Article

Modeling and Analyzing Reaction Systems in Maude

Demis Ballis [1,*] , **Linda Brodo** [2] **and Moreno Falaschi** [3]

[1] Dipartimento di Scienze Matematiche, Informatiche e Fisiche, Università degli Studi di Udine, 33100 Udine, Italy

[2] Dipartimento di Scienze Economiche e Aziendali, Università di Sassari, 07100 Sassari, Italy; brodo@uniss.it

[3] Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche, Università di Siena, 53100 Siena, Italy; moreno.falaschi@unisi.it

[*] Correspondence: demis.ballis@uniud.it

Abstract: Reaction Systems (RSs) are a successful computational framework for modeling systems inspired by biochemistry. An RS defines a set of rules (reactions) over a finite set of entities (e.g., molecules, proteins, genes, etc.). A computation in this system is performed by rewriting a finite set of entities (a computation state) using all the *enabled* reactions in the RS, thereby producing a new set of entities (a new computation state). The number of entities in the reactions and in the computation states can be large, making the analysis of RS behavior difficult without a proper automated support. In this paper, we use the Maude language—a programming language based on rewriting logic—to define a formal executable semantics for RSs, which can be used to precisely simulate the system behavior as well as to perform reachability analysis over the system computation space. Then, by enriching the proposed semantics, we formalize a forward slicer algorithm for RSs that allows us to observe the evolution of the system on both the initial input and a fragment of it (the slicing criterion), thus facilitating the detection of forward causality and influence relations due to the absence/presence of some entities in the slicing criterion. The pursued approach is illustrated by a biological reaction system that models a gene regulation network for controlling the process of differentiation of T helper lymphocytes.

Keywords: reaction systems; natural computing; Maude; rewriting logic; slicing; formal methods

1. Introduction

Reaction systems (RSs) [1,2] are a computational framework inspired by the functioning of living cells and by biochemistry. Their constituents are a finite set of entities (a background set) and a finite set of reactions. Each reaction is a triple that consists of a set of entities whose presence is needed to enable the reaction, called *reactants*; a set of entities whose absence is needed to enable the reaction, called *inhibitors*; and a set of entities that are produced when the reaction takes place, called *products*. All entities must be included in a fixed background set. Applications of RSs are very general and range from the modeling of biological phenomena [3–5] to molecular chemistry [6]. The classical behavior of RSs is defined as a rewriting system whose states are sets of entities (those produced at the previous step, possibly joined with others provided by an external context that models the interaction with the environment).

1.1. Problem Statement

The number of entities in the reactions and in the computation states can be large and difficult to verify for its correctness by the users. Thus, automated verification and debugging tools can be very helpful.

The design of RSs for modeling some natural phenomenon is often conducted by domain experts to validate their hypotheses and requires some degree of abstraction. False assumptions or inaccuracies may be introduced as early as the design stage. Moreover,

writing reactions is an error-prone activity, and verifying their behavior can be difficult even for RSs with just a few dozens of entities and reactions. For example, if some mistake is made at the design level and some inexplicable result is observed during the simulation, then an analysis of the computation may be necessary to understand the nature of the problem.

1.2. The Approach

Program slicing is a traditional method used to circumscribe the key parts of a program that may be responsible for a specific unexpected outcome. Originally, slicing was defined as a static technique by [7]. Later, Ref. [8] expanded upon this concept by introducing dynamic program slicing, which aids in the debugging process by identifying—at runtime—those fragments of a program containing the flawed code. Dynamic program slicing has been applied to several programming paradigms (see Silva [9] for a survey).

The idea explored in this paper is to define and implement a novel dynamic slicing technique, which builds on top of a formal executable semantics for RSs, suitable for analyzing RS processes. Both the RS semantics and the slicing algorithm will be specified in Maude [10]: a high-level programming language and system based on the rewriting logic framework of [11]. Maude [12] supports functional, concurrent, logic, and object-oriented computations and provides equational rewriting and reasoning modulo user-defined equations and algebraic axioms such as associativity, commutativity, and identity. Maude is well-suited to specify software systems. In fact, a Maude specification combines a set of rewrite rules R, which specifies the state transitions of a system, with an equational theory $\mathcal{E}$ that specifies system states as terms of an algebraic data type. The equations in $\mathcal{E}$ are implicitly oriented from left to right as rewrite rules and operationally used as simplification rules to perform rewriting modulo equations and axioms.

1.3. Contribution

The main contributions of this paper are listed below.

1. We provide an elegant and concise Maude specification that rigorously formalizes an executable semantics for reaction systems. We also prove correctness of our implementation with regard to the usual set-theoretic reaction system characterization.

2. We apply the previous formalization to a biological model for gene regulation and we take advantage of the rich Maude development environment [12] to support and simplify the analysis phase. More specifically, we first illustrate how the Maude built-in search facility can be used to perform reachability analysis on the model. Then, we employ the ANIMA system [13]—a visual program explorer for Maude computations that we developed in a previous work—to provide an incremental view of the evolution of a reaction system. In ANIMA, system biologists can explore the computation space of RSs in a stepwise manner by expanding state transitions one at a time, thereby focusing on selected aspects of the biological processes.

3. By enriching the reaction system semantics, we also define a forward slicing algorithm for RSs and we formally prove its correctness. Forward slicing is a powerful tool to detect (forward) causality and influence relations among the entities produced in a biological model. It shows how (parts of) an initial input affect the production of (parts of) the output and helps estimate which input data need to be modified to accomplish a change in the outcome. Similarly to [2,14], our approach considers context-independent RSs (that is, RSs where the environment only provides a set of initial entities C_0). More precisely, our forward slicing methodology requires the user to select a subset C_0' of the initial input C_0 specifying the entities to be observed. Then, it proceeds by producing a computation that encodes two reaction system processes that progress in parallel: the original process that stems from C_0 and a secondary process (the sliced one) that originates from C_0' and that includes only the entities and reactions which are related to the partial input C_0'. This way, the two processes can be easily compared to detect errors and/or causality relations.

4. By applying our forward slicing technique to a gene regulation model, we show how computation states can be drastically reduced in size, favoring the system comprehension and detection of influence relations among entities.

1.4. Organization

In Section 2, we recall the basics of RSs. In Section 3, we introduce some preliminary notions on rewriting logic and Maude. We assume some basic knowledge of term rewriting [15] and some familiarity with the Maude language. In Section 4, we define a Maude specification that provides a precise execution model for RSs and we formally prove its correspondence with regard to the set-theoretic representation of reaction systems presented in Section 2. In Section 5, the Maude specification in Section 4 is used to explore and analyze an RS that specifies a gene regulation network. In Section 6, we first define a forward slicing algorithm for RSs and prove its correctness. Then, we show how our forward slicing method works in practice using the gene regulation example in Section 5. We discuss some related work in Section 7 and future work in Section 8, together with concluding remarks.

2. Reaction Systems

Let us recall some basic notions about reaction systems that are relevant to this work. For a full discussion on this topic, it is possible to consult [1,2].

An *entity* is a generic element (e.g., molecules, ions, atoms, and other chemicals) that may be present in biochemical reactions. Let S be a finite set of entities (the *background set*). A *reaction* in S is a triple (R, I, P), where $R, I,$ and P are finite sets of entities such that $R, I, P \subseteq S$ and $R \cap I = \emptyset$. The sets R, I, P, respectively, model sets of *reactants*, *inhibitors*, and *products* of the reaction. By $rac(S)$, we denote the set of all reactions on S.

Informally, a reaction can take place whenever all of its reactants are present in a given state while none of its inhibitors are present. If this happens, the reaction is enabled and creates its products. More formally, given a set $T \subseteq S$, a reaction $a = (R_a, I_a, P_a)$ is *enabled* by T, denoted $en_a(T)$, if and only if $T \subseteq R_a$ and $T \cap I_a = \emptyset$. The *result* (or *outcome*) of a on T is defined by

$$res_a(T) = \begin{cases} P_a & \text{if } en_a(T) \\ \emptyset & \text{otherwise} \end{cases}$$

We assume that for each reaction (R, I, P), R, I, and P are non-empty. This guarantees that, for each reaction a, there is always at least one product of a that is produced by at least one reactant of a. Note that, if an entity in a state is not sustained by at least one reaction which produces it, then it will disappear in the following state. This behavior reflects one of the core assumption of RSs, namely, the *no permanency principle* [2]. In other words, an entity from a current state vanishes unless it is produced by the system.

Given a set of reactions $A \subseteq rac(S)$ and a set of entities $T \subseteq S$, the *result* (or outcome) of A on T is $\bigcup_{a \in A} res_a(T)$. This definition assumes that there is no conflict on the available resources. Indeed, RSs enjoy the *threshold supply principle* [2]: either an entity is present in the state and there is "enough" of it, or an entity is not present. To put it differently, two or more reactions that are enabled by T always generate their products, even if they share some entities, because there is always a sufficient amount of reactants in T to activate all the enabled reactions. Note that this assumption implies a major difference with standard models of concurrent systems, such as Petri nets.

Definition 1. *A reaction system (RS, in short) is a pair $\mathcal{A} = (S, A)$ such that S is a finite set of entities and $A \subseteq rac(S)$.*

The dynamics of reaction systems is defined through the notion of interactive process, that is, a process that may react to external stimuli. More formally,

Definition 2. *Let $\mathcal{A} = (S, A)$ be a reaction system and let $n \geq 0$ be an integer, an n-step interactive process in $\mathcal{A}$ is a pair $\pi = (\gamma, \delta)$ such that $\gamma = C_0, C_1, \ldots, C_n$ and $\delta = D_0, D_1, \ldots, D_n$ where*

1. $C_i, D_i \subseteq S$, for $i = 0, \ldots, n$;
2. $D_0 = \varnothing$;
3. $D_i = res_A(D_{i-1} \cup C_{i-1})$, for $i = 1, \ldots, n$.

The sequence γ is called *context* sequence and represents the environment that may interact with the RS, while δ is the *result* sequence which is completely determined by reactions in A and contexts in γ.

It is worth noting that Definition 2 models reaction systems as open systems that may react to external inputs which are provided by the context sequence. Therefore, distinct context sequences may lead to distinct outcomes for the same RS $\mathcal{A}$, as illustrated in the following example.

Example 1. *Let $\mathcal{A} = (S, A)$ be a RS such that $S = \{a, b, c\}$ and $A = \{a_1, a_2\}$ where*

$$a_1 = (\{a, b\}, \{c\}, \{a\}),$$
$$a_2 = (\{a\}, \{b\}, \{c\}).$$

Consider a three-step interactive process $\pi = (\gamma, \delta)$ where $\gamma = C_0, C_1, C_2$ with $C_0 = \{a, b\}$, $C_1 = \{c\}$ and $C_2 = \{b\}$. Then, $\delta = D_0, D_1, D_2 = \varnothing, \{a\}, \{c\}$.

Now, consider the three-step interactive process $\pi' = (\gamma', \delta')$, where the context sequence $\gamma' = C_0', C_1', C_2' = \{a, b\}, \{b, c\}, \{b\}$ is a slight mutation of γ. Then, we obtain $\delta' = D_0', D_1', D_2' = \varnothing, \{a\}, \varnothing$.

Given an interactive process $\pi = (\gamma, \delta)$ such that $\gamma = C_0, C_1, \ldots, C_n$ and $\delta = D_0, D_1, \ldots, D_n$, we say that π is *a context-independent process* if $C_i \subseteq D_i$, for $i = 1, \ldots, n$. In this special case, the result sequence δ is completely determined by the initial context C_0. More specifically, D_1 only depends on C_0, and D_{i+1} only depends on D_i, with $i = 1, \ldots, (n-1)$. For this reason, we often denote a context-independent process as $\pi = (C_0, \delta)$.

3. Software Systems in Maude

The Maude system [10,12] is a formal language and tool set based on rewriting logic. Rewriting logic [11] is a logical formalism that is based on two simple ideas: states of a system are represented as terms of an algebraic data type, specified in a Maude equational theory, and the behavior of a system is given by local transitions between states described by rewrite rules.

