(P0,T')) .
```

When the `[tick]` rule is fired, the global time is incremented in `T0` units, where `T0` is the minimum time present in the priority queue P, which is modified by removing the process with the minimum execution time. It also adds the pending commands to the priority queue. The pending commands are ask commands that, although they have been activated already for execution, have not been able to execute because their guard has not been met by the state of the corresponding local stores. The tick rule puts all these pending process back in the main queue, so that their guards can be checked again and be executed or put back in the pending queue. Figure 3 depicts the possible transitions an ask command can take between being in the priority queue, in the pending queue, and finally executing. The rules `[ask]` and `[delay]` are introduced below.
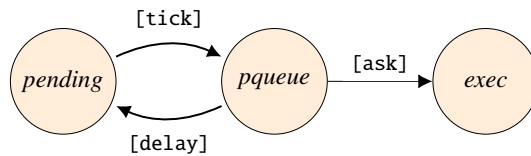


Fig. 3: Possible transitions for ask commands.

The invisible transitions of the semantics are specified with the help of equational rules. Namely, one for removing a `0` command from a configuration and another one to join the contents of two stores of the same space (i.e., two stores with the same `Aid`). The latter type of transition is important because when a new process is spawned in a

agent's space, a store with the empty constraint (i.e., `true`) is created for that space. If such a space existed before, then the idea is that the newly created store is subsumed by the existing one. Note that neither of the invisible transitions takes time, i.e., they are really instantaneous, and they axiomatize structural properties of commands.

```
eq { < L0: process | command: 0, Atts > X }
 = { none X } .
eq < L0: agent | store: B0 > < L0: agent | store: B1 >
 = < L0: agent | store: (B0 and B1) > .
```

The following six rules capture the concurrent observable behavior in $\mathcal{R}$:

```
crl [tell]:
    { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: tell (B1) >
      < I: Simulation | pqueue: H, flg: false, pend: P, Atts > X }
 => { < L0: agent | store: (B0 and B1) >
      < I: Simulation | pqueue: H, flg: true, pend: P, Atts > X }
 if I0 == p1(findMin(H)) .

crl [parallel]:
    { < L0: process | UID: I0, command: (C0 || C1) >
      < I: Simulation | pqueue: H, nextID: N, flg: false, pend: P,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < L0: process | UID: N, command: C0 >
      < L0: process | UID: (N + 1), command: C1 >
      < I: Simulation | pqueue: H, nextID: (N + 2), flg: true,
        pend: H0, MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ H0 := insert(((N, getTimeCmd(MTell: MT, MAsk: MA, MIn: MI,
                                MOut: MO, C0, L0))),
          insert(((N + 1, getTimeCmd(MTell: MT, MAsk: MA, MIn: MI,
                                     MOut: MO, C1, L0))), P)) .

crl [space]:
    { < L0: process | UID: I0, command: (C0 in N0) >
      < I: Simulation | pqueue: H, nextID: N, flg: false, pend: P,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < N0 . L0: agent | store: true >
      < N0 . L0: process | UID: N, command: C0 >
      < I: Simulation | pqueue: H, flg: true, pend: H0, nextID: (N+1),
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ H0:= insert(((N, getTimeCmd(MTell: MT,
                MAsk: MA, MIn: MI, MOut: MO, C0, L0))), P) .

crl [extrussion]:
    { < N0 . L0: process | UID: I0, command: (C0 out N0) >
      < I: Simulation | pqueue: H, nextID: N, flg: false, pend: P,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
```

```
=> { < L0: process | UID: N, command: C0 >
     < I: Simulation | pqueue: H, flg: true, pend: H0, nextID: (N+1),
       MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ H0:= insert(((N, getTimeCmd(MTell: MT,
                  MAsk: MA, MIn: MI, MOut: MO, C0, N0 . L0))), P) .

