A Logic Programming Approach to Reaction Systems

Moreno Falaschi
Department of Information Engineering and Mathematics, University of Siena, Italy
https://www3.diism.unisi.it/people/person.php?id=485
moreno.falaschi@unisi.it

Giulia Palma
Department of Computer Science, University of Pisa, Italy
giuliapalma29@gmail.com

Abstract

Reaction systems (RS) are a computational framework inspired by the functioning of living cells, suitable to model the main mechanisms of biochemical reactions. RS have shown to be useful also for computer science applications, e.g. to model circuits or transition systems. Since their introduction about 10 years ago, RS matured into a fruitful and dynamically evolving research area. They have become a popular novel model of interactive computation. RS can be seen as a rewriting system interacting with the environment represented by the context. RS pose some problems of implementation, as it is a relatively recent computation model, and several extensions of the basic model have been designed. In this paper we present some preliminary work on how to implement this formalism in a logic programming language (Prolog). To the best of our knowledge this is the first approach to RS in logic programming. Our prototypical implementation does not aim to be highly performing, but has the advantage of being high level and easily modifiable. So it is suitable as a rapid prototyping tool for implementing several extensions of reaction systems in the literature as well as new ones. We also make a preliminary implementation of a kind of memoization mechanism for stopping potentially infinite and repetitive computations. Then we show how to implement in our interpreter an extension of RS for modeling a nondeterministic context and interaction between components of a (biological) system. We then present an extension of the interpreter for implementing the recently introduced networks of RS.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Computing methodologies → Symbolic calculus algorithms; Software and its engineering → Interpreters

Keywords and phrases reaction systems, logic programming, non deterministic context

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.6

Acknowledgements We thank the anonymous reviewers for their detailed and very useful criticisms and recommendations that helped us to improve our paper.

1 Introduction

Natural Computing is an area of research which has two main aspects: human designed computing (models and computational techniques) inspired by nature and computation taking place in nature (i.e. it also investigates processes taking place in nature in terms of information processing). The first strand of research is quite well-established. This paper falls into this second strand of research, since it discusses reaction systems which are a formal model for the investigation of the functioning of the living cell introduced by A. Ehrenfeucht and G. Rozenberg [16, 17]. The functioning is viewed in terms of formal processes resulting from interactions between biochemical reactions taking place in the living cell. The basic model of reaction systems abstracts from various (technical) features of biochemical reactions to such an extent that it becomes a qualitative rather than quantitative model [7, 15]. However, it takes into account the basic bioenergetics (flow of energy) of the

living cell, and it also takes into account that the living cell is an open system (in the sense that it interacts with its environment) and its behavior is influenced by its environment. The main focus of research is on understanding processes that take place in these models. The Reaction Systems model has already been applied and extended successfully to various areas of research, since it is relevant in several different fields, such as computer science, biology, molecular chemistry [20, 21, 9, 3].

In this paper we present our preliminary work on how to implement the framework of RS in a logic programming language (Prolog). To the best of our knowledge this is the first approach to RS in logic programming. We illustrate our program by means of some simple basic examples throughout the paper, and then we consider a more complex example, by modeling a reaction system representing a regulatory network for *lac* operon, presented in [12]. We have also implemented a kind of memoization mechanism for stopping potentially infinite and repetitive computations. We discuss also some extensions of the basic framework and their implementation. First, we discuss how to implement an extension of RS for modeling nondeterministic contexts. Then, we show how to implement two RS which interact between them. Finally we present a prototypical implementation of the recently introduced networks of reaction systems [6]. Our interpreter is freely available online.

Structure of the paper. In Section 2, we present the basic framework of Reaction Systems. Then Section 3 is devoted to describing the implementation of the basic framework. Section 4 presents the implementation of a biological example. In Section 5 we discuss some extensions of RS and the corresponding implementation. We draw some conclusions and discuss future work in Section 6.