An equational theory specifies data types by declaring operators whose composition builds complex data structures from simpler ones. Maude equational theories are order-sorted: this means that operators have a typed structure over a set of sorts Γ, and there exists a (possibly empty) poset $(\Gamma, <)$ that models subsort relations between the sorts in Γ. An operator `f` in an equational theory is specified in prefix notation by using the syntax op `f` : $s_1 \ldots s_n \to s$, $n \geq 0$, where $s_1 \ldots s_n$ denotes the sequence of argument sorts (i.e., the arity of `f`), and `s` is the sort of the return value (the keyword `ops` can be used to declare multiple operators with the same type structure: e.g. ops `f g h` : $s_1 \to s_2$.) When the arity of `f` is the empty sequence, `f` is called constant. An operator can be specified in mixfix notation by using underscores as place holders for the input arguments (e.g., $\_ \otimes \_$: $s_1 s_2 \to s$). Binary operators may have attached an axiom declaration that specifies any combinations of algebraic laws such as associativity (`assoc`), commutativity (`comm`), and identity (`id`). By `Ax`, we denote the set of all axioms attached to the operators in the equational theory. Variables are introduced together with their sort by means of the keyword `var` for a single variable declaration or the keyword `vars` for multiple declarations on a single line.

Equations in an equational theory define functions that can be used in term simplification. A term is a variable or an application of an operator to a list of terms. Assuming the equations fully define all the specified functions, each term t has a canonical form (modulo Ax), which is denoted by $t_{\downarrow_{\mathcal{E}}}$. The canonical form $t_{\downarrow_{\mathcal{E}}}$ is obtained by using the equations, oriented left to right, to rewrite t (modulo Ax) until no more rewrites are possible. Equations are specified by using the following syntax: eq l = r, where l and r are terms whose sorts belong to the same connected component of $(\Gamma, <)$. An equational theory $\mathcal{E}$ in Maude is specified by a functional module, which is delimited by keywords `fmod` and `endfm`, that contains sort, variable, operator and equation definitions.

Example 2. *The functional module of Listing 1 specifies an equational theory that models sets of entities. Line 4 defines three atomic entities via the constants* `a`, `b`, `c` *of sort* `Entity`. *Sets are terms of sort* `Entities` *whose form is* $e_1 \dots e_n$, *where each* e_i *is a term of sort* `Entity`. *Sets are built using the associative and commutative operator* `__` *with identity* `empty`, *plus the equation in Line 10 that models idempotence to allow for multiple occurrences of the same entity to be automatically simplified. The identity* `empty` *specifies the empty set. Note that, thanks to the subsort relation of Line 3, each term of sort* `Entity` *is also a term of sort* `Entities`; *hence, atomic entities can be also interpreted as singletons.*

Finally, Lines 11–12 provide an equational definition for the Boolean function `isEmpty(E)` *that verifies whether the set* `E` *is empty. Note that the attribute* `owise` *in Line 12 applies the equation in Line 12 whenever equation in Line 11 cannot be applied (that is, when* `E` *is not empty).*

Canonical forms of a term `t` *in Maude can be computed via the* `red` *command, that simplifies* `t` *using the equations and axioms in the functional module. For instance,*

```
Maude> red a b a a c .
reduce in ENTITYSET : a b a a c .
rewrites: 2 in 0ms cpu (0ms real) (2000000 rewrites/second)
result Entities: a b c
```

and

```
Maude> red isEmpty(a b a a c) .
reduce in ENTITYSET : isEmpty(a b a a c) .
rewrites: 3 in 0ms cpu (0ms real) (3000000 rewrites/second)
result Bool: false
```

Listing 1. An equational theory for modeling sets of entities.

```
1 fmod ENTITYSET is
2    sorts Entity Entities .
3    subsort Entity < Entities .
4    ops a b c : -> Entity .
5    op empty : -> Entities .
6    op __ : Entities Entities -> Entities [assoc comm id: empty] .
7    op isEmpty : Entities -> Bool .
8    var x : Entity .
9    var E : Entities .
10   eq x x = x .
11   eq isEmpty(empty) = true .
12   eq isEmpty(E) = false [owise] .
13 endfm
```

A *conditional* rewrite rule is an expression of the form `crl [lbl] l => r if C`, where `lbl` is a label that identifies the rule, `l` and `r` are terms whose sorts belong to the same connected component of $(\Gamma, <)$, and `C` is a (possibly empty) conjunction of Boolean expressions. When the condition `C` is empty, we simply write `rl [lbl] : l => r`.

A rewrite theory $\mathcal{R}$ consists of an equational theory $\mathcal{E}$ plus a set of rewrite rules R. Rewrite theories in Maude are specified by system modules (delimited by keywords `mod` and `endm`), which include a sequence of rewrite rules and one or more functional modules encoding a given equational theory. A rewrite theory is also called Maude specification.

Concurrent as well as nondeterministic software systems can be formalized through rewrite theories.

A software system, modeled as a rewrite theory, evolves by rewriting terms using *equational rewriting*, i.e., rewriting with the rewrite rules in R *modulo* the equations and axioms in $\mathcal{E}$ [11]. More precisely, (system) computations correspond to (possibly infinite) rewrite sequences $\mathtt{t_0} \xrightarrow{\mathtt{r_0}}_{R,\mathcal{E}} \mathtt{t_1} \xrightarrow{\mathtt{r_1}}_{R,\mathcal{E}} \ldots$, where $\mathtt{t} \xrightarrow{\mathtt{r}}_{R,\mathcal{E}} \mathtt{t}'$ denotes a transition (modulo $\mathcal{E}$) from term $\mathtt{t}$ to $\mathtt{t}'$ via the rewrite rule of R that is uniquely labeled with $\mathtt{r}$. The length of a finite computation $\mathcal{C}$ is the number of transitions it includes. Hence, the length of the computation $\mathtt{t_0} \xrightarrow{\mathtt{r_0}}_{R,\mathcal{E}} \mathtt{t_1} \ldots \xrightarrow{\mathtt{r_{n-1}}}_{R,\mathcal{E}} \mathtt{t_n}$ is n.

Note that each single transition $\mathtt{t} \xrightarrow{\mathtt{r}}_{R,\mathcal{E}} \mathtt{t}'$ is actually implemented as a rewrite chain $\mathtt{t} \rightarrow^*_{\mathcal{E}} (\mathtt{t}_{\downarrow_{\mathcal{E}}}) \xrightarrow{\mathtt{r}} \mathtt{t}'$, where the prefix $\mathtt{t} \rightarrow^*_{\mathcal{E}} \mathtt{t}_{\downarrow_{\mathcal{E}}}$ is an equational simplification sequence that rewrites $\mathtt{t}$ into its canonical form $\mathtt{t}_{\downarrow_{\mathcal{E}}}$ using $\mathcal{E}$; then, $\mathtt{t}_{\downarrow_{\mathcal{E}}}$ is rewritten to $\mathtt{t}'$ using a rewrite rule $\mathtt{r}$ in R. Although advisedly omitted in our notation, all rewrites in the chain (either applying $\mathtt{r}$ or any of the equations in $\mathcal{E}$) are performed *modulo* the equational axioms of $\mathcal{E}$. Terms in a computations are also called states.

Example 3. *Consider the toy rewrite theory $\mathcal{R}_{toy}$ that is encoded in the Maude system module of Listing 2. $\mathcal{R}_{toy}$ includes the equational theory $\mathcal{E}_{toy}$ of Listing 1 via the import command* `pr` *at Line 2. $\mathcal{E}_{toy}$ is used to model states as sets of entities. The rewrite theory also contains three rewrite rules that allow for the current set (state) to be modified by removing an entity (*`a`*,* `b` *or* `c`*) from it. For instance, the following one-step computation can be produced in $\mathcal{R}_{toy}$:*

$$\mathtt{b\ b\ a\ c\ a} \xrightarrow{\mathtt{del\text{-}a}}_{R_{toy},\mathcal{E}_{toy}} \mathtt{b\ c}$$

which is implemented by first simplifying the initial state `b b a c a` *into the canonical form* `a b c` *by using the equation* `eq x x = x`*, associativity and commutativity of the binary operator* `_ _`*. Then, the rewrite rule* `del-a` *is applied to the canonical form* `a b c` *to obtain* `b c`*. In symbols,* $\mathtt{b\ b\ a\ c\ a} \rightarrow^*_{\mathcal{E}} \mathtt{a\ b\ c} \xrightarrow{\mathtt{del\text{-}a}} \mathtt{b\ c}$

Listing 2. A toy rewrite theory.

```
1 mod TOY is
2   pr ENTITYSET .
3   var S : Entities .
4   rl [del-a] : (a S) => S .
5   rl [del-b] : (b S) => S .
6   rl [del-c] : (c S) => S .
7 endm
```

The transition space of all computations in $\mathcal{R}$ from the initial state $\mathtt{t_0}$ can be represented as a *computation tree* $T_{\mathcal{R}}(\mathtt{t_0})$ whose branches specify all of the computations in $\mathcal{R}$ that originate from $\mathtt{t_0}$.

4. Formalizing Reaction Systems in Maude

In this section, we show how the set-theoretic representation of reaction systems of Section 2 can be encoded in Maude. More precisely, given an RS $\mathcal{A}$, we formulate a rewrite theory $\mathcal{R}_{\mathcal{A}}$ that provides an executable model for $\mathcal{A}$. We call $\mathcal{R}_{\mathcal{A}}$ the (Maude) encoding of $\mathcal{A}$. To achieve this goal, we proceed in two steps. First, we formulate an equational theory $\mathcal{E}_{\mathcal{A}}$ that implements the basic data structures and functionality which are required to model

system states. Subsequently, we introduce a set of rewrite rule $R_{\mathcal{A}}$, which acts on top of $\mathcal{E}_{\mathcal{A}}$, to specify the system behavior of $\mathcal{A}$.

For the sake of readability, this section only describes the main code snippets of our Maude encoding. For the full implementation, please visit [16].

4.1. The Equational Theory $\mathcal{E}_{\mathcal{A}}$

Given a reaction system $\mathcal{A} = (S, A)$, the equational theory $\mathcal{E}_{\mathcal{A}}$ defines the type structure of the components of $\mathcal{A}$ via the sorts and subsort relations of Listing 3.

Listing 3. The sorts and sort poset of $\mathcal{E}_{\mathcal{A}}$.

```
1 sorts Bool Entity Entities Reaction Reactions Sequence MSequence State .
2 subsort Entity < Entities < Sequence < MSequence .
3 subsort Reaction < Reactions .
```

The equational theory $\mathcal{E}_{\mathcal{A}}$ is then populated with operators and equations that specify the building blocks of $\mathcal{A} = (S, A)$ according to the sort poset of Listing 3.

Assuming that $\mathcal{E}_{\mathcal{A}}$ includes a constant s of sort `Entity` for each entity $s \in S$, Listing 4 specifies the operators required to build the following data structures: set of entities, sequences of set of entities, and multisets of sequences of set of entities.

Listing 4. Equational definition for entities.

```
1 var x : Entity .
2 vars A B S S' : Entities .
3 --- Sets of entities
4 op empty : -> Entities .
5 op __ : Entities Entities -> Entities [ assoc comm id: empty ] .
6 eq x x = x .
7 op _in_ : Entity Entities -> Bool .
8 eq x in (x A) = true .
9 eq x in A = false [ owise ] .
10 op _subset_ : Entities Entities -> Bool .
11 eq empty subset A = true .
12 eq (x A) subset B = (x in B) and (A subset B) .
13 op intersection : Entities Entities -> Entities .
14 eq intersection(A,empty) = empty .
15 eq intersection(A,B) = $intersect(A, B, empty) .
16 op $intersect : Entities Entities Entities -> Entities .
17 eq $intersect(empty, S', A) = A .
18 eq $intersect((x S), S', A) = $intersect(S, S', if x in S' then (x A) else A fi) .
19 op isDisjoint : Entities Entities -> Bool .
20 eq isDisjoint(empty,A) = true .
21 eq isDisjoint((x A), B) = if x in B then false else isDisjoint(A,B) fi .
22 --- Sequences of set of entities
23 op _,_ : Sequence Sequence -> Sequence [ assoc ] .
24 op empty : -> Sequence .
23 --- Multisets of Sequences
24 op _+_ : MSequence MSequence -> MSequence [ comm assoc ] .
```

More specifically, a set of entities is specified as already illustrated in Example 2, that is, using the associative and commutative operator `__` with identity `empty` and the idempotence equation of Line 5. Hence, a set of entities is either a term $e_0\ e_1\ \ldots e_n$, with e_i of sort `Entity`, or the constant `empty`. Furthermore, Listing 4 provides the equational definition

of some usual functions that operate over sets and whose names are self-explanatory: `in`, `subset`, `intersection`, and `isDisjoint`.