crl [ask]:
    { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: (ask B1 -> C1) >
      < I: Simulation | pqueue: H, flg: false, pend: P, nextID: N,
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 => { < L0: agent | store: B0 >
      < L0: process | UID: N, command: C1 >
      < I: Simulation | pqueue: H, flg: true, pend: H0, nextID: (N+1),
        MTell: MT, MAsk: MA, MIn: MI, MOut: MO, Atts > X }
 if I0 == p1(findMin(H))
 /\ entails(B0,B1)
 /\ H0:= insert(((N,getTimeCmd(MTell: MT, MAsk: MA, MIn: MI,
                   MOut: MO, C1, L0) plus alpha(MA, L0))), P) .

crl [delay]:
    { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: (ask B1 -> C1) >
      < I: Simulation | pqueue: H, pend: P, Atts > X }
 => { < L0: agent | store: B0 >
      < L0: process | UID: I0, command: (ask B1 -> C1) >
      < I: Simulation | pqueue: deleteMin(H),
                        pend: insert(((I0,T1)),P),Atts > X }
 if I0 == p1(findMin(H))
 /\ not(entails(B0,B1))
 /\ T1:= (p2(findMin(H))) .
```

The [tell] rule implements the execution semantics of a tell command by posting the given constraint in the local store and by removing such a command from the configuration. The [parallel] rule implements the semantics for parallel composition of process by spawning the two process in the current space. Rule [space] creates a new agent's space denoted by N0.L0, with an empty store (i.e., true), beside the execution of program C0 within the new agent's space. The [extrusion] rule executes C0 in the parent's space L0. The [ask] rule executes a command C1 when the guard B1 in ask B1 -> C1 holds: that is, when B1 is a semantic consequence of the contents B0 of the local store. Note that the semantic consequence relation of the constraint system is queried by asking the SMT solver. The [delay] rule represents the negative answer for the ask rule: whenever B0 does not entail B1 or B1 is not a semantic consequence of the contents B0 of the store. The [delay] rule moves an ask command into the pending heap, where it will remain until the tick rule executes again.

## 4 Reachability Analysis

The goal of this section is to explain how the rewriting logic semantics $\mathcal{R}$ of rtsccp and rewriting modulo SMT can be used as an automatic mechanism for solving existential reachability goals in the initial model $\mathcal{T}_{\mathcal{R}}$. This approach can be useful for symbolically proving or disproving real-time safety properties of $\mathcal{T}_{\mathcal{R}}$. The approach presented in this section mainly relies on Maude's `search` command, but it can be easily extended to be useful in the more general setting of Maude's LTL Model Checker.

The examples presented in this section use the following time functions for the processes:

```
MAsk  : root |-> 1/20, (0 . root) |-> 1/10, (0 . 1 . root) |-> 3/20,
        (1 . root) |-> 1/10, (2 . root) |-> 1/10
MTell : root |-> 1/10, (0 . root) |-> 3/20, (0 . 1 . root) |-> 1/5,
        (1 . root) |-> 3/20, (2 . root) |-> 3/20
MSP   : root |-> 1/2, (0 . root) |-> 7/10, (0 . 1 . root) |-> 4/5,
        (1 . root) |-> 13/20, (2 . root) |-> 3/5
MExt  : root |-> 1/2, (0 . root) |-> 13/20, (0 . 1 . root) |-> 1,
        (1 . root) |-> 1/2, (2 . root) |-> 3/5
```

For example, querying the store at the root agent takes 1/20 time units.

*Fault tolerance* is the property that ensures a system to continue operating properly in the event of the failure; *consistency* means that a local failure does not propagate to the entire system. In $\mathcal{R}$, this means that if a store becomes inconsistent, it is not the case that such an inconsistency spreads to the entire system. Of course, inconsistencies can appear in other stores due to some unrelated reasons.

Searching an inconsistent store can be easily implemented with the help of $\mathcal{R}$ and Maude's `search` command. The answer of this command in the positive would mean that from some initial state, there is a state in which a store becomes inconsistent at some point of execution within a given time interval. Taking advantage of $\mathcal{R}$ and the rewriting modulo SMT approach, also is possible to know when a store is inconsistent. As an example, consider the container system in Figure 1a and the following `search` command with a the time interval $[0, 1.5]$:

```
search in SCCP :
    { < root : agent | store : (W:Integer === (9).Integer) >
      < 0 . root : agent | store : (X:Integer >= 11) >
      < 0 . 1 . root : agent | store : (Y:Integer > 5) >
      < 1 . root: agent | store: true > < 2 . root: agent | store: true >
      < root : process | UID : 1, command : ((((ask X:Integer > 2 ->
        (tell(Y:Integer < 10) in 0 in 1 out 0)) in 0)
        || ((tell(Z:Integer =/== (10).Integer)
        || (tell(T:Integer === 1) in 3)) in 2))
        || (tell(X:Integer <= 10) in 0)) >
      < 1 : Simulation | gtime : 0,pqueue : T(1,((1,0)),empty,empty),
        pend : empty,nextID : 19, flg : false, ... (time maps)}
=>* { < 1 : Simulation | gtime : T:Time, Atts:AttributeSet >
      < A:Aid : agent | store : B0:Boolean > C:Configuration }
    such that check-unsat(B0:Boolean) and T < 3/2 .
```

Note that a store is inconsistent if it is unsatisfiable, thereby checking whether a store is inconsistent is accomplished with the function `check-unsat`. The aforementioned command searches for an inconsistent store during the first 1.5 units of time of the system's execution. This command does not find an inconsistent store between the first 1.5 units of time in any of the 56 reachable states. However, it is possible to make a store inconsistent by adding inconsistent information, for example by adding the process `tell(X <= 10) in 0`. The output for the `search` command is:

```
Solution 1 (state 159)
states: 160  rewrites: 16666 in 876ms cpu (875ms real)
        (19025 rewrites/second)
C:Configuration --> < root:agent | store:(W:Integer === (9).Integer) >
  < 0 . 1 . root : agent | store : (Y:Integer > 5) >
  < 1 . root:agent | store:true > < 2 . root:agent | store:true >
  < 2 . root:process | UID: 28,
                       command: tell(Z:Integer =/== (10).Integer) >
  < 2 . root:process | UID: 27,
                       command: (tell(T:Integer === (1).Integer) in 3) >
  < 0 . root:process | UID : 24,
                       command: (tell(Y:Integer < 10) in 0 in 1 out 0) >
A:Aid --> 0 . root
B0 --> X:Integer >= (11).Integer and X:Integer <= (10).Integer
Atts --> pqueue : T(2,(25,1/20),T(1,(24,4/5),empty,empty),T(1,(28,1/10),
  T(1,(27,3/5),empty,empty),empty)),pend : empty,nextID : 29,
  flg : true, ... (time maps)
T --> 1/2
```

There are 238 reachable states (from the initial state) and 74 of them have an inconsistent store between the first 1.5 units of time. The first inconsistency appears in 0.5 time units, and the last one in 1.3 times units. Note that, the system continues evolving even though there is an inconsistency. It is possible to verify that there are states with consistent and inconsistent stores at the same time by slightly modifying the above search command.

*Knowledge inference* refers to acquiring new knowledge from existing facts. In the setting of $\mathcal{R}$, this means that from a given initial state an agent, at some point, has gained enough information to infer new facts. A positive answer to such a query, means that from some initial state, at some moment during execution, there is at least one agent that has gained enough information to infer the given facts. As an example, consider the container system in Figure 1a and the following `search` command:

```
search in SCCP :
    { < root : agent | store : (W:Integer === (9).Integer) >
      < 0 . root : agent | store : (X:Integer >= 11) >
      < 0 . 1 . root : agent | store : (Y:Integer > 5) >
      < 1 . root : agent | store : true >
      < 2 . root : agent | store : true >
      < root : process | UID : 1, command : (((ask X:Integer > 2 ->
```

```
            (tell(Y:Integer < 10) in 0 in 1 out 0)) in 0)
            || ((tell(Z:Integer =/== (10).Integer)
            || (tell(T:Integer === 1) in 3)) in 2)) >
        < 1 : Simulation | gtime : 0,pqueue : T(1,((1,0)),empty,empty),
          pend : empty, nextID : 19, flg : false, ... (time maps) > }
=>* { < 1 : Simulation | gtime : T:Time, Atts:AttributeSet >
      < A:Aid : agent | store : B0:Boolean > C:Configuration }
    such that entails(B0:Boolean, Y:Integer < 15) and T:Time > 0
               and T:Time < 2 .
```

It checks if there is a state, reachable from the given initial state, in which some store logically implies $Y < 15$ in the time interval $(0, 2)$. This command does not find a container with enough information in such time interval in any of the 56 reachable states. However, if the time interval in the command is changed to $(2, 3)$ the query finds two solutions:

```
Solution 1 (state 54)
states: 55  rewrites: 7466 in 360ms cpu (358ms real)
                    (20738 rewrites/second)
C:Configuration --> < root:agent | store:(W:Integer === (9).Integer) >
  < 0 . root : agent | store : (X:Integer >= 11) >
  < 1 . root : agent | store : true >
  < 2 . root : agent | store : (Z:Integer =/== (10).Integer) >
  < 3 . 2 . root : agent | store : (T:Integer === (1).Integer) >
A:Aid --> 0 . 1 . root
B0:Boolean --> Y:Integer > (5).Integer and Y:Integer < (10).Integer
Atts:AttributeSet --> pqueue : T(1,(29,1/10),empty,empty),pend : empty,
  nextID : 30,flg : true, ... (time maps)
T:Time --> 5/2

Solution 2 (state 55)
states: 56  rewrites: 7601 in 368ms cpu (366ms real)
                    (20654 rewrites/second)
C:Configuration --> < root:agent | store:(W:Integer === (9).Integer) >
  < 0 . root : agent | store : (X:Integer >= 11) >
  < 1 . root : agent | store : true >
  < 2 . root : agent | store : (Z:Integer =/== (10).Integer) >
  < 3 . 2 . root : agent | store : (T:Integer === (1).Integer) >
A:Aid --> 0 . 1 . root
B0:Boolean --> Y:Integer > (5).Integer and Y:Integer < (10).Integer
Atts:AttributeSet --> pqueue : empty,pend : empty,nextID : 30,
  flg : false, ... (time maps)
T:Time --> 13/5
...
```

The webpage at

    http://escher.javerianacali.edu.co/rtsccpe/index.html

contains more details about this example and other examples about reachability analysis with $\mathcal{R}$, including knowledge inference and equivalence of knowledge.

# 5 Related Work and Concluding Remarks

In addition to the related work included in Section 1, it is important to mention other related research in the area of timing semantics for concurrent constraint programming. An extension of concurrent constraint programming in [24] presents a timed asynchronous computation model and propose an implementation using loop-free deterministic finite automata, a declarative framework for reactive systems where time is represented as discrete time units. More recently, J. A. Pérez and C. Rueda [20] propose an operational semantics based on probabilistic automaton, extending the work in [24], with probabilistic and non-deterministic choices for processes. The inclusion of stochastic information for processes proposed by J. Aranda et al. in [1] associates to each computation a random variable determining its time duration: given a set of competing actions, the fastest action is executed, that is, the one with the shortest duration. Finally, G. Sarria and C. Rueda [27] present a real-time extension of ccp with application to music interaction.

In the realm of rewriting logic, Degano et al. [7] provide a rewriting logic semantics for Milner's CCS with interleaving behavior. Additionally, a set of axioms is defined for a logical characterization of the concurrency of CCS processes. In [28], the authors use rewriting logic to represent the semantics of CCS and a modal logic for describing local capabilities of CCS processes. In particular, they study how to make executable the SOS semantics of CSS and present a fully executable specification of the semantics. More recently, M. Romero and C. Rocha [22] have proposed a symbolic rewriting logic semantics of the spatial modality of ccp with extrusion.

This paper has presented a real-time rewriting logic semantics for spatial concurrent constraint programming (rtsccp) that is fully executable in the Maude system. The intended models of rtsccp are spatially-distributed multi-agent reactive systems that may have different computing capabilities and be subject to real-time requirements. In this setting, time attributes are associated to process-store interaction, as well as to process mobility in the space structure, by means of maps from agents to non-negative real quantities. Details about the underlying constraint system have been given as materialized with the help of rewriting modulo SMT. Furthermore, examples of reachability analysis performed on this semantics have been given to illustrate certain aspects of the timing behavior of agents distributed across hierarchical spaces, such as fault-tolerance and consistency.

Future work can span in two directions. One interesting direction to follow is to pursue challenging case studies in which, with the help of the real-time rewriting logic semantics for rtsccp presented in this work, other key aspects of spatially distributed concurrent processes such as privacy can be analyzed. The other direction, is to pursue a more general and fully symbolic rewriting logic semantics for rtsccp where time information can also be modeled as shared variables under the control of the SMT decision procedures. In such a setting, interesting properties of real-time systems such as missed deadlines and deadlocks could be fully analyzed, e.g., for infinitely many initial states in a system.

