Noname manuscript No.
(will be inserted by the editor)

ccReact: a Rewriting Framework for the Formal Analysis of Reaction Systems

Demis Ballis · Linda Brodo · Moreno Falaschi · Carlos Olarte

Received: date / Accepted: date

Abstract Reaction Systems (RSs) are a computational framework inspired by biochemical systems, where entities produced by reactions can enable or inhibit other reactions. RSs interact with the environment through a sequence of sets of entities called the context. In this work, we introduce **ccReact**, a novel interaction language for implementing and verifying RSs. **ccReact** extends the classical RS model by allowing the specification of recursive, nondeterministic, and conditional context sequences, thus enhancing the interactive capabilities of the models.

We provide a rewriting logic (RL) semantics for **ccReact**, making it executable in the Maude system. We prove that our RL embedding is sound and complete, thereby offering a robust tool for analyzing RSs. Our approach enables various formal analysis techniques for RSs, including simulation of RSs interacting with **ccReact** processes, verification of reachability properties, model checking of temporal (LTL and CTL) formulas, and exploring the system evolution through a graphical tool to better understand its behavior.

We apply our methods to analyze RSs from different domains, including computer science and biological systems. Notably, we examine a complex breast cancer case study, demonstrating that our analysis can suggest improvements to the administration of monoclonal antibody therapeutic treatments in certain scenarios.

Keywords reaction systems · model checking · SOS · rewriting logic · Maude

Research supported by the Next Generation EU project PNRR ECS00000017 "THE - Tuscany Health Ecosystem" (Spoke 3 - CUP B63C22000680007), Next-GenerationEU projects MEDICA (PRIN 2022, CUP_B53D23013170006), and DELICE (PRIN PNRR 2022, P20223T2MF)

D. Ballis
DMIF, University of Udine, Italy
E-mail: demis.ballis@uniud.it

L. Brodo
Dipartimento di Scienze economiche e aziendali, Università di Sassari, Italy
E-mail: brodo@uniss.it.

M. Falaschi
Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche
Università di Siena, Italy E-mail: moreno.falaschi@unisi.it.

C. Olarte
LIPN, CNRS UMR 7030, Université Sorbonne Paris Nord, France
E-mail: olarte@lipn.univ-paris13.fr

1 Introduction

Reaction Systems (RSs) [21] are a formal framework inspired by the functioning of living cells and by biochemistry. RSs have proven to be a versatile computational model with a wide range of applications, including the modeling of biological processes [9, 29, 8, 14], molecular chemistry [42], and computer science systems [21, 31, 32]. A RS comprises a finite set of entities and a finite set of reactions acting on the entities. Each reaction is described by a triple (R, I, P), where R denotes *reactants* (entities that must be present for the reaction to occur), I denotes the *inhibitors* (entities that must be absent to enable the reaction), and P denotes the *products* (entities generated if the reaction takes place).

The behavior of a RS is typically specified as a rewrite system that models *processes interacting* with an external *context.* The external context specifies the entities provided by the environment at each step of the process. The current system state is determined by the union of the entities coming from the environment with those produced in the previous state. A state transition is then determined by applying all and only the enabled reactions in the current state.

The design of RSs for modeling some natural phenomena is often done by domain experts to validate their hypotheses and requires some degree of abstraction. At the design stage, false assumptions or inaccuracies may be easily introduced. Furthermore, the specification of RSs may be an error-prone activity, since system states typically contain a large number of entities which might be deeply interconnected due to the reactions in play. In this scenario, the verification of RSs is a challenging and nontrivial task that demands appropriate formal tools with automated support.

Rewriting Logic (RL) [38] is a very general *logical* and *semantic framework* that is particularly suitable for modeling and analyzing complex nondeterministic and concurrent systems. The specification unit in RL is a *rewrite theory* $\mathcal{R}$, that combines an equational theory $\mathcal{E}$, representing system states as terms of an algebraic datatype, with a set of rewrite rules R, modeling the state transitions of the system. The system evolves by applying the rules in R to the system states by means of rewriting modulo the equational theory $\mathcal{E}$.

Maude [30] is a high-performance language that efficiently implements RL and seamlessly integrates functional, logic, concurrent, and object-oriented computations. Maude provides a large set of analysis methods, including simulation by (strategic) rewriting, explicit-state and symbolic reachability analysis, and model checking temporal formulas [30].

Contributions. This paper presents a new interaction language, called **ccReact**, for specifying RSs. The language is equipped with an *executable* rewriting logic semantics, suitable for both simulation and verification. **ccReact** extends the classic RS computational model by allowing the specification of context sequences that model external stimuli coming from the environment. It exploits process algebraic operators [40, 41, 34] (such as action prefix, sum and recursion) and offers the possibility to express conditional statements. The advantage of having a language for specifying contexts is twofold. First, we can recursively define contexts that possibly exhibit complex nondeterministic behaviors in a precise way, as sometimes have already appeared in the literature [35, 24, 25]; second, we are able to represent a collection of experiments within a single semantic object, so that the consequences of some variation in the context sequence can then be more easily compared and analyzed. Our approach thus overcomes the usual limitation of *in silico* experiments, where a single context sequence is synthesized and one can only observe the outcomes for that specific input without the possibility of analyzing context changes in real time.

The semantics of **ccReact** is formalized within the RL framework, thus making available for it the rich set of Maude analysis tools. Our rewrite theory can be directly used to: (i) simulate biological experiments; (ii) perform reachability analysis; and (iii) model-check CTL and LTL temporal properties.

As we said, RSs have applications in many different areas. We demonstrate that our approach is sufficiently general and it can be useful in multiple application domains, also beyond the pure biological setting. In this regard, we consider some case studies from computer science [21] and a complex biological case study from [33]. In the latter, the protein signaling network for the HER2-positive breast cancer subtype is analyzed in the presence of different combinations of monoclonal antibody drugs. The authors in [33] aim at achieving the best drug treatment for three different breast cancer representative cell lines: BT474, SKBR3 and HCC1954. Our analyses not only corroborate previous results in the literature concerning this system, but also allow us to validate new hypotheses.

We therefore contribute with a novel and expressive language, coupled with several associated automatic verification tools, all grounded on a robust and verifiable framework. Our work supports the design of correct RSs and fosters *in silico* experimentation using this model.

This article builds on a conference paper that was presented in [13]. With respect to [13], the novel contributions are listed below.

1. We have included many more details and explanations. In particular, we have clarified how to derive RSs for our biological case study from a boolean network via the methodology proposed by [15]. We have also enriched **ccReact** with new derived language constructors that simplify the specification of contexts.
2. Full proofs of the main technical results have been added to formally show the soundness and completeness of our framework.
3. We borrow from [36] new verification problems, including RSs for systems in biology and computer science. We show how they can be specified and verified in **ccReact** as well.
4. We show how to use the `ANIMA` [7] tool to graphically and stepwise explore the computation space of a RS specified in **ccReact**. Specifically, we use `ANIMA` to find out a counterexample for a refuted CTL formula. This allows us to better understand the origin of an unexpected behavior.

5. We have greatly expanded the related work section and added hints to future lines of work.

Organization. Section 2 recalls the basics of RSs and introduces the biological case study used as a running example. Section 3 first introduces the syntax for **ccReact** and then its operational semantics. In this section, we also prove that the chosen formal semantics adequately captures the behavior of RS interactive processes (Theorem 1). After a brief overview on RL, Section 4 provides a RL characterization of the operational semantics of **ccReact**. We show in Corollary 1 that the transition systems induced by the operational semantics of **ccReact** and the proposed rewrite theory are bisimilar. Section 5 shows how our analysis methods can be used to validate already known and new hypotheses about the case study. Additionally, we show how to verify some rsCTL [36] (temporal logic for reaction systems) formulas within our framework. Section 6 reviews related work and Section 7 concludes the paper and give directions of future work. The complete Maude specification and case studies are available in the companion tool [12].

2 Reaction Systems

In this section we recall some basic notions about RSs that are relevant to this work. For a full discussion refer to [21]. We also introduce our case study (Section 2.1).

We use the term *entities* to denote generic molecular substances (e.g., atoms, ions, molecules) that may be present in the states of a biochemical system. The main mechanisms that regulate the functioning of a living cell are *facilitation* and *inhibition.* These mechanisms are based on the presence and absence of entities and are reflected in the basic definitions of RSs.

Definition 1 (Reaction Systems) Let S be a (finite) set of entities. A reaction in S is a triple $a = (R, I, P)$, where $R, I, P \subseteq S$ are finite sets and $R \cap I = \emptyset$. We use $rac(S)$ to denote the set of all reactions on the set of entities S. A Reaction System (RS) is a pair $\mathcal{A} = (S, A)$ s.t. S is a finite set of entities, and $A \subseteq rac(S)$ is a finite set of reactions.

The sets R, I, P are the sets of *reactants*, *inhibitors*, and *products*, respectively. Due to biological considerations, the sets R and P are usually nonempty: there is no creation from nothing ($R \neq \emptyset$), and if a reaction takes place then something is produced ($P \neq \emptyset$). A reaction can take place whenever all of its reactants are present in a given state while none of its inhibitors is present (predicate $en_a(\cdot)$ below). If this happens, the reaction is enabled and creates its products (function $res_A(\cdot)$ below). More formally,

Definition 2 (Reaction Result) Given a (finite) set of entities S, and a subset $W \subseteq S$, we define the following:

1. Let $a = (R, I, P) \in rac(S)$ be a reaction in S. The result of reaction a on W, denoted by $res_a(W)$, is defined by:

$$res_a(W) \triangleq \begin{cases} P & \text{if } en_a(W) \\ \emptyset & \text{otherwise} \end{cases}$$

where the *enabling* predicate is defined by

$$en_a(W) \triangleq (R \subseteq W) \wedge (I \cap W = \emptyset)$$

2. Let $A \subseteq rac(S)$ be a finite set of reactions. The result of A on W, denoted by $res_A(W)$, is defined by: $res_A(W) \triangleq \bigcup_{a \in A} res_a(W)$.

Reaction Systems are based on three assumptions:

1. **No permanency**, i.e., any entity vanishes unless it is sustained by a reaction. Therefore, if an entity in a system state is not supported by at least one reaction which produces it, then it will disappear in the following state.
2. **No counting**, the basic model of RSs is very abstract and qualitative, i.e. the quantity of entities that are present in a cell is not considered.
3. **Threshold nature of resources**, it is assumed that either an entity is available for all reactions, or it is not available at all. In other terms, two or more reactions that are enabled by a set of entities W always generate their products, even if they share some entities, because there is always a sufficient amount of reactants in W to activate all the enabled reactions.

The previous assumptions imply that nonempty intersection of sets of reactants of enabled reactions never generates conflicts, which is a major difference with other models for concurrency (e.g., Petri nets). For instance, given the set of reactions $A = \{a_1, a_2\}$, where $a_1 = (\{\mathsf{a}, \mathsf{b}\}, \{\mathsf{c}\}, \{\mathsf{b}\})$ and $a_2 = (\{\mathsf{a}\}, \{\mathsf{d}\}, \{\mathsf{c}\})\}$, both a_1 and a_2 are enabled on the set of entities $D = \{\mathsf{a}, \mathsf{b}\}$. Moreover, both take place simultaneously, producing $\{\mathsf{b}\}$ and $\{\mathsf{c}\}$.

Since living cells are seen as open systems that interact with the external environment, the behavior of a RS is formalized through the notion of *interactive process*, that is, a process that react to external stimuli.

Definition 3 (Interactive Process) Let $\mathcal{A} = (S, A)$ be a RS and let $n \geq 0$. An n-step *interactive process* in $\mathcal{A}$ is a pair $\pi = (\gamma, \delta)$ s.t. $\gamma = \{C_i\}_{i \in [0,n]}$ is the *context sequence* and $\delta = \{D_i\}_{i \in [0,n]}$ is the *result sequence*, where $C_i, D_i \subseteq S$ for any $i \in [0, n]$, $D_0 = \emptyset$, and $D_{i+1} = res_A(D_i \cup C_i)$ for any $i \in [0, n-1]$. We call $\tau = W_0, \ldots, W_n$ with $W_i \triangleq C_i \cup D_i$, for any $i \in [0, n]$ the *state sequence*.