Sequences of sets of entities are specified via the associative infix operator `_,_` and the constant `empty`. Non-empty sequences are thus terms of the form $C_0, C_1, \ldots, C_m$, with C_i of sort `Entities`, while `empty` denotes the empty sequence. Terms of sort `Sequence` will be used to define context sequences that feed reaction system processes.

Similarly, the commutative and associative operator `_+_` defines multisets of sequences of sort `MSequence`; hence, several context sequences can be encoded in a single term of sort `MSequence`. Note that the subsort relations in Listing 3 allow for a single entity `e` to be interpreted as a set of entities, a sequence of sets of entities, and a multiset of sequences of sets of entities.

Reactions and their interaction with entities are specified by Listing 5.

Listing 5. Equational definition for reactions.

```
1 vars R I P T : Entities .
2 var a : Reaction .
3 var As : Reactions .
4 --- A reaction is a triple of terms of sort Entities
5 op [_,_,_] : Entities Entities Entities -> Reaction .
6 op _;_ : Reactions Reactions -> Reactions [assoc id: empty] .
7 op empty : -> Reactions .
8 op en : Reaction Entities -> Bool .
9 eq en([R,I,P],T)  = (R subset T) and isDisjoint(I,T) .
10 op apply : Reaction Entities -> Entities .
11 eq apply([R,I,P],T) = if en([R,I,P],T) then P else empty fi .
12 op applyAll : Reactions Entities -> Entities .
13 eq applyAll(empty,T) = empty .
14 eq applyAll((a ; As),T) = apply(a,T) applyAll(As,T) .
15 op <_|_|_> : Reactions MSequence Entities -> Conf .
```

A reaction is a term of the form `[R,I,P]` of sort `Reaction`, where R, I, P are sets of entities that, respectively, identify reactants, inhibitors, and products of the specified reaction. Lists of reactions are then modeled by means of the associative infix operator `_;_` that builds terms of sort `Reactions`. An empty list of reactions is specified by the constant `empty`. Again, note that any reaction is also a list of reactions by the subsort relation `Reaction < Reactions`.

The application of a reaction (R, I, P) on a given set of entities T is defined at an equational level by means of the function `apply([R,I,P],T)` (Lines 10-11) that returns P if the function call `en([R,I,P],T)` is evaluated to be true; otherwise, it returns `empty`. This is a direct and straightforward encoding of the notion of reaction activation that we presented in Section 2, which is based on the set operators of Listing 4. For instance, the enabling predicate $en_a(T)$, with $a = (R, I, P)$, is specified by the equation in Line 9 that verifies reactant inclusion $\mathcal{R} \subseteq T$ via (`R subset T`) and absence of inhibitors $I \cap T = \varnothing$ via `isDisjoint(I,T)`. Therefore, given an RS $\mathcal{A} = (S, A)$, a reaction $a = (R, I, P) \in A$ and $T \subseteq S$, the function `apply([R,I,P],T)` computes a term representing $res_a(T)$.

The function `applyAll(A,T)` computes the set of entities $res_A(T)$ by applying all the reactions in the reaction list A to the set of entities encoded by the term T.

Example 4. *Consider the RS $\mathcal{A} = (S, A)$ of Example 1. Then, reactions in A are encoded by the term* `[ a b,c,a ] ; [ a,b,c ]` *and the function*

$$\texttt{applyAll([a b,c,a] ; [a,b,c], a b)}$$

yields the term `a`, *which is the term encoding the unique product created by the reaction set A on* $\{a, b\}$ *(in symbols, $res_A(\{a, b\}) = \{a\}$).*

Finally, the ternary operator <_|_|_> in Listing 5 defines states of a reaction system as terms of sort State with form < A | M | D >, where A is a list of reactions, M is a multiset of context sequences, and D is a set of the entities that are currently present in the system. Roughly speaking, a term of sort State provides a snapshot of the whole configuration of the reaction system at a given time instant.

4.2. The Set of Rewrite Rules $R_{\mathcal{A}}$

The dynamics of a RS $\mathcal{A} = (S, A)$ involves state transitions orchestrated by the rewrite rules process and choice of Listing 6.

Listing 6. Rewrite rules of $R_{\mathcal{A}}$.

```
1 var A : Reactions .
2 vars C D : Entities .
3 vars Cs : Sequence .
4 var M : MSequence .
5 rl [choice] :    < A | Cs + M | D > => < A | Cs | D > .
6 rl [process] : < A | C,Cs | D > => < A | Cs | applyAll(A, C D) > .
```

The process rule implements state transitions obtained by reaction applications. More precisely, given a configuration < A | C,Cs | D > where A encodes a list of reactions, (C,Cs) is a term encoding a context sequence, and D encodes the set of entities currently present in the system state, process consumes the first context C of (C,Cs) and generates a new configuration $<$ A $|$ Cs $|$ D$'$ $>$, where D$'$ encodes the set of entities obtained by applying the reactions specified by A to the set of entities $C \cup D$ encoded by the term (C D) (in symbols, D$'$ = applyAll(A, C D)).

It is worth noting that the process rule precisely captures the notion of interactive process of Definition 2, as stated by the following proposition:

Proposition 1. *Let $\mathcal{A} = (S, A)$ be an RS and let $\mathcal{R}_{\mathcal{A}}$ be the Maude encoding of $\mathcal{A}$.*

(i) *If $\pi = (\gamma, \delta)$ is an n-step interactive process in $\mathcal{A}$, $n \geq 0$, such that $\gamma = C_0, C_1, \ldots, C_n$ and $\delta = D_0, D_1, \ldots, D_n$, then there exists the computation $\mathcal{C}_{\mathcal{A}}$ of length n in $\mathcal{R}_{\mathcal{A}}$*

$$< \texttt{A} \,|\, \texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_n \,|\, \texttt{empty} > \stackrel{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_1, \ldots, \texttt{C}_n \,|\, \texttt{D}_1 >$$

$$\vdots$$

$$< \texttt{A} \,|\, \texttt{C}_{n-1} \,|\, \texttt{D}_{n-1} > \stackrel{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_n \,|\, \texttt{D}_n >$$

where $\texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_n$ is a term encoding the context sequence $C_0, C_1, \ldots, C_n$ and $D_0, D_1, \ldots, D_n$ is a term encoding the result sequence $D_0, D_1, \ldots, D_n$.

(ii) *If*

$$< \texttt{A} \,|\, \texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_n \,|\, \texttt{empty} > \stackrel{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_1, \ldots, \texttt{C}_n \,|\, \texttt{D}_1 >$$

$$\vdots$$

$$< \texttt{A} \,|\, \texttt{C}_{n-1} \,|\, \texttt{D}_{n-1} > \stackrel{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_n \,|\, \texttt{D}_n >$$

is a computation of length n in $\mathcal{R}_{\mathcal{A}}$, $n \geq 0$, then there exists an n-step interactive process $\pi = (\gamma, \delta)$ in $\mathcal{A}$, with $\gamma = C_0, C_1, \ldots, C_n$, $\delta = \varnothing, D_1, \ldots, D_n$ where $\texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_n$ and $\texttt{empty}, \texttt{D}_1, \ldots, \texttt{D}_n$ are the terms encoding the sequences $C_0, C_1, \ldots, C_n$ and $\varnothing, D_1, \ldots, D_n$.

Proof. *(i)* Let $\mathcal{A} = (S, A)$ be an RS and let $\mathcal{R}_{\mathcal{A}}$ be the Maude encoding of $\mathcal{A}$. Let $\pi = (\gamma, \delta)$ be an *n*-step interactive process in $\mathcal{A}$ such that $n \geq 0$, $\gamma = C_0, C_1, \ldots, C_n$ and $\delta = D_0, D_1, \ldots, D_n$. To prove the proposition, we proceed by induction on *n*.

$n = 0$. This case is straightforward. We have $\pi = (\gamma, \delta) = (C_0, D_0) = (C_0, \varnothing)$ and the initial state $< \texttt{A} \,|\, \texttt{C}_0 \,|\, \texttt{empty} >$, where $\texttt{C}_0$ is the term encoding C_0 and $\texttt{empty}$ the term encoding D_0. Now, it suffices to consider the computation in $\mathcal{R}_{\mathcal{A}}$ of length 0 that originates from $< \texttt{A} \,|\, \texttt{C}_0 \,|\, \texttt{empty} >$.

$n > 0$. Let $\pi = (\gamma, \delta)$ with $\gamma = C_0, \ldots, C_n$ and $\delta = D_0, \ldots, D_n = \varnothing, \ldots, D_n$ be an n-step interactive process in $\mathcal{A}$. We consider the $(n-1)$-step interactive process $\pi' = (\gamma', \delta')$ with $\gamma' = C_0, \ldots, C_{n-1}$ and $\delta' = D_0, \ldots, D_n = \varnothing, \ldots, D_{n-1}$. By induction hypothesis, there exists a computation $\mathcal{C}'_{\mathcal{A}}$ of length $(n-1)$ in $\mathcal{R}_{\mathcal{A}}$

$$\mathcal{C}'_{\mathcal{A}} = < \texttt{A} \,|\, \texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_{n-1} \,|\, \texttt{empty} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_1, \ldots, \texttt{C}_{n-1} \,|\, \texttt{D}_1 >$$

$$\vdots$$

$$< \texttt{A} \,|\, \texttt{C}_{n-2}, \texttt{C}_{n-1} \,|\, \texttt{D}_{n-2} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_{n-1} \,|\, \texttt{D}_{n-1} >$$

where $\texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_{n-1}$ is a term encoding the context sequence $C_0, C_1, \ldots, C_{n-1}$ and $\texttt{D}_0, \texttt{D}_1, \ldots, \texttt{D}_{n-1}$ is a term encoding the result sequence $D_0, D_1, \ldots, D_{n-1}$. Now, observe that, for any state transition

$$< \texttt{A} \,|\, \texttt{C}_\texttt{i}, \texttt{C}_{\texttt{i}+1}, \ldots, \texttt{C}_{n-1} \,|\, \texttt{D}_\texttt{i} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_{\texttt{i}+1}, \ldots, \texttt{C}_{n-1} \,|\, \texttt{D}_{\texttt{i}+1} >$$

that occurs in $\mathcal{C}'_{\mathcal{A}}$, there also exists the state transition

$$< \texttt{A} \,|\, \texttt{C}_\texttt{i}, \texttt{C}_{\texttt{i}+1}, \ldots, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{D}_\texttt{i} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_{\texttt{i}+1}, \ldots, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{D}_{\texttt{i}+1} >$$

where $\texttt{C}_n$ is the term encoding the n-th context C_n of γ. This is because the application of the $\texttt{process}$ rule only consumes the head of the context sequence $\texttt{C}_\texttt{i}, \ldots, \texttt{C}_{n-1}$; thus, adding the context $\texttt{C}_n$ at the end of the current context sequence $\texttt{C}_\texttt{i}, \ldots, \texttt{C}_{n-1}$ does not alter (or disable) the rule application. Thus, we also have the following computation $\mathcal{C}''_{\mathcal{A}}$ of length $n-1$:

$$\mathcal{C}''_{\mathcal{A}} = < \texttt{A} \,|\, \texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{empty} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_1, \ldots, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{D}_1 >$$

$$\vdots$$

$$< \texttt{A} \,|\, \texttt{C}_{n-2}, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{D}_{n-2} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{D}_{n-1} >$$

where $\texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_{n-1}, \texttt{C}_n$ is a term encoding the context sequence $C_0, C_1, \ldots, C_{n-1}, C_n$ and $\texttt{D}_0, \texttt{D}_1, \ldots, \texttt{D}_{n-1}$ is a term encoding the result sequence $D_0, D_1, \ldots, D_{n-1}$.