## References

1. J. Aranda, J. A. Pérez, C. Rueda, and F. D. Valencia. Stochastic Behavior and Explicit Discrete Time in Concurrent Constraint Programming. In M. Garcia de la Banda and E. Pontelli, editors, *Logic Programming*, volume 5366, pages 682–686. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806, pages 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

3. R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, Aug. 2006.

4. D. Chiarugi, M. Falaschi, D. Hermith, R. Marangoni, and C. Olarte. Stochastic modelling of non markovian dynamics in biochemical reactions. In I. Rojas and F. M. O. Guzman, editors, *International Work-Conference on Bioinformatics and Biomedical Engineering, IWBBIO 2013, Granada, Spain, March 18-20, 2013. Proceedings*, pages 537–544. Copicentro Editorial, 2013.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All about Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Number 4350 in Lecture notes in computer science. Springer, Berlin, 2007.

6. F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 161:45–83, 2000.

7. P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. *Theoretical Computer Science*, 275(1-2):259–282, Mar. 2002.

8. B. Dutertre. Yices 2.2. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, A. Biere, and R. Bloem, editors, *Computer Aided Verification*, volume 8559, pages 737–744. Springer International Publishing, Cham, 2014.

9. D. Gilbert and C. Palamidessi. Concurrent constraint programming with process mobility. In *Proc. of the Conference on Computational Logic - CL 2000*, Lecture Notes in Artificial Intelligence, pages 463–477. Springer-Verlag, 2000.

10. V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *Symposium on Principles of Programming Languages*, pages 189–202, 1999.

11. V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2):3–49, 1998.

12. M. Guzmán, S. Haar, S. Perchy, C. Rueda, and F. D. Valencia. Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *Journal of Logical and Algebraic Methods in Programming*, 86(1):107–133, Jan. 2017.

13. S. Knight, C. Palamidessi, P. Panangaden, and F. D. Valencia. Spatial and Epistemic Modalities in Constraint-Based Process Calculi. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Koutny, and I. Ulidowski, editors, *CONCUR 2012 – Concurrency Theory*, volume 7454, pages 317–332. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

14. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, Apr. 1992.

15. J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. *Information and Computation*, 231:38–69, Oct. 2013.

16. M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1):145–188, 2002.

17. C. Okasaki. *Purely Functional Data Structures*. Cambridge Univ. Press, Cambridge, 1. paperback ed., transf. to digital printing edition, 2003. OCLC: 552279078.

18. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, Aug. 2002.

19. P. C. Ölveczky and J. Meseguer. Abstraction and Completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, July 2007.

20. J. A. Pérez and C. Rueda. Non-determinism and Probabilities in Timed Concurrent Constraint Programming. In M. Garcia de la Banda and E. Pontelli, editors, *Logic Programming*, volume 5366, pages 677–681. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

21. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming*, 86(1):269–297, Jan. 2017.

22. M. Romero and C. Rocha. Reachability analysis for spatial concurrent constraint systems with extrusion. `http://camilorocha.info/publications`, 2017.

23. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 4–7 July 1994.

24. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. pages 71–80. IEEE Comput. Soc. Press, 1994.

25. V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 272–285, Jan. 1995.

26. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *POPL'91*, pages 333–352. ACM, 1991.

27. G. Sarria and C. Rueda. Real-time concurrent constraint programming. In *34th Latinamerican Conference on Informatics (CLEI 2008)*, pages 379–391. CLEI, 2008.

28. A. Verdejo and N. Martí-Oliet. Two Case Studies of Semantics Execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27(1-2):113–172, Sept. 2005.