The context sequence γ represents the environment interacting with the RS. It specifies the *input* that comes from the environment. The result sequence δ is entirely determined by γ and the reactions in A. As expected, distinct context sequences may lead to distinct outcomes for the same reaction set A, but a context sequence γ determines a *unique* result sequence δ.

2.1 Case Study

We consider the case study in [33] concerning the protein signaling networks in breast cancer. Breast cancer can be classified into subtypes differing in cellular properties and diagnosis. The HER2-positive subtype is characterized by an over expression of the human epidermal growth factor (EGF) receptor 2 (HER2, also named ErbB-2). We will focus on the activity of ErbB-2 which is part of the human (EGF) receptor family also composed by ErbB-1, ErbB-3, and ErbB-4. All these four receptors are receptor tyrosine kinases (RTKs), i.e. they are essential components for signal transduction pathways mediating cell-to-cell communication. The ErbB-i receptors form different homo- or hetero-dimers when activated by growth factors, for example EGF for ErbB-1 or heregulin, HRG, for ErbB-3, enabling potential signaling pathways that can contribute to the breast cancer tumor genesis and tumor progression. In particular, ErbB-2 is an orphan receptor, i.e. it does not bind EGF-like ligands by itself, while it can be activated when complexed with the receptor ErbB-3; moreover when it is orphan, it is overexpressed in malignant cells in 10-20 percent of breast tumours, diagnosed as HER2-positive.

The therapeutic treatment of the HER2-positive cancers is typically based on the monoclonal antibody drugs erlotinib, trastuzumab and pertuzumab, which are especially designed to target ErbB-2 behavior. Unfortunately, the therapy is often weakened by drug resistance that can be caused by pathway deregulatory activity or by the (over expressions of) ErbB-family receptors that allow to bypass the drug effects. In particular, ErbB-1 expression has been recognized to be able to inhibit the drug effects on ErbB-2.

The experimental analyses in [33] focus on the treatment where different drug combinations are tested to improve drug efficiency. More precisely, the authors in [33] consider:

– three different kinds of HER2-positive subtype breast cancer, identified by the cell lines BT474, HCC1954, and SKBR3;
– three types of drugs: erlotinib (e), pertuzumab (p), trastuzumab (t), provided alone or in combination;
– two types of input stimuli: EGF and HRG;
– two different treatment periods: short- and long-term drug treatments.

Input stimuli enable the activation of the EGF and HRG receptors and start the signaling pathways to be analyzed. Experiments in [33] aim at analyzing specific molecules appearing in *attractors*. Attractors represent a stable behavior of the system; they can be composed by one or more states composing a loop toward which the system tends to evolve over time. The interesting molecules to be checked in attractors are key entities in the tumor genesis and tumor progression. The search for such molecules is carried out with respect to a persistent stimulus that includes both EGF and HRG and different drug combinations. Both short- and long-term experiments have been conducted by checking two distinct classes of molecules in the attractors: for short-term analysis AKT, ERK1/2, and p70S6K are checked, and for long-term analysis RPS6 and RB are checked.

Table 1 summarizes the results of [33] for the six considered systems: one per kind of cancer cell line, short- and long-term versions. In the first column the drug combinations are reported: x stands for no drug treatment. Each row of the table presents the efficacy of a drug combination under the same external stimulus for the three kinds of cancer. The treatment is effective when the molecules of interest (AKT, ERK1/2, p70S6K, for short-term, and RPS6, RB for long-term) are not present in attractor states, i.e. the cell value in the Table is equal to 0.

In [33] a qualitative model, using boolean networks, based on a biological dataset, is used to describe and simulate the systems under consideration. To generate our RS models, we follow the approach adopted in [15]: each boolean formula is translated into a set of reactions considering the following rules:

– all the elements appearing in the formula in their positive form become reactants;
– all the elements appearing in the formula in their negative form become inhibitors;
– all the reactants connected with the logical operator AND are put together in the same reaction;
– the reactants connected with the logical operator OR are encoded in different reactions;
– all the inhibitors connected with the logical operator AND are encoded in the same reaction;
– all the inhibitors connected with the logical operator OR are put in different reactions.

	Short-term									Long-term					
	BT474			HCC1954			SKBR3			BT474		HCC1954		SKBR3	
	AKT	ERK1/2	p70S6K	AKT	ERK1/2	p70S6K	AKT	ERK1/2	p70S6K	RPS6	RB	RPS6	RB	RPS6	RB
x	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0
e	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0
p	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
t	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0
e,p	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0
e,t	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0
p,t	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
e,p,t	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0

Table 1 Short-term and long-term experiments from [33].

As an example, let us consider the boolean rule for the receptor ErbB-1:

$$\mathsf{ErbB\text{-}1} = (\mathsf{ErbB\text{-}1} \lor \mathsf{EGF} \lor \mathsf{PLC}) \land \neg(\mathsf{E} \lor \mathsf{P}),$$

as in the Additional file 5 of [33]. Here the receptor ErbB-1 is activated when ErbB-1 is already active or when EGF or PLC are present in the actual state of the system, and the receptor ErbB-1 is not active when one of the drugs erlotinib (E) or pertuzumab (P) is present. The reactions generated by this boolean rule are

$$\begin{array}{l} a_1 : (\mathsf{ErbB\text{-}1};\ \mathsf{E},\mathsf{P}\ ;\mathsf{ErbB\text{-}1}) \\ a_2 : (\mathsf{EGF};\quad \mathsf{E},\mathsf{P}\ ;\mathsf{ErbB\text{-}1}) \\ a_3 : (\mathsf{PLC};\quad \mathsf{E},\mathsf{P}\ ;\mathsf{ErbB\text{-}1}) \end{array}$$

where there are two inhibitors and the presence of one of them inhibits all the three reactions; the receptor ErbB-1 is activated in three different situations: when it is already active (reaction a_1), when EGF is present (a_2), and when PLC is present (a_3).
The resulting RSs we obtain for the six systems in Table 1, from left to right, include 39, 39, 34, 47, 50 and 55 reaction rules respectively and they can be found in the companion tool [12].

Below, we give a simple example of an interactive process using some entities and three reactions resulting from the translation of the boolean rules in [33] to illustrate the dynamics of the resulting RS. Please note that the names of entities are all written in lowercase characters and without the dash symbol (-) as in the tool syntax.

Example 1 Let $\mathcal{A} = (S, A)$ be the RS where

$$\begin{array}{ll} S &= \{\mathsf{egf},\mathsf{e},\mathsf{p},\mathsf{erbb1},\mathsf{erk12},\mathsf{p70s6k}\} \\ A &= \{a_1, a_2, a_3\} \\ a_1 &= (\{\mathsf{egf}\}, \{\mathsf{e},\mathsf{p}\}, \{\mathsf{erbb1}\}) \\ a_2 &= (\{\mathsf{egf}\}, \emptyset, \{\mathsf{erk12}\}) \\ a_3 &= (\{\mathsf{erk12}\}, \emptyset, \{\mathsf{p70s6k}\}) \end{array}$$

Consider the context sequence $\gamma = C_0, \cdots, C_3$, with $C_0 = \{\mathsf{egf},\mathsf{e}\}$, $C_1 = \emptyset$, $C_2 = \{\mathsf{egf}\}$, and $C_3 = \emptyset$. Then, $\pi = (\gamma, \delta)$ in an interactive process where $\delta = D_0, \cdots, D_3$, with $D_0 = \emptyset$, $D_1 = \{\mathsf{erk12}\}$, $D_2 = \{\mathsf{p70s6k}\}$, and $D_3 = \{\mathsf{erbb1},\mathsf{erk12}\}$. The resulting state sequence is $\tau = W_0, \cdots, W_3$, where $W_0 = \{\mathsf{egf},\mathsf{e}\}$, $W_1 = \{\mathsf{erk12}\}$, $W_2 = \{\mathsf{egf},\mathsf{p70s6k}\}$, and $W_3 = \{\mathsf{erbb1},\mathsf{erk12}\}$.

3 ccReact: A Language for Executing and Analyzing RSs

Process algebras such as CCS [40], the π-calculus [41], CCP [43] and CSP [34], among several others, are mathematical formalisms to model and reason about concurrent systems. They provide a simple language (few well-defined constructors) to represent *processes* that *interact*, together with an *operational semantics* to represent computational steps. This section introduces the syntax and the operational semantics for **ccReact**, a language of processes that allows for the definition of external contexts interacting with a RS. The purpose of **ccReact** processes is to provide stimuli to the RS in the form of a context C, i.e., a subset of the set of entities *controlled* by the environment. Our language includes constructors to model the situation in *in silico* experiments where different alternative inputs need to be tested; additionally, some of these alternatives are *guarded*, i.e., they depend on the current state of the system. **ccReact** thus opens up the possibility of designing new experiments as we will show in Section 5.

From now on, we set $\mathcal{A} = (S, A)$ to be an arbitrary RS. The following definition formalizes the syntax of **ccReact**.

Definition 4 (ccReact Processes) The language of processes is defined as:

$$\begin{array}{lll} c & ::= e \ \mid\ \overline{e} \ \mid\ c \land c \ \mid\ c \lor c & \text{(Conditions)} \\ p & ::= C \ \mid\ c \to C & \text{(Prefixes)} \\ K, M, N & ::= \mathbf{0} \ \mid\ X \ \mid\ \sum_{i \in I} p_i.K_i \ \mid & \text{(Processes)} \\ & \quad\ K \parallel M \ \mid\ \mathbf{rec}\ X.K & \end{array}$$

where $e \in S$ is an entity, X is a process variable, I is a finite nonempty set of indices, and $C \subseteq S$ is a possibly empty set of entities. The set of processes is denoted by **Proc** and the set of prefixes by **Pref**. Processes are quotiented by a structural congruence relation $\equiv$ satisfying: (1) $K \equiv M$ if they differ only by a renaming of local variables in recursive definitions (alpha-conversion); (2) $K \parallel M \equiv M \parallel K$; (3) $K \parallel (M \parallel N) \equiv (K \parallel M) \parallel N$; and (4) $K \parallel \mathbf{0} \equiv \mathbf{0} \parallel K \equiv K$.

ccReact *processes* are built from *prefixes* that, in turn, are built from *conditions*. Below we give an intuitive description of the constructors of the language to later introduce their formal semantics.

A process $p.K$, where p is a prefix and K is a process, represents the situation where the current stimulus to the RS is determined by p. Then, in the next interaction, the stimulus will be determined by K. A prefix can be simply a (possibly empty) set of entities C or a *guarded prefix* of the form $c \to C$, where c is a *condition*, read as "*when c produce C*".

Let $D \subseteq S$ be a set of entities. The *prefix* C is always *enabled* in D and it *produces* the set of entities C. The prefix $c \;\to\; C$ is *enabled* if condition c holds in D. This happens when the function $check(c, D)$ evaluates to true, where $check(c, D)$ is defined as:

- $check(e, D) \triangleq e \in D$,
- $check(\overline{e}, D) \triangleq e \notin D$, and
- $check(c \bullet c', D) \triangleq check(c) \bullet check(c')$ with $\bullet \in \{\wedge, \vee\}$.

For instance, the prefix $e_1 \wedge e_2 \wedge \overline{e_3} \wedge \overline{e_4} \;\to\; d$ is enabled in D if $\{e_1, e_2\} \subseteq D$ and $\{e_3, e_4\} \cap D = \emptyset$ (i.e., e_1, e_2 are present and e_3, e_4 absent). In that case, the entity d is produced.

The process $\mathbf{0}$ represents the end of a context sequence. The process X represents the *call* to a recursively defined process. The process $\sum_{i \in I} p_i.K_i$ chooses, non-deterministically, one of the enabled prefixes p_i and precludes the others from execution. To simplify the notation: when the set of indices I is a singleton, we omit the "$\sum_{i \in I}$" part and we write $p.K$; if $|I| = 2$, we write $p_1.K_1 + p_2.K_2$.