1.1 Related work

Reaction systems pose some interesting problems of implementation, as it is a recent computation model, and several extensions of the basic model have been designed. In particular, RS are amenable to both theoretical studies and as a modeling tool for biological processes. Their dynamics occupy an intermediate position between Cellular Automata [19] and Boolean Automata Networks [13]. However, since reaction systems have also been employed to model real-world systems, the availability of fast and efficient simulators is essential for a more widespread use of them as a modeling tool. The first available simulator was *brsim* [1], written in Haskell and it has been the fastest CPU-based simulator available for a relatively long time. Later, a GPU-based approach to the simulation of reaction systems has been explored with *HERESY* in [23], written using CUDA. It has been shown to be the fastest simulator for large-scale systems, due to its ability to exploit the large number of computational units inside GPUs. It also provides a CPU-based simulator written in Python 2, however it is more a "fallback" simulator when GPUs are not available, and is slower than *brsim*. Both simulators employ the same direct simulation method, which is based on the set-theoretic definition of the reaction systems' dynamics. Recently, in [18] the authors provide an optimized Common Lisp simulator, employing the direct simulation method, which is able to offer performances comparable with the GPU-based simulator on a large-scale real-world model, the *ErbB* model. It has been shown to be the fastest CPU-based simulator currently available. They also explore other ways of performing the simulation, in particular, by looking at the graph of dependencies between reactions, it is possible to avoid performing the simulation of parts of the reactions that cannot produce any effect on its dynamics; and by rewriting the dynamical evolution of a reaction system in terms of matrix-vector multiplications, vector additions, and clipping operations, they exploit the existing high-performance linear algebra libraries to

perform the simulation and therefore they use a proof-of-concept implementation employing Python 3 and Numpy. Regarding the non-determinism of the context, some results are illustrated in [9], in which the authors consider the *link*-calculus [8, 11], which allows to model multiparty interaction in concurrent systems, and show that it allows to embed reaction systems, by representing the behaviour of each entity and preserving faithfully their features. Such a framework contribute to increase the expressiveness of reaction systems, indeed it exploits the interaction among different reaction systems. In [10] the authors show how to define a context which can be really expressive, by adding to it a non deterministic and a recursive operator.

In this paper we present some preliminary work on how to implement the formalism of RS in a logic programming language: Prolog. Although this prototypical implementation is not highly performing and competitive compared to the above mentioned implementation, it has the considerable advantage of being very high level and easily modifiable. Therefore, it is suitable as a working rapid prototyping tool for implementing extensions of reaction systems, as we show in this paper.

2 Reaction Systems

Natural Computing is concerned with human-designed computing inspired by nature as well as with computation taking place in nature. The theory of Reaction Systems [16, 7] was born in the field of Natural Computing to model the behavior of biochemical reactions taking place in living cells. The original motivation was to understand interactions of biochemical reactions in the living cell from the natural computing point of view. These interactions are based on mechanisms of *facilitation* and *inhibition*, which underlie the definition of *reaction system*. A *reaction* is a chemical process in which substances act mutually on each other and are changed into different substances, or one substance changes into other substances. A reaction takes place if all its *reactants* are present and none of its *inhibitors* is present. If a reaction takes place, then it creates its *products*. Therefore to specify a reaction one needs to specify its set of *reactants*, its set of *inhibitors* and its set of *products*.

▶ **Definition 1** (Reaction). *A reaction is a triplet $a = (R, I, P)$, where R, I, P are finite sets. If S is a set such that $R, I, P \subseteq S$, then a is a reaction in S.*

The sets R, I, P are also written R_a, I_a, P_a and called the reactant set of a, the inhibitor set of a, and the product set of a, respectively. Also, $R_a \cup I_a$ is the set of the resources of a and $rac(S)$ denotes the set of all reactions in S. Because R and I are non empty, all products are produced from at least one reactant and every reaction can be inhibited in some way. Sometimes artificial inhibitors are used that are never produced by any reaction. For the sake of simplicity, in some examples, we will allow I to be empty.
The effect of a reaction a is conditional: if R_a is present and no element of I_a is present, then P_a is produced. Otherwise, the reaction does not take place, and "nothing" is produced.