To conclude the proof, just note that there exists the state transition

$$c_n = < \texttt{A} \,|\, \texttt{C}_{n-1}, \texttt{C}_n \,|\, \texttt{D}_{n-1} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_n \,|\, \texttt{D}_n >$$

since it is immediate to observe that the function call $\texttt{applyAll}(\texttt{A}, \texttt{C}_{n-1} \, \texttt{D}_{n-1})$ yields $\texttt{D}_n$, which is the encoding of the result set D_n in δ. Therefore, by concatenating the computation $\mathcal{C}''_{\mathcal{A}}$ with the state transition c_n, we obtain a computation $\mathcal{C}_{\mathcal{A}}$ such that

$$\mathcal{C}_{\mathcal{A}} = < \texttt{A} \,|\, \texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_n \,|\, \texttt{empty} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_1, \ldots, \texttt{C}_n \,|\, \texttt{D}_1 >$$

$$\vdots$$

$$< \texttt{A} \,|\, \texttt{C}_{n-1} \,|\, \texttt{D}_{n-1} > \overset{\texttt{process}}{\longrightarrow}_{R_{\mathcal{A}}, \mathcal{E}_{\mathcal{A}}} < \texttt{A} \,|\, \texttt{C}_n \,|\, \texttt{D}_n >$$

where $\texttt{C}_0, \texttt{C}_1, \ldots, \texttt{C}_n$ is a term encoding the context sequence $C_0, C_1, \ldots, C_n$ and $\texttt{D}_0, \texttt{D}_1, \ldots, \texttt{D}_n$ is a term encoding the result sequence $D_0, D_1, \ldots, D_n$.

(ii) Proof of *(ii)* is similar to *(i)* and proceeds by induction on the length n of the computation

$$< \mathtt{A} \,|\, \mathtt{C_0}, \mathtt{C_1}, \ldots, \mathtt{C_n} \,|\, \mathtt{empty} > \overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}} < \mathtt{A} \,|\, \mathtt{C_1}, \ldots, \mathtt{C_n} \,|\, \mathtt{D_1} >$$

$$\vdots$$

$$< \mathtt{A} \,|\, \mathtt{C_{n-1}} \,|\, \mathtt{D_{n-1}} > \overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}} < \mathtt{A} \,|\, \mathtt{C_n} \,|\, \mathtt{D_n} >$$

$\square$

Note that the reaction set A and the context sequence γ of an RS $\mathcal{A}$ totally determine any interactive process on $\mathcal{A}$. In other words, when A and γ are fixed, there is only one computation of length n in $\mathcal{R}_\mathcal{A}$ that specifies an n-step interactive process on $\mathcal{A}$.

The `choice` rule introduces non-determinism into reaction systems by allowing for multiple context sequences to be specified within a single state. By doing so, one has the possibility to feed a reaction system with distinct external inputs as already illustrated in Example 1. More specifically, given a state `< A | Cs + M | D >`, the `choice` rule non-deterministically selects a context sequence `Cs` within the multiset `Cs + M`, thereby yielding a new configuration `< A | Cs | D >` which only contains the selected context sequence `Cs`. Therefore, in this more general scenario, a computation in $\mathcal{R}_\mathcal{A}$ has typically the following form:

$$\mathcal{C}_\mathcal{A} = < \mathtt{A} \,|\, (\mathtt{C_{0_0}} \ldots \mathtt{C_{n_0}}) + \ldots + (\mathtt{C_{0_m}} \ldots \mathtt{C_{n_m}}) \,|\, \mathtt{empty} > \overset{\mathtt{choice}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}} < \mathtt{A} \,|\, (\mathtt{C_{0_i}} \ldots \mathtt{C_{n_i}}) \,|\, \mathtt{empty} >$$

$$\overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}} < \mathtt{A} \,|\, (\mathtt{C_{1_i}} \ldots \mathtt{C_{n_i}}) \,|\, \mathtt{D_{1_i}} >$$

$$\vdots$$

$$\overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}} < \mathtt{A} \,|\, \mathtt{C_{n_i}} \,|\, \mathtt{D_{n_i}} >$$

with integers $m \geq 0$, $i \in \{0, \ldots, m\}$, $n_i \geq 0$. Intuitively, the first state transition uses the `choice` rule to select a context sequence $(\mathtt{C_{0_i}} \ldots \mathtt{C_{n_i}})$ among the ones that are available in the initial state, and then the reaction system evolves, using its reactions and the selected context sequence, by repeatedly applying the `process` rule.

Example 5. *Consider the RS $\mathcal{A} = (A, S)$ of Example 1. Let $\mathcal{R}_\mathcal{A}$ be the Maude encoding of $\mathcal{A}$.*

Then, the two interactive processes π and π' of Example 1 are, respectively, simulated by the following two computations in $\mathcal{R}_\mathcal{A}$:

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, (\mathtt{a\,b, c, b}) + (\mathtt{a\,b, b\,c, b}) \,|\, \mathtt{empty} > \overset{\mathtt{choice}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}}$$

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, (\mathtt{a\,b, c, b}) \,|\, \mathtt{empty} > \overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}}$$

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, (\mathtt{c, b}) \,|\, \mathtt{a} > \overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}}$$

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, \mathtt{b} \,|\, \mathtt{c} >$$

and

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, (\mathtt{a\,b, c, b}) + (\mathtt{a\,b, b\,c, b}) \,|\, \mathtt{empty} > \overset{\mathtt{choice}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}}$$

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, (\mathtt{a\,b, b\,c, b}) \,|\, \mathtt{empty} > \overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}}$$

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, (\mathtt{b\,c, b}) \,|\, \mathtt{a} > \overset{\mathtt{process}}{\longrightarrow}_{R_\mathcal{A}, \mathcal{E}_\mathcal{A}}$$

$$< [\mathtt{a\,b, c, a}]; [\mathtt{a, b, c}] \,|\, \mathtt{b} \,|\, \mathtt{empty} >$$

5. Exploring Computations in a Reaction System

The Maude formalization of Section 4 provides a generic, abstract framework for the execution and exploration of computations in a reaction system. Arbitrary reaction systems as well as arbitrary external context sequences can be plugged into the framework and then analyzed using the tools available in the Maude ecosystem. In the following, we illustrate such Maude capabilities using a more realistic biological example.

Example 6. *Let us consider the RS $\mathcal{A}_{GR} = (S_{GR}, A_{GR})$ of [17] that models a network for gene regulation. Roughly speaking, these networks represent the interactions among genes regulating*

the activation of specific cell functions. The considered RS specifies a fragment of the network for controlling the process of differentiation of T helper lymphocytes, which play a fundamental role in the immune system. Entities in S_{GR} represent genes, gene expression levels, and other functional molecules. High and medium expression levels for the same gene are specified by two distinct constants, e.g., tbeth *and* tbetm *represent high expression level and medium expression level for the tbet gene, respectively. Listing 7 illustrates the list of the 32 reactions that specifies $\mathcal{A}_{GR}$.*

Listing 7. Reactions for controlling the differentiation of T helper (Th) lymphocytes.

```
op GR : -> Reactions [ctor] .
eq GR = [ stat4 , irak s4ir tbeth, ifngammam ] ;
        [ tbetm, irak s4ir, ifngammam ] ;
        [ tbetm, s4ir tbeth, ifngammam ] ;
        [ stat4 tbetm , s4ir , ifngammam  ] ;
        [ stat4 tbetm, irak tbeth, ifngammam ] ;
        [ stat4 irak, empty, ifngammah ] ;
        [ tbeth, empty, ifngammah ] ;
        [ gata3, stat1h stat1m, il4 ] ;
        [ ifngammam, empty, ifngammarm ] ;
        [ ifngammah socs1, empty, ifngammarm ] ;
        [ ifngammah, socs1, ifngammarh ] ;
        [ il4, socs1, il4r ] ;
        [ il12, stat6, il12r ] ;
        [ il18, stat6, il18r ] ;
        [ ifnbeta, empty, ifnbetar ] ;
        [ ifnbetar, ifngammarh, stat1m ] ;
        [ ifngammarm, empty, stat1m ] ;
        [ ifngammarh, empty, stat1h ] ;
        [ il4r, empty, stat6 ] ;
        [ il12r, gata3, stat4 ] ;
        [ il18r, empty, irak ] ;
        [ stat1h, empty, socs1 ] ;
        [ stat1m, empty, socs1 ] ;
        [ tbeth, empty, socs1 ] ;
        [ tbetm, empty, socs1 ] ;
        [ stat6, tbeth tbetm, gata3 ] ;
        [ gata3, tbeth tbetm, gata3 ] ;
        [ tbetm, tbeth gata3 stat1h, tbetm ] ;
        [ stat1m, tbeth gata3 stat1h, tbetm ] ;
        [ tbeth, gata3, tbeth ] ;
        [ stat1h, gata3, tbeth ] ;
        [ il12r il18r, gata3, s4ir ] .
```

Given the Maude encoding $\mathcal{R}_{\mathcal{A}_{GR}}$ of $\mathcal{A}_{GR}$, interactive processes can be directly generated in $\mathcal{R}_{\mathcal{A}_{GR}}$ using the Maude built-in command `rew`, that produces rewrite sequences (computations) in $\mathcal{R}_{\mathcal{A}_{GR}}$. More specifically, the *n*-step interactive process $\pi = (\gamma, \delta)$, with $\gamma = C_0, \ldots, C_n$ and $\delta = D_0, \ldots, D_n$ is simulated in $\mathcal{R}_{\mathcal{A}_{GR}}$ by executing the command `rew [n]` $<$ `GR` $\mid$ $C_0, \ldots, C_n$ $\mid$ `empty` $>$, where `GR` specifies the reactions in Listing 7.

Example 7. *Consider the following initial state*

init = $<$ GR $\mid$ ifngammah, empty, empty, stat1h il4, empty, empty, empty $\mid$ empty $>$

where the external input is provided by the first context ifngammah *and the fourth context* stat1h il4*, while the remaining contexts are empty. Then, the command*

```
Maude> rew [7] init .
```

```
rewrite [7] in RS : < GR | ifngammah,empty,empty,stat1h il4,empty,empty,empty |
                                  empty > .
rewrites: 1626 in 0ms cpu (0ms real) (8603174 rewrites/second)
result State: < [stat4,irak s4ir tbeth,ifngammam] ;
                 [tbetm,irak s4ir,ifngammam]
                  ...
                 [il12r il18r, gata3,s4ir] |
                                   empty |
                 tbeth stat1m ifngammarm socs1 ifngammah >
```

generates a computation of length 7 that models a seven-step interactive process, whose final state contains the entities tbeth, stat1m, ifngammarm, socs1, ifngammah. In particular, we can observe that the provided context sequence enforces the presence of the tbet gene (entity tbeth) at step 7.

The Maude system is also endowed with a search facility that allows one to explore (following a breadth-first strategy) the reachable state space originating from an initial state. Reachability queries can be specified via the `search` command by means of the following syntax:

$$\texttt{search } [\texttt{n,m}] \texttt{ st } =>* \texttt{ sp such that Cond} \tag{1}$$

where n, m are, respectively, (optional) upper bounds on the number of solutions to be found and the depth of the search, st is an initial state, sp is a state pattern that models the form of the terms that must be reached, cond is an optional Boolean condition that must be satisfied by the reached states. When there is no condition, the syntax is simplified in `search [n,m] st =>* sp`. A solution of the reachability query (1) is any computation from the initial state st to a state st′ that matches the pattern sp and meets the condition Cond.