The parallel composition of two processes is represented by $K \parallel M$, where the outputs of K and M are merged together. Due to the structural congruence, the parallel operator is associative and commutative, with $\mathbf{0}$ as identity (i.e., $(\mathbf{Proc}, \parallel, \mathbf{0})$ is a commutative monoid).

The process $\mathbf{rec}\ X.K$ allows for recursive definitions. We assume that all occurrences of X in $\mathbf{rec}\ X.K$ appear in the scope of a prefix (e.g., $\mathbf{rec}\ X.p.X$ is guarded while $\mathbf{rec}\ X.X$ is not). Moreover, all the occurrences of X in a process K must be bound by the binder "$\mathbf{rec}\ X.$" (i.e., there are no free variables in a well-formed process). These conditions are usually required in process algebras to guarantee, e.g., that the resulting transition system is finitely branching. Note also that, due to the structural congruence (alpha conversion), the process variables can be always renamed, e.g., $\mathbf{rec}\ X.p.X \equiv \mathbf{rec}\ Y.p.Y$.

To avoid parentheses, the convention is that $\wedge$ binds tighter than $\vee$, and, in decreasing order of precedence, we have recursion, prefix, sums and parallel composition. For instance, $\mathbf{rec}\ X.K \parallel p_1.K_1 + p_2.K_2$ must be understood as $(\mathbf{rec}\ X.K) \parallel (p_1.K_1 + p_2.K_2)$.

Below we illustrate the above constructors and define some useful derived operators.

Example 2 (Derived Constructors)

In some experiments, we are interested in providing the same context to the RS at each computation step. We then define the process

$$!C \triangleq \mathbf{rec}\ X.C.X$$

that produces C now, C in the next time instant and so on. More generally, given a finite nonempty set of prefixes $P = \{p_1, \cdots, p_n\}$, we define

$$!P \triangleq \mathbf{rec}\ X.(p_1.X + \cdots p_n.X)$$

This means that the process $!P$ provides as context stimulus, in every time-unit, one of the enabled prefixes $p_i \in P$ (if any).

Given a set $D \subseteq S$ of entities controlled by the environment, a common pattern in experiments is to provide as stimulus any subset of D, i.e., the context will be a set of entities $C \in \mathbb{P}(D)$ (where $\mathbb{P}(D)$ denotes the set of subsets of D). This behavior can be captured by the process $!\{C \mid C \in \mathbb{P}(D)\}$. Alternatively, we can express the same behavior as follows. Let $D = \{d_1, \cdots, d_n\}$. Then we define the process:

$$K_{\mathbb{P}}(D) \triangleq \mathbf{rec}\ X.((d_1.X + \emptyset.X \parallel \cdots \parallel d_n.X + \emptyset.X))$$

The process $K_{\mathbb{P}}(D)$, in each interaction, chooses between producing d_i or nothing ($\emptyset$) for every $d_i \in D$. Then, the parallel composition will collect the entities produced by each alternative, thus leading to a subset of D. Similarly, given a set of set of entities $\mathfrak{D}$, we define the process

$$K_{\mathbb{P}}(\mathfrak{D}) \triangleq \mathbf{rec}\ X. \sum_{D \in \mathfrak{D}} D.X$$

that provides, in each interaction, one of the context D in $\mathfrak{D}$.

Consider now the situation where the stimulus must include any subset of a set of entities D *until* a certain entity c is produced. This context can be specified by the process below:

$$\begin{array}{l}\mathbf{until}\ c\ \mathbf{produce}\ P \triangleq \\ \quad \mathbf{rec}\ X.(c \;\to\; \emptyset.\mathbf{0} \;+ \sum_{C \in \mathbb{P}(D)} (\overline{c} \;\to\; C).X)\end{array}$$

Finally, the process below specifies the usual if-then-else constructor in programming languages:

$\textbf{if } c \textbf{ then } C_1.K_1 \textbf{ else } C_2.K_2 \triangleq$
$(c \ \to \ C_1.K_1) + (\neg c \ \to \ C_2.K_2)$

where $\neg e = \overline{e}$, $\neg \overline{e} = e$, $\neg(c_1 \wedge c_2) = \neg c_1 \vee \neg c_2$, and $\neg(c_1 \vee c_2) = \neg c_1 \wedge \neg c_2$, for any entity e and conditions c_1 and c_2.

3.1 Operational Semantics.

Now we define the input/output behavior of **ccReact** processes when interacting with a reaction system. For that, we consider three reduction relations:

1. $p \overset{D}{\rightsquigarrow} C$, determining that the prefix p produces the output C when the current state is D;
2. $K \xrightarrow{(D,C)} K'$, determining that the context specified by the process K reduces to K' on input D. On doing that, it produces the context C; and
3. $(D, K) \xRightarrow{(C)} (D', K')$, capturing the interaction of K with the RS where the process K outputs C on D. After that, the new *configuration* is (D', K'), where the state of the RS is D' and the context is provided by K'.

The next definition formalizes the relations (1) and (2). Later, Definition 6 formalizes the relation (3).

Definition 5 (Semantics of processes) The relations $\rightsquigarrow$: $\mathbf{Pref} \times S \times S$ and $\longrightarrow$: $\mathbf{Proc} \times S \times S \times \mathbf{Proc}$ are the least relations satisfying, respectively, the rules prefix and prefix-c, and the rules sum, sum$_\emptyset$, parallel and rec, in Figure 1.

The rule prefix produces the output C on input D when the prefix p is C, whereas prefix-c implements conditional prefixes, that is, it produces C if p is of the form $c \ \to \ C$ and c holds in D (i.e., $\mathit{check}(c, D) = \mathit{true}$).

The rule sum selects for execution one of the enabled prefixes and rules out the others. If none of the prefixes is enabled, then the empty set of entities is produced (rule sum$_\emptyset$) and the interaction ends (process $\mathbf{0}$).

In the parallel composition $K \parallel M$, the rule parallel combines the outputs C_K and C_M produced, respectively, by the processes K and M.

The rule rec unfolds the recursive definition. Finally, note that the process $\mathbf{0}$ does not exhibit any transition (no rule for it), since it simply terminates a given process and halts the associated production of contexts.

The interaction of a process K with a reaction system $\mathcal{A}$ is a possibly infinite sequence of contexts produced by K together with the output (i.e., the result set) generated by $\mathcal{A}$ (under the contexts produced by K). More precisely,

$$\frac{\mathit{check}(c, D) = \mathit{true}}{(c \ \to \ C) \overset{D}{\rightsquigarrow} C}\ \textsf{prefix-c} \qquad \frac{}{C \overset{D}{\rightsquigarrow} C}\ \textsf{prefix}$$

$$\frac{p_j \overset{D}{\rightsquigarrow} C_j \quad \text{for some } j \in I}{\sum_I p_i.K_i \xrightarrow{(D,C_j)} K_j}\ \textsf{sum}$$

$$\frac{p_j \overset{D}{\not\rightsquigarrow} \quad \text{for all } j \in I}{\sum_I p_i.K_i \xrightarrow{(D,\emptyset)} \mathbf{0}}\ \textsf{sum}_\emptyset$$

$$\frac{K \xrightarrow{(D,C_K)} K' \text{ and } M \xrightarrow{(D,C_M)} M'}{K \parallel M \xrightarrow{(D,C_K \cup C_M)} K' \parallel M'}\ \textsf{parallel}$$

$$\frac{K[\textbf{rec } X.K/X] \xrightarrow{(D,C)} K'}{\textbf{rec } X.K \xrightarrow{(D,C)} K'}\ \textsf{rec}$$

$$\frac{K \xrightarrow{(D,C)} K' \text{ and } D' = \mathit{res}_{\mathcal{A}}(D \cup C)}{(D, K) \xRightarrow{(C)} (D', K')}\ \textsf{output}$$

Fig. 1 Input/output behavior of **ccReact** processes

Definition 6 (Interaction) Let $\mathcal{A} = (S, A)$ be a RS. $\Longrightarrow\ : S \times \mathbf{Proc} \times S \times S \times \mathbf{Proc}$ is the least relation satisfying the rule output in Figure 1. A run ρ of $K = K_0$ on $\mathcal{A}$ is a sequence of the form

$$(D_0, K_0) \xRightarrow{(C_0)} (D_1, K_1) \xRightarrow{(C_1)} \cdots$$

where $D_0 = \emptyset$. The context sequence associated to ρ, notation $\gamma(\rho)$, is the sequence $\{C_i\}_{i \geq 0}$; the result sequence $\delta(\rho)$ is the sequence $\{D_i\}_{i \geq 0}$; and the state sequence associated to ρ is the sequence $\{C_i \cup D_i\}_{i \geq 0}$.

Note that the interaction $(D, K) \xRightarrow{(C)} (D', K')$ is only possible if K is able to produce a context C on input D, i.e., $K \xrightarrow{(D,C)} K'$ (see premise of rule output). The output D' is computed as in Definition 3 ($D' = \mathit{res}_{\mathcal{A}}(D \cup C)$). In turn, K can produce an output if K is not the process $\mathbf{0}$.

Some **ccReact** processes may engage in an infinite sequence of interactions with a RS (take for instance the process $\textbf{rec } X.C.X$); while some other processes produce a finite sequence of interactions (e.g., $C.\mathbf{0}$). Given an infinite sequence s, we use $s \downarrow n$ to denote the first $n + 1$ elements of s. Given a context sequence $\gamma = \{C_i\}_{i \in [0,n]}$, we use $P(\gamma)$ to denote the **ccReact** process $C_0. \cdots .C_n.\mathbf{0}$. Note that, given an arbitrary RS $\mathcal{A}$, there exists a unique run of length n of $P(\gamma)$ on $\mathcal{A}$.

The following result shows that n-step interactive processes (Definition 3) coincide with the associated state sequences of **ccReact** processes (Definition 6).

This follows directly from the definition of the rule output (where outputs are computed from input contexts exactly as in Definition 3).

Theorem 1 *Let $\mathcal{A} = (S, A)$ be a RS.*

1. *Let $\gamma = \{C_i\}_{i\in[0,n]}$ be a context sequence, and ρ the unique run of $P(\gamma)$ on $\mathcal{A}$. Then, $\pi = (\gamma(\rho), \delta(\rho))$ is an n-step interactive process in $\mathcal{A}$; and*
2. *let K be a process and ρ a run of K on $\mathcal{A}$. If ρ is finite with length n, then $\pi = (\gamma(\rho), \delta(\rho))$ is an n-step interactive process in $\mathcal{A}$. If ρ is infinite, then for all $n \geq 0$, $\pi = (\gamma(\rho) \downarrow n\ ,\ \delta(\rho) \downarrow n)$ is an n-step interactive process in $\mathcal{A}$.*

Proof **(1.)** Consider a process of the form $K = C.K'$. Using the rules in Figure 1, we have $(D, K) \xRightarrow{(C)} (D', K')$ where $D' = res_{\mathcal{A}}(D \cup C)$. Hence, the unique run ρ on the process $P(\gamma)$ is:

$$(D_0, C_0.\cdots.C_n.\mathbf{0}) \xRightarrow{(C_0)} (D_1, C_1.\cdots C_n.\mathbf{0}) \xRightarrow{(C_1)} \cdots (D_n, C_n.\mathbf{0}) \xRightarrow{(C_n)} (D_{n+1}, \mathbf{0})$$

where $D_0 = \emptyset$ and $D_{i+1} = res_{\mathcal{A}}(D_i \cup C_i)$ for $i \in [0, n]$. Clearly, the context sequence γ is equal to the context sequence associated to the run ρ, i.e., $\gamma = \gamma(\rho)$. By induction on the length n of the context sequence γ, and using the previous facts, we conclude that $\pi = (\gamma(\rho), \delta(\rho))$ is an n-step interactive process.
(2.) Consider an infinite run ρ of the process $K = K_0$:

$$(D_0, K_0) \xRightarrow{(C_0)} (D_1, K_1) \cdots \xRightarrow{(C_n)} (D_{n+1}, K_{n+1}) \cdots$$

where $D_0 = \emptyset$. By rule output, we know that $D_{i+1} = res_{\mathcal{A}}(D_i \cup C_i)$ for $i \geq 0$. Hence, for any (finite) ρ's prefix of length n, $\pi = (\gamma(\rho) \downarrow n\ ,\ \delta(\rho) \downarrow n)$ is necessarily a n-step interactive process in $\mathcal{A}$. The case of a finite run ρ of K follows similarly.