▶ **Definition 2** (Result of Reaction). *Let a be a reaction, A a finite set of reactions and T a finite set.*

- *a is enabled by T if $R_a \subseteq T$ and $I_a \cap T = \emptyset$ (indicated by $en_a(T)$);*
- *The result of a on T is defined by:*

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

- *The results of A on T is defined by $res_A(T) = \bigcup_{a \in A} res_a(T)$.*

Now that the formal notion of a reaction and its effect on states have been established, we can proceed to define reaction systems, which are an abstract model of the functioning of the living cell. A *reaction system* is essentially a set of reactions. We also specify the background set, which consists of entities needed for defining the reactions and for reasoning about the system.

▶ **Definition 3** (Reaction Systems). *A reaction system, abbreviated rs, is an ordered pair $\mathcal{A} = (S, A)$ such that S is a finite set, and $A \subseteq rac(S)$.*

The set S is called the *background set* of $\mathcal{A}$. Its elements are called *entities*, they represent molecular entities (e. g. atoms, ions, molecules) that may be present in the state of a biochemical system modeled by $\mathcal{A}$. The set A is the set of *reactions* of $\mathcal{A}$. Since S is finite, so is A. All the notations introduced for sets of reactions carry over to reaction systems: $T \subseteq S, en_{\mathcal{A}}(T) = en_A(T); res_{\mathcal{A}}(T) = res_A(T)$; T is active in $\mathcal{A}$, if $en_{\mathcal{A}}(T) \neq \emptyset$. The theory of Reaction Systems is based on the following assumptions:

1. **No permanency.** An entity of a set T vanishes unless it is sustained by a reaction. This reflects the fact that a living cell would die for lack of energy, without chemical reactions.
2. **No counting.** The basic model of reaction systems is very abstract and qualitative, i.e. the quantity of entities that are present in a cell is not taken into account. In fact the number of reagents does not count in the reaction systems model, unlike the stoichiometric equations, in which the quantities are fundamental.
3. **Threshold nature of resources.** From the previous item, we assume that either an entity is available and there is enough of it (i.e. there are no conflicts), or it is not available at all.

The dynamic behavior of reaction systems is captured through the notion of interactive process:

▶ **Definition 4** (Interactive Process). *Let $\mathcal{A} = (S, A)$ be a reaction system. An interactive process in $\mathcal{A}$ is a pair $\pi = (\gamma, \delta)$ of finite sequences such that: $\gamma = C_0, C_1, \ldots, C_n$, $\delta = D_1, \ldots, D_n$, $n \geq 1$, where $C_0, \ldots, C_n, D_1, \ldots, D_n \subseteq S, D_1 = res_{\mathcal{A}}(C_0)$, and $D_i = res_{\mathcal{A}}(D_{i-1} \cup C_{i-1})$ for $2 \leq i \leq n$.*

Living cells are seen as open systems that continuously react with the external environment, in discrete steps. The sequence γ is the *context sequence* of π and represents the influence of the environment on the Reaction System. The sequence δ is the *result sequence* of π and it is entirely determined by γ and A. Note that C_i and D_i do not have to be disjoint. Let $W_0 = C_0$ and $W_i = C_i \cup D_i$ for all $1 \leq i \leq n$. The sequence $W_0, \ldots, W_n$ is the *state sequence of* π, $sts(\pi)$. W_0 is the initial state of π. For each $0 \leq j \leq n$, C_j is the *context* of W_j.

Let us consider a clarifying example that illustrates the concepts introduced in Section 2.