Example 8. *Consider the initial state* `init` *of Example 7 whose initial context includes* `ifngammah`. *The following reachability query*

$$\texttt{search } [1,7] \texttt{ init } =>* < \texttt{Rs : Reactions | Cs : Sequence | tbetm T:Entities} >$$

verifies whether there exists a computation (up to length 7) that starts from `init` *and reaches a state that contains the entity* `tbetm` *in its products. The outcome of the query is the following:*

```
No solution.
states: 7  rewrites: 1625 in 0ms cpu (0ms real) (5584192 rewrites/second)
```

Since there is no solution, we can derive that we cannot reach a medium gene expression level for tbet (`tbetm`*), if we start from an initial context that includes a highly expressed gene ifngamma (*`ifngammah`*) using the context sequence encoded in* `init`.

On the contrary, from `ifngammah`*, we are always able to reach* `tbeth`*, as witnessed by the execution of the following reachability query:*

$$\texttt{search } [1,7] \texttt{ init } =>* < \texttt{Rs : Reactions | Cs : Sequence | tbeth T:Entities} >$$

whose solution is

```
Solution 1 (state 3)
states: 4  rewrites: 788 in 0ms cpu (0ms real) (4581395 rewrites/second)
Rs --> [stat4,irak s4ir tbeth,ifngammam] ;
       [tbetm,irak s4ir,ifngammam] ;
       ...
       [il12r il18r,gata3,s4ir]
Cs --> stat1h il4,empty,empty,empty
T:Entities --> socs1
```

Therefore, starting from `init`*, the system evolves to a state that contains* `tbeth` *and* `socs1` *in exactly four steps.*

Although Maude `rew` and `search` commands are two powerful tools for the execution and exploration of reaction systems, they definitely lack a user-friendly interface that facilitates the inspection of reaction system behavior. A valid solution to this problem is offered by the `ANIMA` system [13]—a visual program explorer for Maude computations that we developed in a previous work. In `ANIMA`, system biologists can explore the computation space of RSs in a stepwise manner by expanding state transitions one at a time with a simple point-and-click interface, thereby producing an incremental visual representation of the whole computation tree with regard to a given initial state. Let us see an example.

Example 9. *Consider the Maude encoding $\mathcal{R}_{\mathcal{A}_{GR}}$ of $\mathcal{A}_{GR}$ together with the following initial state:*

`init = < GR | ifngammah,empty,empty,empty,empty + ifngammam,empty,empty,empty,empty | empty >`

that includes two distinct context sequences, respectively, modeling a high gene expression level `ifngammah` *and a medium gene expression level* `ifngammam` *in their initial contexts. By feeding* `ANIMA` *with* $\mathcal{R}_{\mathcal{A}_{GR}}$ *and* `init`*, we can interactively generate the computation tree with regard to* `init` *by clicking on the node tree to be expanded. Figure 1a zooms into the first (non-deterministic) transition that applies the* `choice` *rule to select one of the two context sequences available in* `init`*, while Figure 1b illustrates a partial expansion of the two possible evolutions of the system that depend on the chosen context sequence.*

Note that the reader can fully reproduce this example by simply accessing `ANIMA` *at the* http://safe-tools.dsic.upv.es/anima*, (accessed on 18 March 2024) and selecting* `Gene-Regulation-RS` *from the list of pre-loaded examples.*
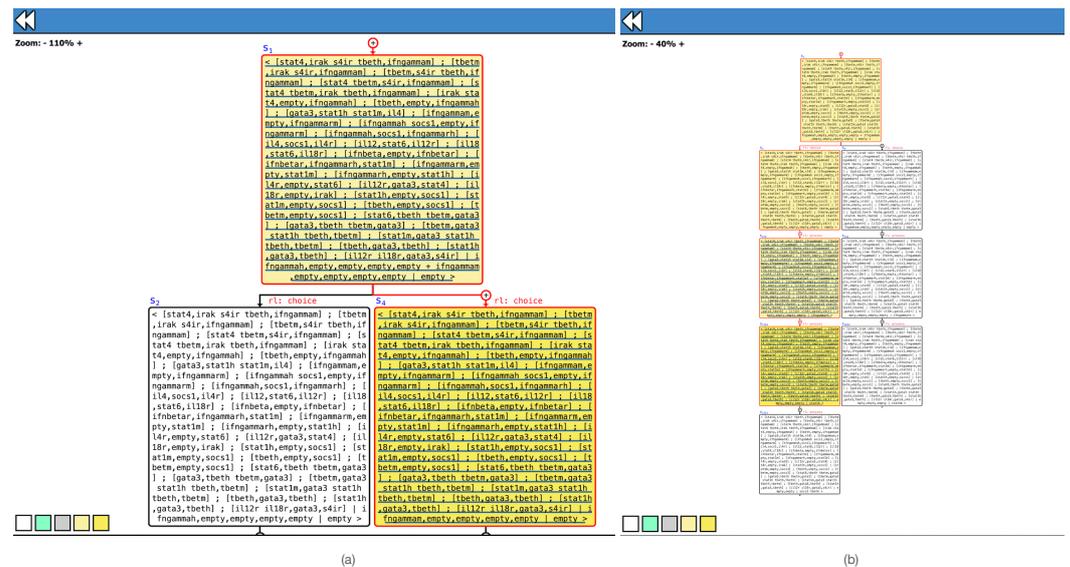
Figure 1. Visual exploration of reaction system computations in `ANIMA`.

As witnessed by Example 9, computations in a Maude specification $\mathcal{R}_{\mathcal{A}}$ can be textually large, hindering the comprehension and analysis of the reaction system behavior. The next section specifies a forward slicing technique that drastically reduces the complexity of system computations and facilitates the detection of causality relations in context-independent processes.

6. Forward Slicing of Context-Independent Processes

In this section, we present a forward slicing technique for context-independent processes. Although the generalization of the proposed slicing technique to generic interactive processes is not technically difficult, we prefer to focus on context-independent processes, since their behavior is totally defined by a single input context C_0 and having such a limited input allows one to precisely study the forward causalities of a reaction system. This

approach is also advocated by [2], where a rigorous notion of minimal *influence distance* between entities is formalized within all possible context-independent state sequences.

Given a reaction system $\mathcal{A} = (S, A)$ and a context-independent process $\pi = (C_0, \delta)$ in $\mathcal{A}$, with $\delta = D_0, \ldots, D_n$, forward slicing aims at producing a partial view of π that only depends on a selected subset C_0' of C_0 which is called *slicing criterion*. Forward slicing allows one to evaluate the impact of a set of input entities across the whole process, thereby giving answers to questions such as *does the presence (or the absence) of an entity e in the slicing criterion affect the production of the entity e' in a later stage of π?*

The forward slicing of context-independent processes can be specified by a rewrite theory $\mathcal{R}_{\mathcal{A}}^{\triangleright}$ that slightly modifies and improves the Maude encoding $\mathcal{R}_{\mathcal{A}}$ of Section 4. $\mathcal{R}_{\mathcal{A}}^{\triangleright}$ includes an extended version $\mathcal{E}_{\mathcal{A}}^{\triangleright}$ of the equational theory $\mathcal{E}_{\mathcal{A}}$ that provides a richer algebraic data type for system states, as well as a new set of rewrite rules $R_{\mathcal{A}}^{\triangleright}$ that implements an augmented reaction system semantics that models both a context-independent process and its sliced counterpart. We say that $\mathcal{R}_{\mathcal{A}}^{\triangleright}$ is the *forward slicing* encoding for $\mathcal{A}$. In the following, we present the main data structures and core functions included in $\mathcal{R}_{\mathcal{A}}^{\triangleright}$. The full specification is available at [16].

6.1. The Equational Theory $\mathcal{E}_{\mathcal{A}}^{\triangleright}$

The code snippet in Listing 8 illustrates the key modifications introduced by $\mathcal{E}_{\mathcal{A}}^{\triangleright}$, with respect to $\mathcal{E}_{\mathcal{A}}$.

Listing 8. Equational theory $\mathcal{E}_{\mathcal{A}}^{\triangleright}$: main changes with regard to $\mathcal{E}_{\mathcal{A}}$.

```
1 sort Result .
2 op {_|_} : Reactions Entities -> Result .
3 vars A A' : Reactions .
4 var Res : Result .
5 vars P P' T : Entities .
6 op fail : -> Reaction .
7 op apply : Reaction Entities -> Reaction .
8 eq apply([R,I,P],T) = if en([ R,I,P],T) then [ R,I,P ] else fail fi .
9 op applyAll+ : Reactions Entities -> Result .
10 eq applyAll+(A,T) = $applyAll(A,T,{ empty | empty }) .
11 op $applyAll : Reactions Entities Result -> Result .
12 eq $applyAll(empty,T,Res) = Res .
13 eq $applyAll(([ R,I,P ] ; A), T , { A' | P' }) =
14          $applyAll(A, T, if apply([ R,I,P ],T) =/= fail  then
15                                  {[ R,I,P ] ; A' | P P'}
16                          else
17                                  { A' | P' }
18                          fi
19                  ) .
20 op <_|_|_|_|_> : Reactions Entities Reactions Entities Reactions -> State .
21 op +<_|_> : Reactions Entities -> State .
22 op -<_|_> : Reactions Entities -> State .
```

Roughly speaking, $\mathcal{E}_{\mathcal{A}}^{\triangleright}$ extends $\mathcal{E}_{\mathcal{A}}$ into two main directions.

Firstly, the function applyAll(A,T) in $\mathcal{E}_{\mathcal{A}}$, which computes the set of entities $res_A(T)$, has been replaced by the function applyAll+(A,T) (see Lines 9–19 of Listing 8) whose resulting outcome is now a pair {A' | T'} that keeps track of both the entities T' produced by applying A on T and the sublist of reactions A' which only includes the reactions of A that are enabled by T and thus are responsible for the production of T'. Note that, to implement this extension, we also need to slightly modify the function apply([R, I, P], T) (Lines 7–8), which now uses the special constant fail to explicitly signal that the reaction [R, I, P] cannot be applied to T. Let us see an example.

Example 10. *Consider the RS $\mathcal{A}_{GR} = (S_{GR}, A_{GR})$ of Example 6, whose reaction list is specified by the term* GR*. The execution of the function call* applyAll+(GR, stat1h gata3) *yields the following outcome:*

```
Maude> red applyAll+(GR,stat1h gata3) .
reduce in FORWARDSLICER-RS : applyAll+(GR, gata3 stat1h) .
rewrites: 341 in 0ms cpu (0ms real) (341000000 rewrites/second)
result Result: {[gata3,tbeth tbetm,gata3] ; [stat1h,empty,socs1] |
gata3 socs1}
```

which shows that only two reactions of GR *(out of 32) are enabled by* stat1h *and* gata3, *thereby creating the new entities* gata3 *and* socs1.

Secondly, $\mathcal{E}_{\mathcal{A}}^{\triangleright}$ refines the notion of state, which we introduced in $\mathcal{E}_{\mathcal{A}}^{\triangleright}$, by means of three novel operators (Lines 20–22) that allow system states to have multiple forms. A system state now can be

1. A term of the form $<$ A | D | Au | D$'$ | Au$'$ $>$, where A, Au, Au$'$ are terms of sort Reactions and D and D$'$ are terms of sort Entities. Intuitively, a state of this form stores the current configuration of a context-independent process together with its sliced counterpart. More formally, given a reaction list A, D represents the entities which are currently present in the system, and Au is a list of reactions —included in A— that have been used to produce D. Similarly, D$'$ is the set of entities currently observed in the sliced counterpart that were produced by using the reactions in Au$'$.
2. A term of the form -$<$ A$'$ | P$'$ $>$, where A$'$ is a term of sort Reactions and P$'$ is a term of sort Entities. This data structure is used to extract, from the current state, the entity set P$'$ of those entities that cannot be computed from the slicing criterion C_0' since the reactions in A$'$ did not take place. In other words, this state highlights the *missing* entities, that is, those entities that can be computed from the initial context C_0 but cannot be computed from the slicing criterion C_0'.
3. A term of the form +$<$ A$'$ | P$'$ $>$, where A$'$ is a term of sort Reactions and P$'$ is a term of sort Entities. This data structure is used to extract, from the current state, the entity set P$'$ of those entities that can be only computed from the slicing criterion C_0' since the reactions in A$'$ did not take place in the original process. Put differently, this state identifies the *spurious* entities, that is, those entities that can be computed from the slicing criterion C_0' but cannot be computed from the initial context C_0. Generation of spurious entities is typically caused by the absence of one or more inhibitors in C_0' that are instead present in C_0. This fact allows for a given reaction to take place in the sliced process, but it is blocked in the original process.