4 Rewriting Logic Semantics for ccReact

This section introduces a rewriting logic (RL) [37, 38] semantics for **ccReact**. The syntax and operational semantics defined in Section 3 are formalized as a *rewrite theory* $\mathcal{R}_{RS}$. In this theory, the states of the system are represented as terms in an algebraic datatype (Section 4.2), and the system's transitions (i.e., the operational semantics) are captured by rewrite rules (Sections 4.3 and 4.4). We adopt the Maude notation to define $\mathcal{R}_{RS}$. Maude [28, 30] is a high-level language and tool supporting the specification and analysis of rewrite theories. This approach enables us to produce an *executable specification* and conduct various analyses, as illustrated in Section 5. These analyses include: generating traces/runs (Definition 6) by rewriting; finding whether a state is reachable or not by explicit-state search commands; and determining if some/all the paths of a system satisfy a given property by model checking. For readability, we have omitted some details of the theory $\mathcal{R}_{RS}$; the complete specification is available in the companion tool of this paper [12].

We start with a quick overview of RL and Maude (see [38, 30] for further details).

4.1 Overview of Rewriting Logic and Maude

A *rewrite theory* [37] is a tuple $\mathcal{R} = (\Sigma, E, L, R)$ such that: $\mathcal{E} = (\Sigma, E)$ is an equational theory where Σ is a signature that declares sorts, subsorts, and function symbols; E is a set of (conditional) equations of the form $t = t'$ **if** ψ, where t and t' are terms of the same sort, and ψ is a conjunction of equations; L is a set of *labels*; and R is a set of labeled (conditional) rewrite rules of the form $l : q \longrightarrow r$ **if** ψ, where $l \in L$ is a label, q and r are terms of the same sort, and ψ is a conjunction of equations and *rewrite expressions* of the form $t \Rightarrow t'$ (can t be rewritten into t'?). This latter expression is interpreted as a rewriting-based reachability goal in $\mathcal{R}$. Intuitively, terms in the equational theory $\mathcal{E}$ define system states, while the behavior of the system is given by local transitions between states which are described by rewrite rules.

$T_{\Sigma,s}$ denotes the set of ground terms (i.e., terms not including variables) of sort s, and $T_\Sigma(X)_s$ denotes the set of terms of sort s over a set of sorted variables X. $T_\Sigma(X)$ and T_Σ denote all terms and ground terms, respectively. A substitution $\sigma : X \rightarrow T_\Sigma(X)$ maps each variable to a term of the same sort, and $t\sigma$ denotes the term obtained by simultaneously replacing each variable x in a term t with $\sigma(x)$.

A *one-step rewrite* $t \longrightarrow_{\mathcal{R}} t'$ holds if there is a rule $l : q \longrightarrow r$ **if** ψ, a subterm u of t, and a substitution σ such that $u = q\sigma$ (modulo equations), t' is the term obtained from t by replacing u with $r\sigma$, and $v\sigma = v'\sigma$ holds for each $v = v'$ in ψ. The reflexive-transitive closure of $\longrightarrow_{\mathcal{R}}$ is denoted as $\longrightarrow^*_{\mathcal{R}}$.

A Maude module (`mod` M `is` ... `endm`) specifies a rewrite theory $\mathcal{R}$. Sorts and subsort relations are declared by the keywords `sort` and `subsort`; function symbols, or *operators*, are introduced with the `op` keyword: `op` f `:` s_1 ... s_n `->` s, where $s_1, \ldots, s_n$ are the sorts of the arguments of operator f, and s is its (value) sort. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions (e.g., `_+_`). Some operators can have equational attributes, such as `assoc`, `comm`, and `id:` t, stating that the operator is, respectively, associative, commutative, and/or has identity element t. Conditional equations are specified by the syntax `ceq` t `=` t' `if` ψ. When condition ψ is empty, we simply write `eq` t `=` t'. Similarly, conditional rewrite rules are specified as `crl [`l`] :` u `=>` v `if` ψ, and their notation is simplified into `rl [`l`] :` u `=>` v for the unconditional case. The mathematical variables in such statements are declared with the notations `var` and `vars` or, on the fly, with the notation `X:S` (variable `X` of sort `S`). Comments in the specification can be added with the syntax `--- Comment`.

Maude provides a large set of analysis methods, including simulation by rewriting (command `rew` t, that rewrites the term t) and explicit-state reachability analysis (`search` t `=>*` t' `such that` ψ, searching for a state reachable from t that matches the pattern t' and satisfies the condition ψ).

The Unified Maude Model-Checking tool [44] (called `umaudemc`) allows for the use of different model checkers to analyze Maude specifications. Besides being an interface for the standard LTL model checker of Maude, it also offers the possibility of interfacing external LTL, CTL and probabilistic model checkers.

4.2 Reaction Systems as Maude's terms.

This section introduces the equational theory associated to the theory $\mathcal{R}_{RS}$ that allows us to represent any reaction system and **ccReact** process as a term. We therefore introduce sorts and operators to build entities, reactions, processes, *etc.*

Entities in a RS are constants of `sort Entity`. For instance, the operators below define some of the entities in Example 1:

```
ops egf e p erbb1 : -> Entity .
```

The sort `SetEntity` represents "`,`"-separated sets of entities:

```
sort SetEntity .
subsort Entity < SetEntity . --- A singleton
op empty : -> SetEntity .
op _,_ : SetEntity SetEntity
  -> SetEntity [assoc comm id: empty] .

var E : Entity .
eq E , E = E  . --- Idempotency
```

Note that the operator "`_,_`", denoting set union, is an `assoc`iative and `comm`utative operator with `empty` as `id`entity, and it satisfies the equation for idempotency ($\forall E : \mathit{Entity},\ \ E\texttt{,}E = E$). Moreover, due to the subsort relation, every entity E is also a set of entities (i.e., a singleton containing only E). For instance, "`e , p`" is a term of sort `SetEntity` and it is equivalent (modulo equations) to the term "`empty , p, e , p`".

Sets of reactions are specified as "`,`"-separated sets of terms of `sort Reaction`, built with the following constructor:

```
op [_;_;_] : SetEntity SetEntity SetEntity
         -> Reaction
```

For instance, the reaction a_1 in Example 1 can be represented with the term `[egf ; (e , p) ; erbb1]` of sort `Reaction`.

Below we introduce the needed sorts and operators to specify **ccReact** processes.

```
sorts Condition Process PreProcess SumProcess .
subsort Qid < Process . --- For processes variables
subsort PreProcess < SumProcess < Process .

subsort Entity < Condition .  --- Entities as conds.
op _^ : Entity -> Condition . --- Absent entity
ops _and_ _or_ : Condition Condition -> Condition .

op {_}._  : SetEntity Process -> PreProcess .
op `[ _-->{_} `] . _  : Condition SetEntity Process
                    -> PreProcess .
op 0       :                    -> Process .
op _+_    : PreProcess SumProcess -> SumProcess .
op _||_    : Process Process
         -> Process [assoc comm id: 0] .
op rec_._ : Qid Process          -> Process .
```

The sort `Qid` in Maude denotes quoted identifiers as in `'X`. They are used here to denote process variables. Terms of sort `PreProcess` are *prefixed* processes of the form $p.K$ where $p \in \mathbf{Pref}$ and $K \in \mathbf{Proc}$. Summations $\sum_{i\in I} p_i.K_i$ are represented as terms of the sort `SumProcess`. For instance, the process

$$\mathbf{rec}\ X.\,(\mathsf{ERBB1}.X + (\mathsf{e} \wedge \overline{\mathsf{p}}\ \ \rightarrow\ \ \mathsf{PLCG}).X)$$

that either provides as input $\mathsf{ERBB1}$ or PLCG (when e is present and p is absent) is represented by the following term:

```
rec 'X . (erbb1 . 'X   +
          [ e and p ^  --> { plcg } ] . 'X)
```

4.3 **ccReact** Semantics in Maude

The behavior of a **ccReact** process interacting with a RS is specified by: (1) a *deterministic* part, using equations, for checking *conditions* and computing the effects on the RS when a set of entities C is provided as input context; and (2) rewrite rules defining the non-deterministic behavior of **ccReact** processes.

The following specification defines the operator `eval`, that recursively computes (by using the auxiliary operator `evalRec`) the output D (a set of entities) from a set of reactions SetR and an input set of entities C.

```
vars C D R I P IN OUT : SetEntity .
vars SetR : SetReaction .

op eval    : SetEntity SetReaction -> SetEntity .
op evalRec : SetEntity SetReaction SetEntity
         -> SetEntity .
eq eval(C, SetR) = eval(C,SetR,empty) .
eq evalRec(C,empty,D) = D .
eq evalRec(C,([R ; I ; P],SetR),D) =
   evalRec(C,SetR,(D,output(C,[R ; I ; P]))) .

--- Returns P if [R ; I ; P] is enabled on C
op output : SetEntity Reaction -> SetEntity .
eq output(C,[R ; I ; P]) =
 if (R subset C) and-then intersection(C,I) == empty
   then P else empty
 fi
```

As expected, the definition of `evalRec` checks every reaction against the current input C. A reaction (R, I, P) adds the products P to the output D, when the reactants in R are a `subset` of the input C and $C \cap I = \emptyset$ (definition of `output`). In this specification, `if_then_else_fi` is the usual ternary operator. Note that the operator `output` specifies the function $res_a(\cdot)$ in Definition 2, and `eval`(W, A) computes $res_A(W)$ for a set of entities W and a set of reactions A.

We also define the following operator that implements the intended meaning of the function *check* of Section 3:

```
vars COND1 COND2 : Condition.

op check : Condition SetEntity -> Bool
eq check(COND1 and COND2,INPUT) =
   check(COND1,INPUT) and check(COND2,INPUT) .
eq check(COND1 or COND2,INPUT) =
   check(COND1,INPUT) or check(COND2,INPUT) .
eq check(E,C) = E in C .
eq check(E ^,C) = not(E in C) .
```

In above equations, `not_`, `_and_` and `_or_` are the boolean operators in Maude, and `_in_` checks if an elements belongs to a set. We also define the auxiliary operator

```
op firable : SetEntity SumProcess -> Bool
```

that checks whether at least one of the prefixes in a summation is enabled.

```
var Kp : PreProcess .  var KS : SumProcess .
var K : Process.
var COND : Condition .
op firable : SetEntity SumProcess -> Bool .
eq firable(IN,{ C } . K) = true .
eq firable(IN,[ COND --> { C } ] . K) =
      check(COND,IN) .
eq firable(IN,Kp + KS) =
      firable(IN,Kp) or firable(IN,KS) .
```

4.4 States and Rules

We consider the following sorts and operators:

```
sorts PState IOState .
subsort PState IOState < State .

op {proc:_ ; in:_} : Process SetEntity -> PState .
op {next:_ ; in:_ ; ctx:_} :
   Process SetEntity SetEntity -> IOState .
op {proc:_ ; in:_ ; out:_ ; ctx:_} :
   Process SetEntity SetEntity SetEntity -> State .
```

Terms of sort `PState` are used to capture the evolution of a process during the current interaction with the RS (relations $\rightsquigarrow$ and $\longrightarrow$ in Figure 1). Terms of sort `IOState` represent the end of an internal computation of the form $K \xrightarrow{(D,C)} K'$, storing the `next` process to be executed (K') along with the context C produced under the input D. The sort `State` is defined in Maude's model-checker prelude [10] to represent the states of the transition system induced by a rewrite theory. The above constructor for this sort allows us to represent the output of a process under the presence of an input set of entities D, and a context C that has been computed by evaluating the process K (relation $(D, K) \xRightarrow{(C)} (D', K')$). The use of the sort `State` will allow us to endow **ccReact** with verification capabilities as shown in Section 5.

Consider a term `{proc: K ; in IN}` of sort `PState`, representing the execution of the process `K` under the input `IN`. The theory $\mathcal{R}_{RS}$ includes rules that allow us to decompose the process K until it produces some context. We consider the different constructors for building the process K.