▶ **Example 5.** Let us consider a reaction system $\mathcal{A} = (\{e_1, e_2, e_3, e_4\}, A)$, where A is the set of the two reactions:

$$a_1 = (\underbrace{\{e_1, e_2\}}_{R_1 = \text{ Reactants of } a_1}, \underbrace{\{e_3\}}_{I_1 = \text{ Inhibitors of } a_1}, \underbrace{\{e_2, e_3\}}_{P_1 = \text{ Products of } a_1})$$

$$a_2 = (\underbrace{\{e_1\}}_{R_2 = \text{ Reactants of } a_2}, \underbrace{\{e_4\}}_{I_2 = \text{ Inhibitors of } a_2}, \underbrace{\{e_1, e_4\}}_{P_2 = \text{ Products of } a_2})$$

The sequence $\tau = \{e_2, e_3, e_1, e_4\}, \varnothing$ is a context-indipendent state sequence of $\mathcal{A}$, assuming that the initial state is $T_0 = \{e_1, e_2\}$. Indeed: $R_1 = T_0$ and $T_0 \cap I_1 = \varnothing$, then reaction a_1 takes place, producing P_1; also $R_2 \subset T_0$ and $I_2 \cap T_0 = \varnothing$, then reaction a_2 takes place producing P_2.

Therefore, we get $T_1 = P_2 \cup P_3 = \{e_2, e_3, e_1, e_4\}$. Now, since $I_1 \cap T_1 = \{e_3\} \neq \varnothing$, then reaction a_1 does not take place. Reaction a_2 does not take place either, in fact $I_2 \cap T_1 = \{e_4\} = \varnothing$. Finally, we get $T_2 = \varnothing$.
Let us now assume that the computation is not context-indipendent. If the context sequence is $\gamma = \{e_1, e_4\}, \{e_4\}$, then the corresponding state sequence is $\tau = \{e_4\}, \varnothing$. Indeed: $G_0 = T_0 \cup C_0 = \{e_1, e_2, e_4\}$. From the fact that $G_0 \cap I_2 = \{e_4\} \neq \varnothing$, we get the reaction a_2 does not take place. Instead reaction a_1 occurs in fact $R_1 \subseteq G_0$ and $I_1 \cap G_0 = \varnothing$. Then we get $T_1 = \{e_2, e_3\}$. Now, we have $G_1 = T_1 \cup \{e_4\} = \{e_2, e_3, e_4\}$. Since $G_1 \cap I_1 = \varnothing$ and $G_1 \cap I_2 = \varnothing$, neither reaction a_1 nor reaction a_2 take place. Therefore, we get $T_2 = \varnothing$.

3 A logic programming approach to Reaction Systems

In this Section we briefly describe a prototypical implementation of the Reaction Systems framework in a logic programming language (Prolog), which is available on-line[1], together with a small manual to use it.

3.1 An Interpreter of Reaction Systems in logic programming

Sets are represented by corresponding lists of values. The background set `S` of a reaction system is represented by a list of distinct constant symbols. A reaction `(R, I, P)` is represented by a triple of lists, where `R` is the list of the reactants, `I` is the list of the inhibitors and `P` is the list of products. The set of reactions in a reaction system is defined by a list of reactions and is introduced by using the predicate `reactionSet/1`. So, this predicate is fundamental and one fact for this predicate must be included. If the computation is context independent `reactionSet/1` is the only predicate for which we have to add a unit fact in the program. If we want to perform a computation context dependent, then we have to add also a unit fact for the other fundamental predicate `context/1`. The predicate `context/1` takes as input a list of context lists. Hence a user has to modify only one, or at most two unit facts to be able to run her reaction system.

3.2 A computation with the interpreter of Reaction Systems

Now we briefly describe some of the main predicates which are part of the interpreter of Reaction Systems.