6.2. The Set of Rewrite Rules $R_{\mathcal{A}}^{\triangleright}$

The forward slicing algorithm of context-independent processes is implemented by the three rewrite rules of Listing 9.

Listing 9. Rewrite rules of $R_{\mathcal{A}}^{\triangleright}$.

```
1 vars D D' D-new D-new' : Entities .
2 vars A Au Au' Au-new Au-new' : Reactions .
3 crl [process-fs] : < A | D | Au | D' | Au' > =>
4                    < A | D-new | Au-new Au | D-new' | Au-new' Au' > if
5                         { Au-new' | D-new' } := applyAll+(A,D') /\
6                         { Au-new  | D-new } := applyAll+(A,D) .
7 rl [slice+] : < A | D | Au | D' | Au' > => +< (Au' \ Au) | D' \ D > .
8 rl [slice-] : < A | D | Au | D' | Au' > => -< (Au \ Au') | D \ D' > .
```

The `process-fs` rule is the backbone of the whole forward slicing algorithm. It allows the system to transition from a state

$$< \text{A} \mid \text{D}_\text{i} \mid \text{Au}_\text{i} \mid \text{D}'_\text{i} \mid \text{Au}'_\text{i} >$$

to a state

$$< \text{A} \mid \text{D}_{\text{i}+1} \mid \text{Au}_{\text{i}+1} \mid \text{D}'_{\text{i}+1} \mid \text{Au}'_{\text{i}+1} >$$

where

- `A` is a reaction list that models the reactions in the RS $\mathcal{A} = (S, A)$;
- `D`$_{\text{i}+1}$ is a term encoding the set of entities obtained by applying the reaction list `A` on the set of entities encoded by `D`$_\text{i}$;
- `Au`$_{\text{i}+1}$ is a reaction list that contains all and only the reactions used to produce `D`$_{\text{i}+1}$;
- `D`$'_{\text{i}+1}$ is a term encoding the set of entities obtained by applying the reaction list `A` on the set of entities encoded by `D`$'_\text{i}$;
- `Au`$'_{\text{i}+1}$ is a reaction list that contains all and only the reactions used to produce `D`$'_{\text{i}+1}$;

Note that the `process-fs` rule uses the function `applyAll+` to generate the entities and reactions at step $(i + 1)$ by exploiting the information at step i.

In this scenario, all the computations of the form

$$< \text{A} \mid \text{C}_0 \mid \text{empty} \mid \text{C}'_0 \mid \text{empty} > \xrightarrow[R^\triangleright_\mathcal{A}, \mathcal{E}^\triangleright_\mathcal{A}]{\texttt{process-fs}} \dots \xrightarrow[R^\triangleright_\mathcal{A}, \mathcal{E}^\triangleright_\mathcal{A}]{\texttt{process-fs}} < \text{A} \mid \text{D}_\text{n} \mid \text{A}_\text{n} \mid \text{D}'_\text{n} \mid \text{A}'_\text{n} >$$

define two context-independent processes in $\mathcal{A}$ that evolve concurrently: $\pi = (C_0, \delta)$, with $\delta = D_0, \dots, D_n$ and $\pi' = (C'_0, \delta')$, with $\delta' = D'_0, \dots, D'_n$. When $C'_0 \subseteq C_0$, we say that π' is a *sliced version* of π. Basically, a sliced version π' of π encodes the behavior of an RS $\mathcal{A}$ on the reduced set of input entities specified by the slicing criterion C'_0.

Proposition 2. *Let $\mathcal{A} = (S, A)$ be an RS and let $\mathcal{R}^\triangleright_\mathcal{A}$ be the forward slicing encoding of $\mathcal{A}$. Let C_0, C'_0 be two sets of entities such that $C'_0 \subseteq C_0$. Let C_0 and C'_0 be two terms, respectively, encoding C_0 and C'_0. If*

$$< \text{A} \mid \text{C}_0 \mid \text{empty} \mid \text{C}'_0 \mid \text{empty} > \xrightarrow[R^\triangleright_\mathcal{A}, \mathcal{E}^\triangleright_\mathcal{A}]{\texttt{process-fs}} < \text{A} \mid \text{D}_1 \mid \text{A}_1 \mid \text{D}'_1 \mid \text{A}'_1 >$$

$$\vdots$$

$$< \text{A} \mid \text{D}_{\text{n}-1} \mid \text{A}_{\text{n}-1} \mid \text{D}'_{\text{n}-1} \mid \text{A}'_{\text{n}-1} > \xrightarrow[R^\triangleright_\mathcal{A}, \mathcal{E}^\triangleright_\mathcal{A}]{\texttt{process-fs}} < \text{A} \mid \text{D}_\text{n} \mid \text{A}_\text{n} \mid \text{D}'_\text{n} \mid \text{A}'_\text{n} >$$

is a computation of length n in $\mathcal{R}^\triangleright_\mathcal{A}$, $n \geq 0$, then there exist two n-step context-independent processes $\pi = (C_0, \delta)$, with $\delta = D_0, \dots, D_n$ and $\pi' = (C'_0, \delta')$, with $\delta' = D'_0, \dots, D'_n$ such that $\text{D}_0, \dots, \text{D}_\text{n}$ and $\text{D}'_0, \dots, \text{D}'_\text{n}$ are the terms encoding the result sequences $D_0, \dots, D_n$ and $D'_0, \dots, D'_n$. and π' is a sliced version *of π.*

Proof. The proof is by induction on the length *n* of the computation

$$\mathcal{C}_n = < \text{Rs} \mid \text{C}_0 \mid \text{empty} \mid \text{C}'_0 \mid \text{empty} > \xrightarrow[R^\triangleright_\mathcal{A}, \mathcal{E}^\triangleright_\mathcal{A}]{\texttt{process-fs}} < \text{A} \mid \text{D}_1 \mid \text{A}_1 \mid \text{D}'_1 \mid \text{A}'_1 >$$

$$\vdots$$

$$< \text{A} \mid \text{D}_{\text{n}-1} \mid \text{A}_{\text{n}-1} \mid \text{D}'_{\text{n}-1} \mid \text{A}'_{\text{n}-1} > \xrightarrow[R^\triangleright_\mathcal{A}, \mathcal{E}^\triangleright_\mathcal{A}]{\texttt{process-fs}} < \text{A} \mid \text{D}_\text{n} \mid \text{A}_\text{n} \mid \text{D}'_\text{n} \mid \text{A}'_\text{n} >$$

$n = 0$. This case is trivial. The computation $\mathcal{C}_0$ does not include any rewrite step and thus leaves the initial state $< \text{Rs} \mid \text{C}_0 \mid \text{empty} \mid \text{C}'_0 \mid \text{empty} >$ unchanged. Note that this state encodes two 0-step context-independent processes: $\pi_0 = (C_0, \delta_0)$, with $\delta_0 = D_0 = \varnothing$ and $\pi'_0 = (C'_0, \delta'_0)$, with $\delta'_0 = D'_0 = \varnothing$.

$n > 0$. By inductive hypothesis, there is a computation of length $n - 1$

$$\mathcal{C}_{n-1} = <\ \texttt{Rs}\ |\ \texttt{C}_0|\ \texttt{empty}\ |\ \texttt{C}'_0\ |\ \texttt{empty}\ > \xrightarrow[R_{\mathcal{A}}^{\triangleright},\mathcal{E}_{\mathcal{A}}^{\triangleright}]{\texttt{process-fs}} <\ \texttt{A}\ |\ \texttt{D}_1|\ \texttt{A}_1\ |\ \texttt{D}'_1\ |\ \texttt{A}'_1\ >$$

$$\vdots$$

$$<\ \texttt{A}\ |\ \texttt{D}_{n-2}|\ \texttt{A}_{n-2}\ |\ \texttt{D}'_{n-2}\ |\ \texttt{A}'_{n-2}\ > \xrightarrow[R_{\mathcal{A}}^{\triangleright},\mathcal{E}_{\mathcal{A}}^{\triangleright}]{\texttt{process-fs}} <\ \texttt{A}\ |\ \texttt{D}_{n-1}|\ \texttt{A}_{n-1}\ |\ \texttt{D}'_{n-1}\ |\ \texttt{A}'_{n-1}\ >$$

such that there exist two $(n-1)$-step context-independent processes $\pi_{n-1} = (C_0, \delta_{n-1})$, with $\delta = D_0, \dots, D_{n-1}$, and $\pi'_{n-1} = (C'_0, \delta'_{n-1})$, with $\delta' = D'_0, \dots, D'_{n-1}$, where $\texttt{D}_0, \dots, \texttt{D}_{n-1}$, and $\texttt{D}'_0, \dots, \texttt{D}'_{n-1}$ are the terms that encode the result sequences $D_0, \dots, D_{n-1}$ and $D'_0, \dots, D'_{n-1}$, respectively. Furthermore, π'_{n-1} is a *sliced version* of π_{n-1}.

Observe also that it is always possible to perform the rewrite step

$$r\ =\ <\ \texttt{A}\ |\ \texttt{D}_{n-1}|\ \texttt{A}_{n-1}\ |\ \texttt{D}'_{n-1}\ |\ \texttt{A}'_{n-1}\ > \xrightarrow[R_{\mathcal{A}}^{\triangleright},\mathcal{E}_{\mathcal{A}}^{\triangleright}]{\texttt{process-fs}} <\ \texttt{A}\ |\ \texttt{D}_n|\ \texttt{A}_n\ |\ \texttt{D}'_n\ |\ \texttt{A}'_n\ >$$

which uses the function `applyAll+` to generate $\texttt{D}_n$ and $\texttt{D}'_n$ from $\texttt{D}_{n-1}$ and $\texttt{D}'_{n-1}$. Since $\texttt{D}_{n-1}$ and $\texttt{D}'_{n-1}$ are the terms encoding the set of entities D_{n-1} and D'_{n-1}, we have that $\texttt{D}_n$ and $\texttt{D}'_n$ are the terms encoding the set of entities D_n and D'_n. Therefore, to prove the proposition, it suffices to consider the computation $\mathcal{C}_n$ of length n that concatenates the computation $\mathcal{C}_{n-1}$ with the rewrite step r. $\square$

While the `process-fs` rule formalizes system progress in $\mathcal{R}_{\mathcal{A}}^{\triangleright}$, the `slice-` and `slice+` rules extract specific pieces of information from a given computation state `st` that allow for the original process π and its sliced counterpart π' to be compared on `st`.

More precisely, given a state $\texttt{st} = <\ \texttt{A}\ |\ \texttt{D}_i\ |\ \texttt{A}_i\ |\ \texttt{D}'_i\ |\ \texttt{A}'_i\ >$, the `slice-` rule uses the set difference operator (To be more precise, $\backslash$ is an overloaded operator that can be applied to both entity sets and reaction lists.)$\backslash$ to compute the new state $-<\ (\texttt{A}_i \backslash \texttt{A}'_i)\ |\ \texttt{D}_i \backslash \texttt{D}'_i\ >$ that isolates

- The entities $(\texttt{D}_i \backslash \texttt{D}'_i)$ in `st` that can be computed in (the i-th step of) π but not in (the i-th step of) π'.
- The reactions $(\texttt{A}_i \backslash \texttt{A}'_i)$ in `st` that were enabled in π but not in π' until the i-th step.