A process $p.K$ (sort `PreProcess`) produces a context according to the prefix p as follows:

```
rl [prefix] : { proc: ({ C } . K)   ; in: IN }
           => { next: K ; in: IN ; ctx: C } .

rl [prefixC] : { proc: ([ G --> { C } ] . K); in: IN }
  =>  if check(G, IN)
        then { next: K  ; in: IN ; ctx: C }
        else { next: 0  ; in: IN ; ctx: empty }
     fi
```

The right-hand side of these rules are terms of sort `IOState` of the form `{next: K ; in: IN ; ctx: CTX}` that determine the next process to be executed and the context produced. The rule `prefix` produces the context C and the process to be executed in the next interaction is K. If the condition/guard G holds, then the rewrite rule `prefixC` produces the context C and the continuation process is again K. Otherwise, this rule produces the empty context and the next process to be executed is $\mathbf{0}$ (see operational rule $\mathsf{sum}_{\emptyset}$ in Figure 1).

In a summation $M = p.K + KS$, where $KS = \sum_{i\in I} p_i.K_i$, all the enabled (if any) prefixes must be considered. Hence, if p is enabled (i.e., `firable` on input `IN`) or none of the processes in KS are enabled, then the process $p.K$ can be scheduled for execution (rule `choiceL` below). Moreover, if there exists at least one enabled prefix in the continuation KS, KS is also considered for execution (rule `choiceR`).

```
crl [choiceL] : { proc: Kp + KS ; in: IN } =>
    { proc: Kp ; in: IN }
  if firable(IN,Kp) or not firable(IN,KS) .


crl [choiceR] : { proc: Kp + KS ; in: IN } =>
    { proc: KS ; in: IN }
  if firable(IN,KS) .
```

The behavior of the parallel composition $K_1 \parallel K_2$ is specified by the rewrite rule `parallel` which exploits rewrite expressions in its condition to compute the contexts produced by both K_1 and K_2 (namely, `CTX1` and `CTX2`). Then, it combines these contexts together by using the set union operator "`_,_`":

```
vars K1 K2 : Process .
vars CTX1 CTX2 : SetEntity .
crl [parallel] : { proc: K1 || K2 ; in: IN }
   => { next: K1' || K2' ; in: IN ; ctx: (CTX1,CTX2) }
 if   { proc: K1; in: IN }
   => { next: K1'; in: IN ; ctx: CTX1 } /\
      { proc: K2 ; in: IN }
   => { next: K2'; in: IN ; ctx: CTX2 } .
```

For recursive processes, the rule below unfolds the definition of the process with the aid of the function `subst` that computes the substitution $P[\textbf{rec}\ X.P/X]$:

```
rl [rec] : { proc: (rec X . K) ; in: IN } =>
           { proc: subst(X,(rec X . K),K) ; in: IN }.
```

Finally, we introduce a rule that allows us to observe the execution of the system across time-instants:

```
crl [next] : { proc: K ; in: IN ; out: OUT ; ctx: CTX }
        =>   { proc: K' ; in: OUT ;
               out: eval((CTX' , OUT), reactions) ;
               ctx: CTX' }
    if { proc: K  ; in: OUT } =>
       { next: K' ; in: OUT ; ctx: CTX' } .
```

Note that this rule uses the current `OUT`put as an input for the next interaction. The condition of this rule checks whether the term `{ proc: K ; in: OUT }`, of sort `PState`, can be rewritten into a term of sort `IOState`, thus producing a context `CTX'`. The new output is computed by means of the function `eval` (see Section 4.3) that applies the set of `reactions` of the RS under consideration to the set of entities `(OUT , CTX')` (that is, the union of `OUT` and the computed context `CTX'`).

4.5 Adequacy

The representation of entities, reactions and processes as terms of the appropriate sorts is quite natural and makes it easy to define maps $[\![\cdot]\!]$ between RS components and their Maude counterparts, e.g., from sets of entities to terms of sort `SetEntity`, reactions to terms of sort `Reaction`, *etc.* In the following, given any syntactic entity ξ (entity, reaction, *etc.*), we shall use $[\![\xi]\!]$ to denote the corresponding term of the appropriate sort, for instance, $[\![(R,I,P)]\!] =$`[` $[\![R]\!]$ `;` $[\![I]\!]$ `;` $[\![P]\!]$ `]`, $[\![\{e,p\}]\!] =$ `e , p`, *etc.*

The three lemmas below establish the correspondence between the reduction relations in Figure 1 and rewriting steps in $\mathcal{R}_{RS}$.

Lemma 1 (Adequacy of $\rightsquigarrow$) *For all process K, prefix p and set of entities C and D:*

1. *if $p = C$ or $p = c \ \to\ C$ and check(c, D) then $p \overset{D}{\rightsquigarrow} C$ iff*
   $\{$ `proc` : $[\![p.K]\!]$; `in` : $[\![D]\!]$ $\} \longrightarrow_{\mathcal{R}_{RS}}$
   $\{$ `next` : $[\![K]\!]$; `in` : $[\![D]\!]$; `ctx` :$[\![C]\!]$ $\}$*; and*
2. *if $p = c \ \to\ C$ and $\neg$check(c, D) then*
   $\{$ `proc` : $[\![p.K]\!]$; `in` : $[\![D]\!]$ $\} \longrightarrow_{\mathcal{R}_{RS}}$
   $\{$ `next` : $[\![\mathbf{0}]\!]$; `in` : $[\![D]\!]$; `ctx` :$[\![\emptyset]\!]$ $\}$

Proof Both propositions are easy to prove by simply inspecting the rewrite rules `prefix` and `prefixC` and noticing that those are the only rules that can be applied on a term of the form $\{$ `proc:` $[\![p.K]\!]$ `;` `in:` $[\![D]\!]$ $\}$.

Lemma 2 (Adequacy of $\longrightarrow$) *For all processes K, K' and sets of entities C, D, $K \xrightarrow{(D,C)} K'$ iff*
$\{$ `proc` : $[\![K]\!]$; `in` : $[\![D]\!]$ $\} \longrightarrow^*_{\mathcal{R}_{RS}}$
$\{$ `next` :$[\![K']\!]$; `in` :$[\![D]\!]$; `ctx` : $[\![C]\!]$ $\}$

Proof Consider the $\Rightarrow$ side. We proceed by induction on the height of the derivation of $K \xrightarrow{(D,C)} K'$ (and case analysis on the last rule applied).

– sum: there exists an index j s.t. $p_j \overset{D}{\rightsquigarrow} C_j$. We then apply $j-1$ times the rewrite rule `choiceR` (to "select" the prefix p_j) followed by an application of the rule `choiceL` and the result follows from Lemma 1 (item (1)).
– sum$_\emptyset$: necessarily none of the prefixes is enabled. Then, the result follows by applying the rule `choiceL` and using Lemma 1 (item (2)).
– parallel: by induction (on the shorter derivations of the premises of this rule), the two rewrite expressions in rule `parallel` compute the correct contexts. The result follows by noticing that the resulting context is the union of these context (as in the rule parallel).

– rec: by induction and noticing that the premise of the rule rec coincides with the right-hand side of the rewrite rule `rec`.

Now consider the $\Leftarrow$ side. We proceed by induction on the number of rewritings n needed to show that $t = \{$ `proc` $: [\![K]\!]$; `in` $: [\![D]\!]$ $\} \longrightarrow^*_{\mathcal{R}_{RS}}$ $\{$ `next` $: [\![K']\!]$; `in` $: [\![D]\!]$; `ctx` $: [\![C]\!]$ $\}$. If $n = 1$, then K is necessarily a prefix and the result follows from Lemma 1. If $n > 1$, we consider all the possible rules that can be applied on term t. If the rule `parallel` was used, $K = K_1 \parallel K_2$ and there are smaller number of rewritings to compute the resulting contexts for K_1 and K_2. The result follows then by induction. If K is a sum, either `choiceL` or `choiceR` were used and the result follows by induction . If K is **rec** $X.K'$, then the rule `rec` is necessarily applied and the result follows by induction (on the smaller derivation starting from $[\![K[\textbf{rec } X.K/X]]\!]$).

Lemma 3 (Adequacy of $\Longrightarrow$) *For all processes K, K' and sets of entities IN, CTX, C, D, D',*

$(D, K) \xRightarrow{(C)} (D', K')$ *iff*
$\{$ `proc` $: [\![K]\!]$; `in` $: [\![IN]\!]$; `out` $: [\![D]\!]$; `ctx` $: [\![CTX]\!]$ $\}$
$\longrightarrow_{\mathcal{R}_{RS}}$
$\{$ `proc` $: [\![K']\!]$; `in` $: [\![D]\!]$; `out` $: [\![D']\!]$; `ctx` $: [\![C]\!]$ $\}$.

Proof The only rule that can be applied to the term on the right of the arrow $\longrightarrow_{\mathcal{R}_{RS}}$ above is the rewrite rule `next`. Note that the condition of this rule holds by virtue of Lemma 2. The result follows by noticing that the resulting outputs D' and $[\![D']\!]$ are computed with the "same" expressions $res_{\mathcal{A}}(D \cup C)$ and `eval((D,C), reactions)` in $\mathcal{R}_{RS}$.

Recall that a transition system $\mathcal{T}_A$ is a triple of the form $(A, s_\iota, \rightarrow_A)$ where A is a set of states, $s_\iota \in A$ is the *initial* state and $\rightarrow_A \subseteq A \times A$ is a *transition relation*. A relation $\sim \subseteq A \times B$ is a bisimulation from $\mathcal{T}_A$ and $\mathcal{T}_B = (B, b_\iota, \rightarrow_B)$ iff: (1) $a_\iota \sim b_\iota$; and (2) for all a, b s.t. $a \sim b$: if $a \rightarrow_A a'$ then there is b' s.t. $b \rightarrow_B b'$ and $a' \sim b'$, and if $b \rightarrow_B b''$, then there exists a'' s.t. $a \rightarrow_A a''$ and $a'' \sim b''$.

The next result shows that the transition system induced by the relation $\Longrightarrow$ in Figure 1 is bisimilar to the transition system induced by the rewrite theory $\mathcal{R}_{RS}$ on terms of sort `State`. Recall that $T_{\Sigma,s}$ denotes the set of terms of sort s.

Corollary 1 *For all reaction system $\mathcal{A} = (S, A)$ and process K, the transition systems*

$$(\mathbb{P}(S) \times \textbf{Proc}, (\emptyset, K), \Longrightarrow) \text{ and } (T_{\Sigma,\texttt{State}}, t_\iota, \longrightarrow_{\mathcal{R}_{RS}})$$

are bisimilar, where t_ι is the term
$\{$`proc` $: [\![K]\!]$; `in` : `empty` ; `out` : `empty` ; `ctx` : `empty`$\}$.

Proof Note that in rule `next` (the only rule that directly applies to terms of sort `State`) the fields "`in:` " and "`ctx:` " in the left-hand side of the rule are not used to compute the next state. What is needed is the process K and the current output which is the input for the next interaction. Hence, the bisimulation $\sim \subseteq (\mathbb{P}(S) \times \textbf{Proc}) \times T_{\Sigma,\texttt{State}}$ that we are looking for relates any (D, K) with a term $\{$`proc` $: [\![K]\!]$; `in` $: [\![IN]\!]$; `out` $: [\![D]\!]$; `ctx` $: [\![CTX]\!]\}$ for any sets of entities IN and CTX. Using Lemma 3 we show that $\sim$ is indeed a bisimulation.

5 Case Studies and Analyses

In this section we show how different analysis methods including simulation by rewriting, reachability analysis, and LTL/CTL model checking are available within our framework. We showcase the use of **ccReact** for the analysis of the case study presented in Section 2. For this system, we verify results already found in the literature [33], and verify new hypotheses. Furthermore, we consider some case studies from [36] in Section 5.3.

In the forthcoming sections, we shall use the following operator:

```
op init : Process -> State .
eq init(K) = { proc: K ; in: empty ;
               out: empty ; ctx: empty } .
```

that allows us to build an `initial` state from a given process K (where the initial input, output and context are set to empty).