Note that $(\texttt{A}_i \backslash \texttt{A}'_i)$ represents the reason of the missing information $(\texttt{D}_i \backslash \texttt{D}'_i)$: in other words, entities encoded by $(\texttt{D}_i \backslash \texttt{D}'_i)$ cannot be produced in the i-th step of π' because reactions $(\texttt{A}_i \backslash \texttt{A}'_i)$ were never enabled in π' up to the i-th step.

Dually, the `slice+` computes the new state $+<\ (\texttt{A}'_i \backslash \texttt{A}_i)\ |\ \texttt{D}'_i \backslash \texttt{D}_i\ >$ that isolates:

- The entities $(\texttt{D}'_i \backslash \texttt{D}_i)$ in `st` that can be computed in (the i-th step of) π' but not in (the i-th step of) π.
- The reactions $(\texttt{A}'_i \backslash \texttt{A}_i)$ in `st` that were enabled in π' but not in π until the i-th step.

Intuitively, `slice+` collects the spurious entities $(\texttt{D}'_i \backslash \texttt{D}_i)$ of `st`, that is, those entities of `st` computed in π' by using the reactions $(\texttt{A}'_i \backslash \texttt{A}_i)$ that are not enabled in π (up to the i-th step).

It is worth noting that, all the three rules `process-fs`, `slice-`, and `slice+` can be non-deterministically applied to each system state. Therefore, at each state, the system can (i) evolve via the `process-fs` rule, (ii) compute the missing entities in π' by applying `slice-`, or (iii) compute the spurious entities by applying `slice+`. Let us see an example.

Example 11. *Let $\mathcal{A}_{GR} = (S_{GR}, A_{GR})$ be the RS of Example 6. Let $C_0 = \{ifngammh, gata3\}$ be an initial context and let $C'_0 = \{ifngammah\}$ be a slicing criterion for C_0. Given the initial state*

$$\texttt{init} = <\ \texttt{GR}\ |\ \texttt{C}_0|\ \texttt{empty}\ |\ \texttt{C}'_0\ |\ \texttt{empty}\ >$$

where `GR` *is the reaction list encoding the 32 reactions in A_{GR}, $\texttt{C}_0$ and $\texttt{C}'_0$ are terms encoding C_0 and C'_0, the (fragment of the) computation tree $T_{\mathcal{R}_{\mathcal{A}_{GR}}^{\triangleright}}(\texttt{init})$ of Figure 2 can be computed in $\mathcal{R}_{\mathcal{A}_{GR}}^{\triangleright}$. Note that each tree level $\texttt{i}$ (except for the root level) contains three states:*

(i) *The current state* $\mathtt{s_i}$ *which is produced by the application of the* `process-fs` *rule on the previous state* $\mathtt{s_{(i-1)}}$;

(ii) *Two states* $\mathtt{s_{(i-1)^-}}$ *and* $\mathtt{s_{(i-1)^+}}$ *that specify the states which are computed by, respectively, applying the* `slice-` *and* `slice+` *rules to the state* $\mathtt{s_{(i-1)}}$.

Compact state views, which are produced by `slice-` *and* `slice+`, *allow one to immediately grasp what is going on in the process and its sliced counterpart.*

For instance, by inspecting all the applications of `slice+` *in the tree, we can immediately note that only state* $\mathtt{s_3}$ *introduces a spurious entity (up to level* 3); *concretely, the entity* `tbeth` *which appears in the state*

$$\mathtt{s_{3+} \; = \; +< \; [stat1h,gata3,tbeth] \; | \; tbeth \; >}$$

Hence, we can infer that the gene expression tbeth can be computed for the slicing criterion C_0' *but not for the full initial context* C_0 *in the considered tree fragment. Indeed, the reaction* `[stat1h,gata3,tbeth]`, *which occurs in* $\mathtt{s_{3+}}$, *is enabled in the sliced process* π' *but not in the original one* π. *Since gata3 belongs to* C_0 *but not to* C_0', *we can safely say that gata3 blocks the creation of tbeth in* π.

Also, by looking at the applications of the `slice-` *rule, we can easily detect the information that the slicing criterion* C_0' *cannot produce for a given computation state. For example, the state* $\mathtt{s_{3-}}$ *tells us that gata3, il4r, and stat6 are missing entities that cannot be created in the sliced process* π' *that reaches state* $\mathtt{s_3}$. *This is due to the fact that reactions* `[gata3,tbeth tbetm,gata3]`, `[il4r,empty,stat6]`, *and* `[il4,socs1,il4r]` *take place only in* π *but not in* π'.

Finally, note that the `ANIMA` *system, which we presented in Section 5, can also be used to interactively execute the proposed forward slicing algorithm, allowing system biologists to visually explore system states and their associated compact views in a more user-friendly and incremental way. In this regard, Figure 3 illustrates the generation of the computation tree of Figure 2 within* `ANIMA`. *To reproduce this experiment, the reader can access* `ANIMA` *at the* http://safe-tools.dsic.upv.es/anima *(accessed on 18 March 2024) and select* Forward Slicing of Gene-Regulation-RS *from the list of pre-loaded examples.*
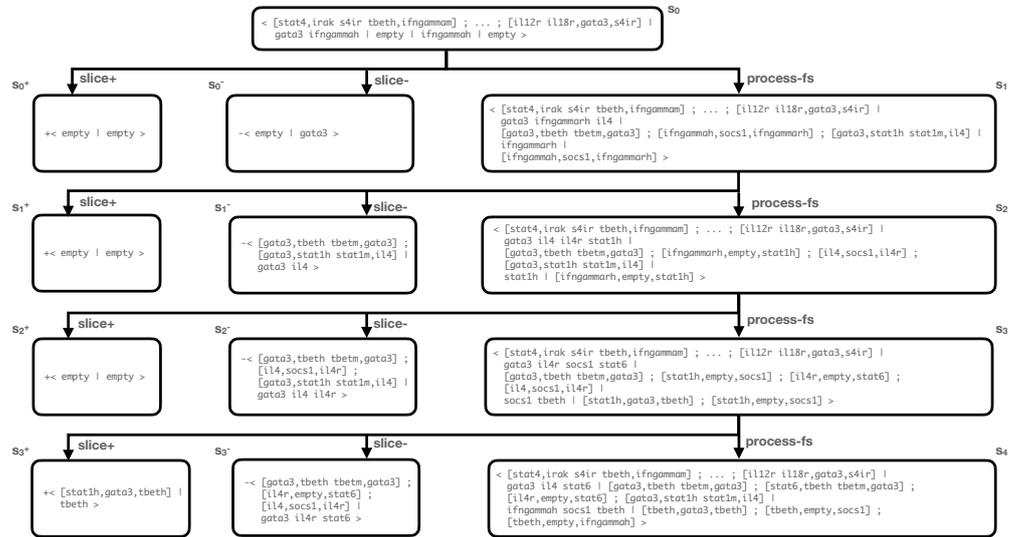

Figure 2. Fragment of the computation tree $T_{\mathcal{R}^{\triangleright}_{\mathcal{A}_{GR}}}(\mathtt{init})$.
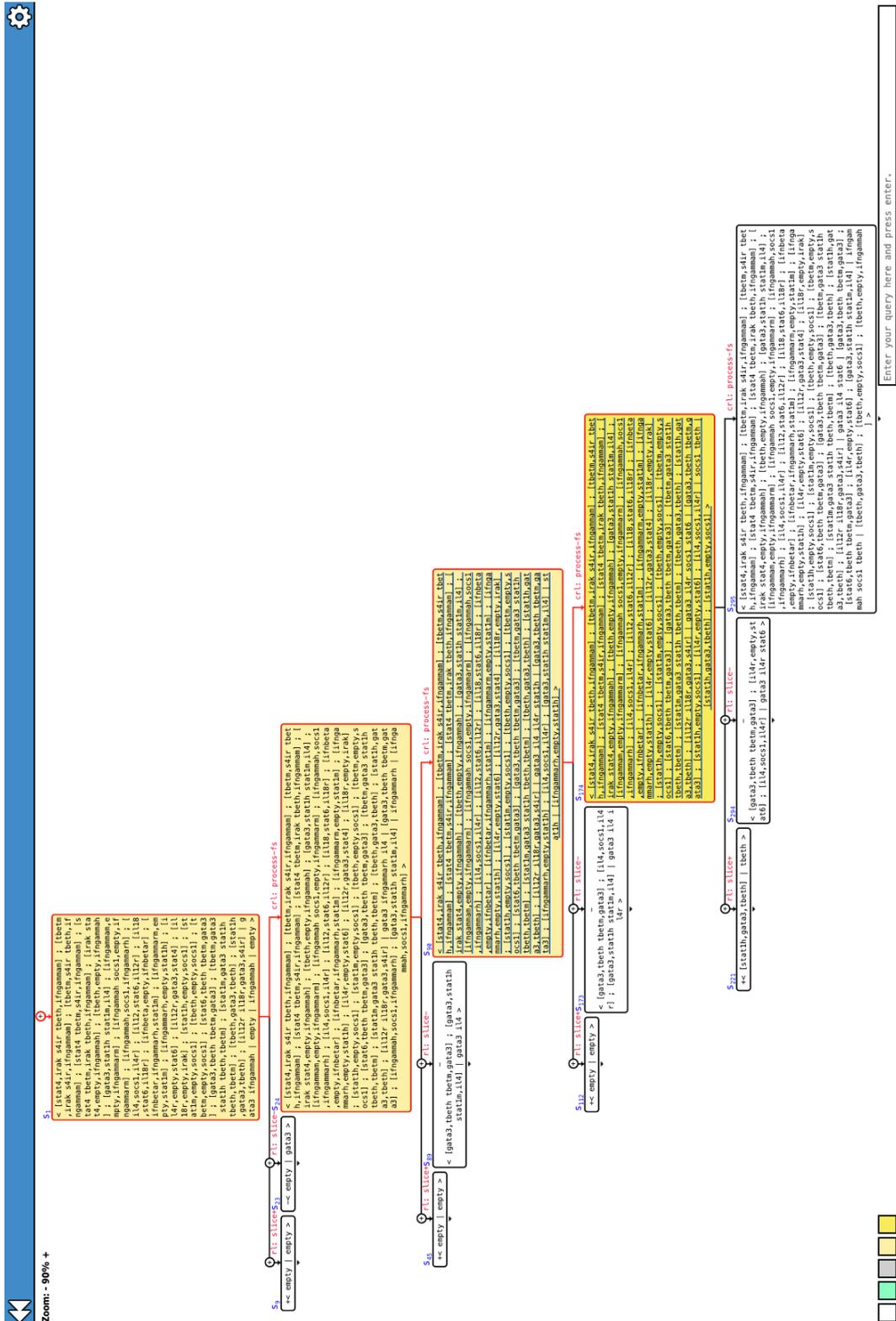
Figure 3. Forward slicing of context-independent processes in `ANIMA`

7. Related Work

Several tools are already available to simulate RSs or to verify that certain properties are met. In [18,19], some authors developed BioReSolve [20]: a PROLOG interpreter for reaction system analysis. The verification capabilities integrated into BioReSolve have been crafted from scratch. In contrast, our approach offers a rewriting-based modeling of RSs,

leveraging the extensive functionalities of the Maude environment, which provides multiple built-in functions and third-party tools for program analysis. Other implementations of RSs include `brsim` [21], a Basic Reaction System Simulator written in Haskell and distributed under the terms of GNU GPLv3 license [22] whose online version `WEBRSIM` makes all the features of `brsim` available through a friendly web interface [23]; `HERESY` [24], a GPU-based Highly Efficient REaction SYstem simulator, that exploits the large number of computational units inside GPUs to boost performance [25]; `cl-rs` [26], an optimised Common Lisp simulator for RSs presented in [27].