5.1 Simulation and State Reachability

Our Maude specification provides an executable model for **ccReact** that can be directly used to simulate interactive processes of RSs. This can be achieved by using the built-in Maude command `rew [n]` that performs a rewrite sequence involving `n` rewrite steps.

For instance, consider the RS in Example 1. The following specification declares the entities and **ccReact** process to represent this system.

```
ops egf  e  p  erbb1  erk12  p70s6k : -> Entity .
eq reactions = [ egf ; (e , p) ; erbb1 ] ,
               [ egf ; empty ; erk12 ],
               [ erk12 ; empty ; p70s6k ] .
op proc : -> Process .
eq proc = {(egf,e)} . {empty}. {egf}. {empty}. 0 .
```

The process `proc` specifies the context sequence $\gamma = C_0, \ldots, C_3$ of the interactive process $\pi = (\gamma, \delta)$ in Example 1. The initial state `init(proc)` sets the first output D_0 to empty. The command `rew` below computes the unique successor of `init` (and the corresponding output D_1). The "continue" commands ("`cont 1`" below) computes D_1 and D_2[1]. Compare these results with those in Example 1:

```
Maude> rew [1] init(proc) .
result State: {proc: {empty} . {egf} . {empty}. 0 ;
               in: empty ; out: erk12 ; ctx: egf , e}

Maude> cont 1 .
result State: { proc: {egf} . {empty} . 0 ;
                in: erk12 ; out: p70s6k ; ctx: empty }

Maude> cont 1 .
result State: { proc: {empty} . 0 ; in: p70s6k ;
                out: erbb1 , erk12 ; ctx: egf }
```

More interestingly, we can search for all the `States` reachable from the initial state `init(proc)` by using the Maude `search` command, thus obtaining the full result sequence δ from the interactive process π in Example 1[2]:

```
Maude> search init(proc) =>* S:State .

Solution 1 (state 0)
S:State --> { proc: {(egf, e)} . {empty}. ... 0 ;
              in: empty ; out: empty ; ctx: empty }

Solution 2 (state 1)
S:State --> { proc: {empty} . {egf} . {empty}. 0 ;
              in: empty ; out: erk12 ; ctx: egf , e }

Solution 3 (state 2)
S:State --> { proc: {egf} . {empty} . 0 ; in: erk12 ;
              out: p70s6k ; ctx: empty }

Solution 4 (state 3)
S:State --> { proc: {empty} . 0 ; in: p70s6k ;
              out: erbb1 , erk12 ; ctx: egf }

Solution 5 (state 4)
S:State --> { proc: 0 ; in: erbb1 , erk12 ;
              out: p70s6k ; ctx: empty }

No more solutions.
```

The Maude search capability can be used to answer more complex reachability queries. For instance, we can fully reproduce the results in [33] (see Table 1). We start by defining a predicate `isSteady` to check whether the steady state (i.e. an attractor) has been reached. In Maude, predicates (or atomic propositions) are terms of the sort `Prop` whose meaning is defined by the operator

```
op _|=_ : State Prop -> Bool .
```

[1] Some outputs are simplified (...). All the experiments can be reproduced with our tool [12].

[2] The arrow `init =>* S:State` in the `search` command means "find the terms of sort `State` that can be reached in zero or more rewriting steps from `init`". Maude outputs an "extra" configuration D_4 (Solution 5), that corresponds to $D_4 = res_A(D_3 \cup C_3)$. "`(state n)`" is the number of the state in the search tree generated by Maude.

This operator specifies when a state (a term of sort `State`) satisfies a given proposition. We then identify steady states by checking when the input and output are the same (operator `==`):

```
op isSteady : -> Prop . --- Is it a steady state?
eq { proc: K ; in: IN ;
     out: OUT ; ctx: CTX } |= isSteady
  =  IN == OUT .
```

In some analyses, we will be interested in checking whether the output of the system contains a given set of entities. Hence, we define the following predicate:

```
--- Sets of entities as propositions
subsort SetEntity < Prop .
eq { proc: K ; in: IN ; out: OUT ; ctx: CTX } |= C
   = C subset OUT .
```

Note that due to the above subsort relation, a set of entities C is also an atomic proposition. The atomic proposition C holds in a given state iff C is a `subset` of the `OUT`put in that state. Similarly, we can check whether a given entity was provided as part of the current context:

```
op inCtx : Entity -> Prop .
eq {proc: K ; in: IN ; out: OUT ; ctx: CTX}
          |= inCtx(E) = E in CTX .
```

The results in Table 1 can be verified by checking the presence of specific molecules (e.g., AKT) in the steady states, when considering a **ccReact** process of the form !(S, D). This process provides a context sequence including the stimulus S ={EGF,HRG} together with the administered drugs, e.g., D = {e, t}. For instance, for the BT474 cancer, the combination of erlotinib and trastuzumab is effective, whereas BT474 is resistant to the treatment with the drugs erlotinib and pertuzumab. In all the queries below, we assume that `St` is a variable of sort `State`:

```
Maude> search [1] init(!(hrg, egf, e, t))
 =>* St such that St |= (akt)  /\ St |= isSteady .

No solution.

Maude> search [1] init(!(hrg, egf, e, p))
 =>* St such that St |= (akt)  /\ St |= isSteady .

Solution 1 (state 3)
St:State --> {proc: ... ; in: ... ;
              out: akt, p70s6k, mtor, ... }
```

5.2 Model-checking

The Unified Maude Model-Checking tool (`umaudemc`)[44] is a general interface to different model checkers for Maude models. Here we show how `umaudemc` can be used to formally verify LTL as well as CTL temporal formulas against our models for HER2-positive cancers.

LTL and CTL temporal formulas are build from atomic `Proposition`, boolean connectives and the usual temporal operators. In the case of LTL, we have $\Box$ (always), $\Diamond$ (eventually), $\circ$ (next), $\mathbf{U}$ (until) and $\mathbf{R}$ (release). These operators are quantified in CTL over the paths of the computation tree (there **E**xists or for **A**ll paths). Temporal logic formulas can then be model-checked using the command `check` of `umaudemc` whose parameters include the file with the Maude specification of the system to be verified, the initial state (a term of sort `State`) and the temporal formula to be checked.

The results of [33] that we reproduced using the `search` commands above can alternatively be obtained via model-checking. More concretely the presence/absence of the AKT molecule in steady states of the BT474 cell line can be checked by feeding `umaudemc` with the LTL formula $\Diamond\Box(\texttt{isSteady} \wedge \texttt{akt})$ (eventually the system remains in a steady state where `akt` is present) and an initial state that specifies the stimulus and drug combination to be considered:

```
--- Stimulus: HRG+EGF Drug combination: e+t
$> umaudemc check BT474 "init(!(hrg, egf, e, t))"
   "<> [] (isSteady /\ akt)"


The property is not satisfied in the initial state ...
    counterexample ...

--- Stimulus: HRG+EGF Drug combination: e+p
$> umaudemc check BT474 "init(!(hrg, egf, e, p))"
   "<> [] (isSteady /\ akt)"


The property is satisfied in the initial state
```

Using temporal formulas, we can go beyond the results in [33] and test other hypotheses. For instance, we can check that it is always the case that once the growth factor receptor ErbB-2 is present in a time-instant t, then it is also present in the time-instant $t + 1$, thus highlighting a persistent activity of the ErbB-2 receptor across the pathway. This property is specified by the LTL formula $\Box(\texttt{erbb2} \rightarrow \circ\texttt{erbb2})$:

```
$> umaudemc check BT474 "init(!(hrg, egf))"
   "[] (erbb2 -> O erbb2)"

The property is satisfied in the initial state
```

We can use CTL to check the behavior of a drug treatment on different runs. For instance, in order to observe the interactions when either erlotinib or pertuzumab are supplied, we consider the following process, providing as stimulus either `e` or `p` under the persistent presence of ligands `hrg` and `egf`:

```
(rec 'Y . { (hrg , egf) } . 'Y) ||
(rec 'X . ({ e } . 'X + { p } . 'X))
```

For this process, Maude reports that the formula $\mathbf{E}\diamond(\texttt{isSteady} \wedge \texttt{akt})$ is satisfied (there **E**xists at least one path where that treatment is not successful) but $\mathbf{A}\diamond(\texttt{isSteady} \wedge \texttt{akt})$ is not (not **A**ll paths end in a steady state where AKT is present).

When a CTL formula is refuted, as happened in the case above, a further analysis can be carried out to identify the path that violates the formula. This can be done by generating and inspecting the computation tree via the Maude `search` command as explained in Section 5.1. However, `search` lacks a user-friendly interface that facilitates the inspection of the computation space of the RS under consideration. A valid solution to this problem is offered by the `ANIMA` system [7] —a visual program explorer for Maude computations that fully accepts **ccReact** specifications. In `ANIMA`, system biologists can stepwise explore the computation space of RSs by expanding state transitions, one at a time, with a simple point-and-click interface. Therefore, it is possible to incrementally produce a visual representation of the whole computation tree w.r.t. a given initial state. For instance, Figure 2 shows a fragment of the computation space, which have been generated in `ANIMA` for the initial state

```
init((rec 'Y . { ( hrg , egf ) } . 'Y) ||
     (rec 'X . ( { e } . 'X + { p } . 'X)))
```

Note that the leftmost branch of the tree reaches a steady state, which does not contain AKT, in just four computation steps, thereby providing a counterexample for the refuted property $\mathbf{A}\diamond(\texttt{isSteady} \wedge \texttt{akt})$.

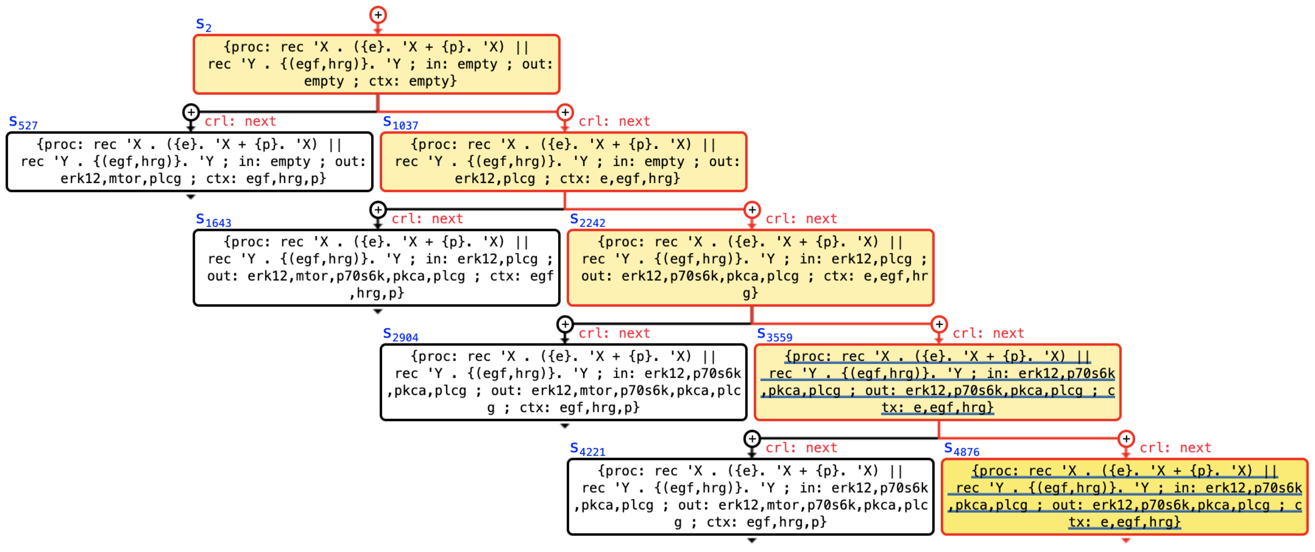
We can also check properties about the transient states, describing the evolution of the RS before reaching the steady state. Consider, e.g., the process

```
(rec 'Y . ({ (hrg , egf )} . 'Y) ||
(rec 'X . (({e} . 'X) + ({p} . 'X) + ({t} . 'X)))
```

Maude returns that, for BT474, the following two LTL formulas hold:
(i) $\Box(\texttt{pdk1} \rightarrow (\Diamond\Box\texttt{akt}))$ and
(ii) $\texttt{erbb1}\ \mathbf{R}\ \neg\texttt{pdk1}$.

This positively answers the following questions:
(i) regardless the drug used, once PDK1 is present, inevitably the steady state includes AKT; and
(ii) PDK1 never appears before ErbB-1 is produced (which basically means that PDK1 is a product of the activation of the ErbB-1 receptor).

The results for the short-term experiments in [33] show that by *permanently* providing the drug erlotinib and the stimulus (EGF and HRG), the AKT molecule is never produced. By using **ccReact** conditional prefixes, we can ask whether the drug needs to be provided in *all* the time-instants or can be provided only when an EGF receptor is activated. For that, consider the following



```
s2:    {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: empty ; out: empty ; ctx: empty}
s527:  {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: empty ; out: erk12,mtor,plcg ; ctx: egf,hrg,p}
s1037: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: empty ; out: erk12,plcg ; ctx: e,egf,hrg}
s1643: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: erk12,plcg ; out: erk12,mtor,p70s6k,pkca,plcg ; ctx: egf,hrg,p}
s2242: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: erk12,plcg ; out: erk12,p70s6k,pkca,plcg ; ctx: e,egf,hrg}
s2904: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: erk12,p70s6k,pkca,plcg ; out: erk12,mtor,p70s6k,pkca,plcg ; ctx: egf,hrg,p}
s3559: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: erk12,p70s6k,pkca,plcg ; out: erk12,p70s6k,pkca,plcg ; ctx: e,egf,hrg}
s4221: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: erk12,p70s6k,pkca,plcg ; out: erk12,mtor,p70s6k,pkca,plcg ; ctx: egf,hrg,p}
s4876: {proc: rec 'X . ({e}. 'X + {p}. 'X) || rec 'Y . {(egf,hrg)}. 'Y ; in: erk12,p70s6k,pkca,plcg ; out: erk12,p70s6k,pkca,plcg ; ctx: e,egf,hrg}
```

Fig. 2 Fragment of a computation tree generated by ANIMA.

process (recall the definition of the process **if** c **then** $C_1.K_1$ **else** $C_2.K_2$ in Example 2):

```
!(hrg , egf)  ||
(rec 'X . if (erbb1 or erbb2) then { e } . 'X
          else { empty } . 'X
```

With respect to the process above, we can check that the properties
(i) $\Box($`isSteady` $\to \neg$`akt`$)$; and
(ii) $\circ\Diamond(\neg$`erbb1` $\wedge \neg$`erb2`$)$
both hold. This means that the production of AKT can be inhibited by providing erlotinib only when receptor ErbB-1 or ErbB-2 are active (Property (i)). Also, by Property (ii), we can also confirm that there is at least one state st where these receptors are not present and thus the drug `e` is not administered in st.

5.3 Further Experiments

The rewriting framework proposed here for the verification of reaction systems is robust enough to check some of the properties specified using the temporal logic for reaction systems (rsCTL) [36]. In rsCTL, the CTL path quantifiers are decorated with a set of sets of entities $\mathfrak{S} \subseteq \mathbb{P}(S')$, where $S' \subseteq S$ is the set of entities controlled by the environment. The set $\mathfrak{S}$ indicates that the paths to be verified are restricted to input contexts in $\mathfrak{S}$. More precisely, a formula $\mathbf{E}_{\mathfrak{S}}\phi$ (resp. $\mathbf{A}_{\mathfrak{S}}\phi$) holds if ϕ holds in some (resp. all) path where every context C_i in the context sequence γ satisfies $C_i \in \mathfrak{S}$.

Below we consider rsCTL benchmarks proposed in [36]. We abstract away from the particular entities and reactions involved in each experiment to focus on the expressiveness of rsCTL and **ccReact** frameworks. The full specification of each rsCTL experiment can be found in [36], while the corresponding Maude files with the systems and formulas verified in **ccReact** can be found in [12].

Heat shock response model. In [9], a RS for modeling the eukaryotic heat shock response system was proposed. This system reacts to the environment when the temperature increases. The environment controls the entities *stress* and *nostress*. rsCTL formulas are used in [36] to verify properties of this system where path quantifiers are indexed with two possible sets:
- $\mathfrak{S}_1 = \{\{stress\}, \{nostress\}\}$; and
- $\mathfrak{S}_2 = \{\{nostress\}\}$.

The set $\mathfrak{S}_1$ models the situation where the environment provides as a signal either the entity *stress* or the entity *nostress* (but never the empty context). In the case of $\mathfrak{S}_2$, the behavior of the system is studied in the absence of stress. Moreover, following [9], the authors of [36] consider three initial states, each of them consisting of a different set $S_{i\in\{1,2,3\}}$ of entities.

Six different rsCTL formulas are considered in [36], all of them of the form $\mathbf{A}_{\mathfrak{S}_j}\Box(\phi \to \mathbf{A} \circ_{\mathfrak{S}_j} \psi)$, for $j \in \{1,2\}$. In these formulas, ϕ and ψ are state formulas, i.e., formulas not containing temporal operators such as $e_1 \wedge \neg e_2$ where $e_{i\in\{1,2\}}$ is an entity.

In order to verify these rsCTL formulas in our framework, it suffices to use `umaudemc` to model check the initial state `init(`K`)` where

$$K = \sum_{i\in\{1,2,3\}} S_i.K_{\mathbb{P}}(\mathfrak{S}_j)$$

against the CTL formula $\mathbf{A}\Box(\phi \to \mathbf{A}\circ\psi)$. Note that the process K provides as stimulus the entities in the initial state S_i. After that, only contexts in $\mathfrak{S}_j$ are generated, thus reflecting the same paths considered by the original rsCTL formulas.

Binary counter. In [21] a n-bit cyclic binary counter is modeled as a RS where the set of entities is $S = \{p_0, \cdots, p_{n-1}, inc, dec\}$. An arbitrary n-bit number is represented by a state containing the entities p_i whose corresponding bit is 1. For instance, the number 1010 in represented by the state $\{p_1, p_3\}$. The environment uses the entities *inc* and *dec* to increase or decrease the counter. The initial state is the empty set representing the number 0.

Below we present the rsCTL formulas verified in [36] along with the corresponding **ccReact** processes and CTL formulas that can be used in our framework to verify the same property. In all the cases, ϕ and ψ are state formulas. Moreover, when the set of contexts is omitted in a path quantifiers of a rsCTL formula (e.g., as in $\mathbf{E}\Diamond\phi$), such set is implicitly taken to be $\mathfrak{S} = \mathbb{P}(\{inc, dec\})$, i.e., all the subsets (including the empty set) of the entities controlled by the environment ($S' = \{inc, dec\}$).

1. $\mathbf{A}\Box(\phi \to \mathbf{E}_{\mathfrak{S}_1}\Diamond\psi)$ where $\mathfrak{S}_1 = \{\{inc\}, \{dec\}\}$. We consider the **ccReact** process K and CTL formula F below:

   $$K = K_{\mathbb{P}}(S')$$
   $$F = \mathbf{A}\Box(\phi \to \mathbf{E} \circ \mathbf{E}(\chi \mathbf{U}(\psi \wedge \chi)))$$

   where $\chi = inCtx(inc) \oplus inCtx(dec)$ and $\oplus$ is the XOR operator. The first temporal connective $\mathbf{A}\Box$ in the rsCTL formula checks all the possible paths where the environment can provide any subset of S' as a context. This is also the case for the process K and the CTL formula $\mathbf{A}\Box$. Moreover, the rsCTL formula holds if for all ϕ-state there is a path leading to a ψ-state where the input contexts are all elements of $\mathfrak{S}_1$. This is captured in our CTL formula by checking that eventually ψ holds and, along the path, the input from the environment is either *inc* or *dec* (but not both), i.e., the formula χ holds in all the states of that path. This means that a path where the context provides none of those entities, or both of them, cannot be used as a witness for the **U**ntil formula in F.
2. $\mathbf{A}(\phi \to \mathbf{A}_{\{\{inc\}\}} \circ \psi)$. In this case, we consider the same process K of the previous item and the CTL formula below:

   $$\mathbf{A}\Box(\phi \to \mathbf{A} \circ ((inCtx(inc) \wedge \neg inCtx(dec)) \to \psi))$$

   This means that in all the successor states of a ϕ-state: either the input context is not $\{inc\}$ (and then such path is not considered by the rsCTL formula and the implication trivially holds) or the input context is $\{inc\}$ (and $inCtx(inc) \wedge \neg inCtx(dec)$ holds) and then ψ must hold in that state too.
3. $\mathbf{E}_{\{inc\}}\Diamond\phi$. In this case, it suffices to consider the process $!inc$ and the CTL formula $\mathbf{E}\Diamond\phi$.
4. $\mathbf{A}\Box(\phi \to \mathbf{E}\Diamond\psi)$. In this case, we consider the process K, that we specified in Item 1, and the CTL formula $\mathbf{A}\Box(\phi \to \mathbf{E}\Diamond\psi)$.

The other two examples in [36], including a mutual exclusion protocol and a pipeline system, use rsCTL formulas similar to the ones presented above. Hence, they can be verified within **ccReact** (please, see [12] for their full formalization in **ccReact**).

6 Related Work

There are some frameworks implemented in Maude that can be compared to ours. [11] provides a Maude formalization of RSs which is suitable for trace slicing, a technique used to simplify computation traces to help the modelist to detect bugs in the computational model. The methodology in [11] needs context sequences to be explicitly given by the user, whereas our approach defines a process algebra which allows us to specify richer and more expressive conditional nondeterministic contexts. Pathway Logic (PL) [46] is a symbolic approach to the modeling and analysis of biological systems that is implemented in Maude. PL allows signaling pathways to be simulated and formally verified. However, PL is limited to the analysis of cell models, while our framework is more abstract and can be applied to multiple application domains (even beyond the biological context). In [6], an anti-unification algorithm, specified in Maude, has been used to extract similarities and pinpoint discrepancies between two cell models that express distinct cellular states that appear in the MAPK (Mitogen-Activated Protein Kinase) signaling pathway that regulates growth, survival, proliferation, and differentiation of mammalian cells. [6] does not consider RSs and process algebras, the flexible contexts we provide here, and the model checking tools for the verification of properties.

[36] introduces rsCTL, a CTL logic for RSs where a collection of entity sets is provided by the context, thus generating multiple paths. As shown in Section 5.3, the rsCTL formulas considered in [36] can be model checked within our framework. In [39], an LTL logic was introduced for a version of RSs with discrete concentrations: in each reaction, the sets of reactants, inhibitors, and products are multisets, and the interactive process computes the maximum concentration among all the

enabled reactions for each produced entity. In [39], the interactive process is modeled as an automaton whose transitions are labeled with the entities to be produced. These context automata can be directly encoded as **ccReact** processes (using recursion and prefixes). As illustrated in Section 5, our framework supports model checking LTL formulas. Considering concentrations in our framework remains future work though: it can be achieved by removing idempotency in sets (thus turning them into multisets) or, more efficiently, by considering maps (an algebraic data type already specified in Maude) from entities to natural numbers. Unlike our work, [36] and [39] use ad-hoc implemented model checkers. In contrast, it is possible to connect `umaudemc` with any CTL/LTL model checker, thereby making our results verifiable by different tools and enhancing the robustness of our approach.

Nondeterministic contexts have already appeared in the literature [35, 23, 24], but here we are able to represent a collection of experiments within a single semantic object, so that the consequences of some variation in the context sequence can then be more easily compared and analyzed. Our transition labels record the information about the usage of each entities involved in the transition step, thus we can specify finer properties than in [23, 24]. The works in [23, 24], on the other hand, define also the reactions of the RS as processes, while our definition of the set of reactions is not compositional, as we fix it from the beginning. The major difference with respect to [24] is that we can model check CTL and LTL formulas to verify temporal properties of our specifications. Also, our work adds the *conditional* construct to the context behavior. This construct is particular useful when the context should provide one or more entities in response to the presence or absence of some entities in the system states.