Model checking has been deeply studied in the context of reaction systems. [28] defines *rsCTL*, a temporal logic for reaction systems. The logic is interpreted over the models for the context-restricted reaction systems that generalize standard reaction systems by controlling context sequences. In [29,30], a variant of Linear Time Temporal Logic that is interpreted over models of reaction systems with discrete concentrations is presented: the approach adopts a suitable encoding in SMT together with bounded model checking for the formal verification of temporal properties over RSs. The verification technique has been implemented into the *ReactICS* system [31]. A more theoretical work that investigates model checking of reaction systems through computational complexity lenses is also presented in [32]. This work provides several complexity results for decision procedures related to properties of central interest in biomodeling (e.g., mass conservation, steady states, stationary processes).

We believe that model-checking and (trace and program) slicing may be successfully combined together to improve the analysis and comprehension of counter-examples generated by model-checkers when properties of interest are refuted.

Notably, the Maude language has already been successfully used in the analysis of biological systems. For instance, Pathway Logic [33] is a symbolic approach to the modeling and analysis of biological systems that is implemented in Maude. This logical framework allow metabolic pathways to be simulated and formally verified.

Dynamic program slicing has been applied previously to other programming paradigms, such as imperative programming [8], functional programming [34], and term rewriting and its extensions [35,36]. None of these approaches are suitable for RSs, which have a completely different computation behavior based on the non-monotonic enabling mechanism of reactions. In [37], we extended BioReSolve with a slicing algorithm which proceeds backwards. Some entities in the last state of a computation of an RS are selected and then the algorithm proceeds backwards, simplifying the computation and leaving only the entities which are essential to derive the selected ones. Unfortunately, this way we cannot know in advance which entities of the initial state of the computation will be left in the backward sliced computation. In this paper, we define a dual forward slicing algorithm in which we observe some entities in the initial state and we proceed forward. The two (backward and forward) slicing methodologies clearly derive different information as they focus on input information from different states and apply different algorithms. While the forward slicing results in a form of impact analysis that identifies the scope and potential consequences of changing the program input, backward slicing allows for provenance analysis to be performed in search of the origins of the selected entities.

8. Conclusions and Future Work

In this paper, we have defined the first implementation of reaction systems in Maude, a language based on rewriting logic, which has a very well-developed environment with tools for program analysis and verification. Our Maude implementation provides an interpreter for RSs that supports exploration capabilities via Maude built-in commands as well as third-party systems such as the ANIMA system. Also, we have defined an algorithm for forward slicing of context-independent processes. Our implementation of forward slicing gives a parallel execution of the standard and the sliced computation, so that users can compare each state of the original computation with the corresponding sliced one, and obtain useful information to analyze and possibly debug RS processes. To evaluate the

feasibility and usefulness of the proposed approach, our forward slicing algorithm has been used to analyze an example of a gene regulatory network.

As future work, we want to study the combination of the forward slicing algorithm with the backward one which we have defined in [37]. We believe that this integration can improve the quality of the analysis, allowing for more erroneous/unexpected behaviours to be detected. We also want to extend the forward slicing algorithm to a language for concurrent multiparty communications [38], and investigate possible extensions that exploit static analysis techniques [39–41] as well as dynamic verification methodologies [42,43].

Author Contributions: Methodology, D.B., L.B. and M.F.; Investigation, D.B., L.B. and M.F.; Writing – original draft, D.B., L.B. and M.F.; Writing – review & editing, D.B., L.B. and M.F. All authors have read and agreed to the published version of the manuscript.

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Ehrenfeucht, A.; Rozenberg, G. Reaction Systems. *Fundam. Informaticae* **2007**, *75*, 263–280.
2. Brijder, R.; Ehrenfeucht, A.; Main, M.G.; Rozenberg, G. A Tour of Reaction Systems. *Int. J. Found. Comput. Sci.* **2011**, *22*, 1499–1517. [CrossRef]
3. Azimi, S.; Iancu, B.; Petre, I. Reaction System Models for the Heat Shock Response. *Fundam. Informaticae* **2014**, *131*, 299–312. [CrossRef]
4. Corolli, L.; Maj, C.; Marini, F.; Besozzi, D.; Mauri, G. An excursion in reaction systems: From computer science to biology. *Theor. Comput. Sci.* **2012**, *454*, 95–108. [CrossRef]
5. Azimi, S. Steady States of Constrained Reaction Systems. *Theor. Comput. Sci.* **2017**, *701*, 20–26. [CrossRef]
6. Okubo, F.; Yokomori, T. The computational capability of chemical reaction automata. *Nat. Comput.* **2016**, *15*, 215–224. [CrossRef]
7. Weiser, M. Program slicing. *IEEE Trans. Softw. Eng.* **1984**, *10*, 352–357. [CrossRef]
8. Korel, B.; Laski, J. Dynamic Program Slicing. *Inf. Process. Lett.* **1988**, *29*, 155–163. [CrossRef]
9. Silva, J. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.* **2012**, *44*, 12:1–12:41. [CrossRef]
10. Durán, F.; Eker, S.; Escobar, S.; Martí-Oliet, N.; Meseguer, J.; Rubio, R.; Talcott, C.L. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.* **2020**, *110*, 100497. [CrossRef]
11. Meseguer, J. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theor. Comput. Sci.* **1992**, *96*, 73–155. [CrossRef]
12. Clavel, M.; Durán, F.; Eker, S.; Escobar, S.; Lincoln, P.; Martí-Oliet, N.; Meseguer, J.; Rubio, R.; Talcott, C. *Maude Manual (Version 3.2.1)*; Technical Report; SRI International Computer Science Laboratory , 2022. Available online: https://maude.lcc.uma.es/maude321-manual-html/maude-manual.html (accessed on 14 March 2024).
13. The Anima Website. 2015. Available online: http://safe-tools.dsic.upv.es/anima (accessed on 14 March 2024).
14. Brijder, R.; Ehrenfeucht, A.; Rozenberg, G. A Note on Causalities in Reaction Systems. *Electron. Commun. Easst* **2010**, *10*.
15. TeReSe. *Term Rewriting Systems*; Cambridge University Press: Cambridge, UK, 2003.
16. The RS-MAUDE System. 2024. Available online: https://github.com/DemisGIT/RSMaude (accessed on 14 March 2024).
17. Barbuti, R.; Gori, R.; Milazzo, P. Encoding Boolean Networks into Reaction Dystems for Investigating Causal Dependencies in Gene Regulation. *Theor. Comput. Sci.* **2021**, *881*, 3–24. [CrossRef]
18. Brodo, L.; Bruni, R.; Falaschi, M. Enhancing Reaction Systems: A Process Algebraic Approach. In *Art of Modelling Computational Systems*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11760, pp. 68–85. [CrossRef]
19. Brodo, L.; Bruni, R.; Falaschi, M. A logical and graphical framework for reaction systems. *Theor. Comput. Sci.* **2021**, *875*, 1–27. [CrossRef]
20. The BioReSolve System. 2021. Available online: http://www.di.unipi.it/~bruni/LTSRS/ (accessed on 14 March 2024).
21. The brsim System. 2014. Available online: https://github.com/scolobb/brsim/ (accessed on 14 March 2024).
22. Azimi, S.; Gratie, C.; Ivanov, S.; Petre, I. Dependency graphs and mass conservation in reaction systems. *Theor. Comput. Sci.* **2015**, *598*, 23–39. [CrossRef]

23. Ivanov, S.; Rogojin, V.; Azimi, S.; Petre, I. WEBRSIM: A Web-Based Reaction Systems Simulator. In *Enjoying Natural Computing—Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday*; Lecture Notes in Computer Science; Díaz, C.G., Riscos-Núñez, A., Paun, G., Rozenberg, G., Salomaa, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 11270, pp. 170–181. [CrossRef]

24. The HERESY System. 2017. Available online: https://github.com/aresio/HERESY (accessed on 14 March 2024).

25. Nobile, M.S.; Porreca, A.E.; Spolaor, S.; Manzoni, L.; Cazzaniga, P.; Mauri, G.; Besozzi, D. Efficient Simulation of Reaction Systems on Graphics Processing Units. *Fundam. Informaticae* **2017**, *154*, 307–321. [CrossRef]

26. The cl-rs System. 2020. Available online: https://github.com/mnzluca/cl-rs (accessed on 14 March 2024).

27. Ferretti, C.; Leporati, A.; Manzoni, L.; Porreca, A.E. The Many Roads to the Simulation of Reaction Systems. *Fundam. Informaticae* **2020**, *171*, 175–188. [CrossRef]

28. Meski, A.; Penczek, W.; Rozenberg, G. Model Checking Temporal Properties of Reaction Systems. *Inf. Sci.* **2015**, *313*, 22–42. [CrossRef]

29. Meski, A.; Koutny, M.; Penczek, W. Towards Quantitative Verification of Reaction Systems. In Proceedings of the Unconventional Computation and Natural Computation—15th International Conference, UCNC 2016, Manchester, UK, 11–15 July 2016; Proceedings; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9726, pp. 142–154.

30. Meski, A.; Koutny, M.; Penczek, W. Verification of Linear-Time Temporal Properties for Reaction Systems with Discrete Concentrations. *Fundam. Informaticae* **2017**, *154*, 289–306. [CrossRef]

31. ReactICS: Reaction Systems Verification Toolkit. 2023. Available online: https://arturmeski.github.io/reactics/ (accessed on 14 March 2024).

32. Azimi, S.; Gratie, C.; Ivanov, S.; Manzoni, L.; Petre, I.; Porreca, A.E. Complexity of Model Checking for Reaction Systems. *Theor. Comput. Sci.* **2016**, *623*, 103–113. [CrossRef]

33. Talcott, C. Pathway Logic. In Proceedings of the 8th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2008), Bertinoro, Italy, 2–7 June 2008; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5016, pp. 21–53.

34. Ochoa, C.; Silva, J.; Vidal, G. Dynamic slicing of lazy functional programs based on redex trails. *Higher Order Symbol. Comput.* **2008**, *21*, 147–192. [CrossRef]

35. Field, J.; Tip, F. Dynamic dependence in term rewriting systems and its application to program slicing. *Inf. Softw. Technol.* **1998**, *40*, 609–636. [CrossRef]

36. Alpuente, M.; Ballis, D.; Frechina, F.; Sapiña, J. Assertion-based Analysis via Slicing with ABETS. *Theory Pract. Log. Program.* **2016**, *16*, 515–532. [CrossRef]

37. Brodo, L.; Bruni, R.; Falaschi, M. Dynamic Slicing of Reaction Systems Based on Assertions and Monitors. In Proceedings of the Practical Aspects of Declarative Languages—25th International Symposium, PADL 2023, Boston, MA, USA, 16–17 January 2023; Hanus, M., Inclezan, D., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2023; Volume 13880, pp. 107–124. [CrossRef]

38. Brodo, L.; Olarte, C. Symbolic Semantics for Multiparty Interactions in the Link-Calculus. In Proceedings of the SOFSEM'17, Limerick, Ireland, 16–20 January 2017; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10139, pp. 62–75. [CrossRef]

39. Bodei, C.; Brodo, L.; Focardi, R. Static Evidences for Attack Reconstruction. In *Programming Languages with Applications to Biology and Security—Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*; Lecture Notes in Computer Science; Bodei, C., Ferrari, G., Priami, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9465, pp. 162–182. [CrossRef]

40. Bodei, C.; Brodo, L.; Gori, R.; Levi, F.; Bernini, A.; Hermith, D. A static analysis for Brane Calculi providing global occurrence counting information. *Theor. Comput. Sci.* **2017**, *696*, 11–51. [CrossRef]

41. Alpuente, M.; Ballis, D.; Sapiña, J. Static Correction of Maude Programs with Assertions. *J. Syst. Softw.* **2019**, *153*, 64–85. [CrossRef]

42. Alpuente, M.; Ballis, D.; Romero, D. A Rewriting Logic Approach to the Formal Specification and Verification of Web Applications. *Sci. Comput. Program.* **2014**, *81*, 79–107. [CrossRef]

43. Alpuente, M.; Ballis, D.; Frechina, F.; Sapiña, J. Combining Runtime Checking and Slicing to improve Maude Error Diagnosis. In *Logic, Rewriting, and Concurrency—Festschrift Symposium in Honor of José Meseguer*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9200, pp. 72–96.