7 Conclusions and Future Work

Modeling and analyzing biological systems are challenging tasks that demand the support of appropriate languages and tools to help streamline and automate system biologists work. In this paper, we presented **ccReact**, a language of processes that formally captures the dynamic behavior of RSs. **ccReact** is endowed with a rich process algebra that allows for the definition of multiple external contexts that can drive the execution of a given RS w.r.t. different environment stimuli. This way, several experiments can be simulated under different external conditions within a single **ccReact** run, thereby overcoming the usual limitation of *in silico* experiments where distinct external contexts are tried sequentially. **ccReact** has been specified in Maude, a rewriting-based system that offers both fast performance and an easy interconnection with several analysis tools. To this respect, we have shown how Maude built-in reachability capabilities, the `umaudemc` tool, and the graphical tool `ANIMA` can be employed to effectively analyze different RSs. We have also shown how our analysis methods can corroborate previous results in the literature ([36], [33]) and validate new hypotheses related to the HER2-positive breast cancer subtype.

As future work we want to consider other case studies, such as the rare disease Alkaptonuria [20], for which we are interested to analyze the evolution of the disease under different drug treatments. We want also to investigate a possible integration of static analysis and slicing techniques [18, 19, 5, 1, 2, 3, 17] within our framework, with the aim of applying it to distributed systems [22]. Furthermore, we plan to enrich the data structure for entities and reactions with more structured values, exploiting some ideas from [4, 6]. We are also exploring the integration of our framework with BioReSolve [16, 24], an implementation of RSs in SWI-Prolog [45] that allows one to (i) specify biological systems in a pure logic setting, and (ii) assess bio-similarities between different systems. We note that BioReSolve does not provide either model checking capabilities, or conditional contexts. By defining an interface between our implementation in Maude and BioReSolve we could combine the reciprocal advantages. Finally, another future research line would be to extend our work to the case of quantitative RSs [27, 26].

References

1. Alpuente M, Ballis D, Romero D (2014) A Rewriting Logic Approach to the Formal Specification and Verification of Web Applications. Science of Computer Programming 81:79–107, doi:https://doi.org/10.1016/j.scico.2013.07.014
2. Alpuente M, Ballis D, Frechina F, Sapiña J (2015) Combining Runtime Checking and Slicing to improve Maude Error Diagnosis. In: Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer, Springer, Lecture Notes in Computer Science, vol 9200, pp 72–96, doi:10.1007/978-3-319-23165-5_3
3. Alpuente M, Ballis D, Frechina F, Sapiña J (2016) Assertion-based Analysis via Slicing with ABETS. Theory and Practice of Logic Programming 16(5–6):515–532, doi:10.1017/S1471068416000375
4. Alpuente M, Ballis D, Cuenca-Ortega A, Escobar S, Meseguer J (2019) ACUOS^2: A High-Performance

System for Modular ACU Generalization with Subtyping and Inheritance. In: Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA 2019), Springer, Lecture Notes in Computer Science, vol 11468, pp 171–181, doi:10.1007/978-3-030-19570-0_11

5. Alpuente M, Ballis D, Sapiña J (2019) Static Correction of Maude Programs with Assertions. Journal of Systems and Software 153:64–85, doi:10.1016/j.jss.2019.03.061
6. Alpuente M, Ballis D, Escobar S, Sapiña J (2022) Variant-Based Equational Anti-unification. In: Proc. of Logic-Based Program Synthesis and Transformation - 32nd International Symposium, LOPSTR 2022, Springer, Lecture Notes in Computer Science, vol 13474, pp 44–60, doi:10.1007/978-3-031-16767-6_3
7. Anima (2015) The Anima Website. Available at: `http://safe-tools.dsic.upv.es/anima`
8. Azimi S (2017) Steady states of constrained reaction systems. Theor Comput Sci 701(C):20–26, doi:10.1016/j.tcs.2017.03.047
9. Azimi S, Iancu B, Petre I (2014) Reaction system models for the heat shock response. Fundam Informaticae 131(3-4):299–312, doi:10.3233/FI-2014-1016
10. Bae K, Meseguer J (2010) The linear temporal logic of rewriting maude model checker. In: Ölveczky PC (ed) Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 6381, pp 208–225, doi:10.1007/978-3-642-16310-4_14
11. Ballis D, Brodo L, Falaschi M (2024) Modeling and Analyzing Reaction Systems in Maude. Electronics 13(6,1139), doi:https://doi.org/10.3390/electronics13061139
12. Ballis D, Brodo L, Falaschi M, Olarte C (2024) ccReact: An interacting language for reaction systems. Available at `https://github.com/carlosolarte/ccReact`
13. Balls D, Brodo L, Falaschi M, Olarte C (2024) Process calculi and rewriting techniques for analyzing reaction systems. In: Computational Methods in Systems Biology (CMSB'24), Springer Nature Switzerland, Pisa, URL `https://depot.lipn.univ-paris13.fr/olarte/reaction-systems-maude`
14. Barbuti R, Gori R, Levi F, Milazzo P (2016) Investigating dynamic causalities in reaction systems. Theor Comput Sci 623:114–145, doi:https://doi.org/10.1016/j.tcs.2015.11.041
15. Barbuti R, Gori R, Milazzo P (2021) Encoding boolean networks into reaction systems for investigating causal dependencies in gene regulation. Theoretical Computer Science 881:3–24, doi:https://doi.org/10.1016/j.tcs.2020.07.031, special Issue on Reaction Systems
16. bioresolve (2021) BioReSolve web page, a prolog interpreter for Reaction Systems analysis. URL `http://pages.di.unipi.it/bruni/LTSRS/`, accessed: 25 July 2024
17. Bodei C, Brodo L, Degano P, Gao H (2010) Detecting and preventing type flaws at static time. Journal of Computer Security 18(2):229–264
18. Bodei C, Brodo L, Focardi R (2015) Static evidences for attack reconstruction. In: Bodei C, Ferrari G, Priami C (eds) Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday, Springer, Lecture Notes in Computer Science, vol 9465, pp 162–182, doi:10.1007/978-3-319-25527-9_12
19. Bodei C, Brodo L, Gori R, Levi F, Bernini A, Hermith D (2017) A static analysis for brane calculi providing global occurrence counting information. Theor Comput Sci 696:11–51, doi:10.1016/J.TCS.2017.07.008, URL `https://doi.org/10.1016/j.tcs.2017.07.008`
20. Braconi D, Millucci L, Spiga O, Santucci A (2020) Cella and tissue models of alkaptonuria. Drug Discovery Today: Disease Models 31:3–10, doi:10.1016/j.ddmod.2019.12.001
21. Brijder R, Ehrenfeucht A, Main M, Rozenberg G (2011) A tour of reaction systems. Int J Found Comput Sci 22(07):1499–1517, doi:https://doi.org/10.1142/S0129054111008842
22. Brodo L (2011) On the expressiveness of the π-calculus and the mobile ambients. In: Johnson M, Pavlovic D (eds) Algebraic Methodology and Software Technology, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 44–59, doi:10.1007/978-3-642-17796-5_3
23. Brodo L, Bruni R, Falaschi M (2019) Enhancing reaction systems: A process algebraic approach. In: Alvim M, Chatzikokolakis K, Olarte C, Valencia F (eds) The Art of Modelling Computational Systems, Springer Berlin, LNCS, vol 11760, pp 68–85, doi:10.1007/978-3-030-31175-9_5
24. Brodo L, Bruni R, Falaschi M (2021) A logical and graphical framework for reaction systems. Theoretical Computer Science 875:1–27, doi:https://doi.org/10.1016/j.tcs.2021.03.024

25. Brodo L, Bruni R, Falaschi M (2021) A process algebraic approach to reaction systems. Theor Comput Sci 881:62–82, doi:10.1016/j.tcs.2020.09.001
26. Brodo L, Bruni R, Falaschi M, Gori R, Levi F, Milazzo P (2021) Exploiting modularity of SOS semantics to define quantitative extensions of reaction systems. In: Aranha C, Martín-Vide C, Vega-Rodríguez MA (eds) Proceedings of TPNC 2021, Springer, Cham, Lecture Notes in Computer Science, vol 13082, pp 15–32, doi:10.1007/978-3-030-90425-8_2
27. Brodo L, Bruni R, Falaschi M, Gori R, Levi F, Milazzo P (2023) Quantitative extensions of reaction systems based on SOS semantics. Neural Comput Appl 35(9):6335–6359, doi:10.1007/s00521-022-07935-6, URL `https://doi.org/10.1007/s00521-022-07935-6`
28. Clavel M, Durán F, Eker S, Lincoln P, Martí-Oliet N, Meseguer J, Talcott CL (eds) (2007) All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol 4350. Springer, doi:10.1007/978-3-540-71999-1
29. Corolli L, Maj C, Marinia F, Besozzi D, Mauri G (2012) An excursion in reaction systems: From computer science to biology. Theor Comput Sci 454:95–108, doi:https://doi.org/10.1016/j.tcs.2012.04.003
30. Durán F, Eker S, Escobar S, Martí-Oliet N, Meseguer J, Rubio R, Talcott CL (2020) Programming and symbolic computation in Maude. J Log Algebraic Methods Program 110, doi:10.1016/J.JLAMP.2019.100497
31. Ehrenfeucht A, Main MG, Rozenberg G (2010) Combinatorics of life and death for reaction systems. Int J Found Comput Sci 21(3):345–356, doi:10.1142/S0129054110007295
32. Ehrenfeucht A, Main MG, Rozenberg G (2011) Functions defined by reaction systems. Int J Found Comput Sci 22(1):167–178, doi:10.1142/S0129054111007927
33. der Heyde SV, Bender C, Henjes F, Sonntag J, Korf U, Beißbarth T (2014) Boolean ErbB network reconstructions and perturbation simulations reveal individual drug response in different breast cancer cell lines. BMC Systems Biology 8(1):75, doi:10.1186/1752-0509-8-75
34. Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall
35. Kleijn J, Koutny M, Mikulski Ł, Rozenberg G (2018) Reaction systems, transition systems, and equivalences. In: Böckenhauer H, Komm D, Unger W (eds) Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday, Springer, LNCS, vol 11011, pp 63–84, doi:10.1007/978-3-319-98355-4_5
36. Męski A, Penczek W, Rozenberg G (2015) Model checking temporal properties of reaction systems. Information Sciences 313:22–42, doi:10.1016/j.ins.2015.03.048
37. Meseguer J (1992) Conditioned rewriting logic as a united model of concurrency. Theor Comput Sci 96(1):73–155, doi:10.1016/0304-3975(92)90182-F
38. Meseguer J (2012) Twenty years of rewriting logic. J Log Algebraic Methods Program 81(7-8):721–781, doi:10.1016/J.JLAP.2012.06.003
39. Meski A, Koutny M, Penczek W (2017) Verification of linear-time temporal properties for reaction systems with discrete concentrations. Fundam Informaticae 154(1-4):289–306, doi:10.3233/FI-2017-1567
40. Milner R (1989) Communication and concurrency. PHI Series in computer science, Prentice Hall
41. Milner R, Parrow J, Walker D (1992) A calculus of mobile processes (I and II). Information and Computation 100(1):1–77
42. Okubo F, Yokomori T (2016) The computational capability of chemical reaction automata. Natural Computing 15(2):215–224, doi:10.1007/s11047-015-9504-7
43. Olarte C, Rueda C, Valencia FD (2013) Models and emerging trends of concurrent constraint programming. Constraints An Int J 18(4):535–578, doi:10.1007/S10601-013-9145-3, URL `https://doi.org/10.1007/s10601-013-9145-3`
44. Rubio R, Martí-Oliet N, Pita I, Verdejo A (2021) Strategies, model checking and branching-time properties in maude. J Log Algebraic Methods Program 123:100700, doi:10.1016/J.JLAMP.2021.100700
45. swiprolog (2021) SWI-Prolog home page. URL `https://www.swi-prolog.org/`, accessed: 25 July 2024
46. Talcott C (2008) Pathway Logic. In: Proceedings of the 8th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2008), Springer, Lecture Notes in Computer Science, vol 5016, pp 21–53, doi:10.1007/978-3-540-68894-5_2