When evaluating a query to our interpreter, the predicate which needs to be called is `computation(InitialState, ListOfStates)`. The first input argument `InitialState` is the list of the reagents to be put in the initial state. The second argument `ListOfStates` is the list of states which is computed in the reaction system by our interpreter. So, a query to our interpreter consists of a call to `computation/2`.

The execution of the predicate `computation/2` starts by making some preliminary checks (predicate `preliminaryCheck/1`) to verify that the basic assumptions on reaction systems are respected. Namely, for each reaction `(R,I,P)` the set of reagents `R` and the sets of inhibitors `I` are non empty, and they don't share elements. Then, the interpreter will give the user some choices:

1) whether she wants to make a context independent computation or a computation which interacts with the context.

[1] `https://www3.diism.unisi.it/~falaschi/ReactionSystems`

2) whether she wants to make a computation with a limited maximal number of steps, or if the computation should be of a possibly unlimited length.

Then the appropriate predicate corresponding to the choice of the user is selected, between the following four ones:

```
computationLimitedToKStepsContextIndependent/2
computationLimitedToKSteps/2
unlimitedComputationContextIndependent/2
unlimitedComputation/2
```

Notice that `unlimitedComputation/2` and `unlimitedComputationContextIndependent/2` may enter in a loop if there is a reaction `(R,I,P)` in which the same reactant in `R` appears in `P`, or more in general when there are dependencies between the reactants in R and the ones computed by some other reaction. A loop can be stopped by using the inhibition mechanism, or by a memoization mechanism, as explained at the end of this section.

A single step of the computation is performed as follows. The result of a single reaction (without the context) is computed by the predicate `result (T, R, I, P, P1)`. Given the state `T` and the reaction `(R, I, P)`, the result `P1` will be `P` if the reaction is enabled in `T` (that is, if the predicate `enable` is true), otherwise it will return the empty set. The predicate `enable (R, I, T)` checks if the reaction with reactants `R` and inhibitors `I` is enabled in the state `T`. We recall that in a reaction system for a reaction to occur it must hold: $\texttt{R} \subseteq \texttt{T}$ and $\texttt{I} \cap \texttt{T} = \emptyset$. Then the result of all reactions on T is computed by the predicate `resultallreactions (T, ReactionSet, T1)`, which recursively calls `result/5` and collects the union of its outcomes.

Let us see a trivial example.

▶ **Example 6.** Let us define the predicate `reactionset` as follows:

```
reactionset ([([ e1, e2 ], [ e3 ], [ e2, e3 ]),  ([e1], [ e4 ], [ e1, e4 ])]).
```

This means that there are two reactions in the system. We execute the following query:

```
? - computation ( [ e1, e2], L ).
```

Then, by selecting the modality "computation context independent" we get:

```
L = [ [ e2, e3, e1, e4 ], [ ] ]
```

that is the next state `[ e2, e3, e1, e4 ]` and the final state `[ ]`. The computation in this case uses an empty context represented internally by an empty list.

We now modify the example in order to show the interactive influence of the context. To consider the effect of the context, we need to add a unit fact for the predicate `context/1`. The input argument of this predicate must be a list of context reagent lists. The list in position k corresponds to the context to be added at step k of the computation. For instance: `context([[e1, e2], [e3, e2, e5]])`.

If the context list has length m, and the computation is longer, it continues from step $m+1$ as context independent. We define the predicate that calculates a computation starting from an initial state, returning a list of states. The context is taken into account now. We report here a small fragment of the interpreter. The predicate `computeWithContext(ComputState,Context,Reactions,ComputStateSequence)` takes in input the current `ComputState`, the `Context` sequence, the list of `Reactions` in the Reaction System, and returns the computed `ComputStateSequence`.

```
unlimitedComputation(InitialState,L):-
     reactionSet(R),context([C0|Cs]),
     union(InitialState,C0,SC),computeWithContext(SC,Cs,R,L).

computeWithContext ([  ], C, R, [ ]).
computeWithContext ([ X | L ], [  ], R, [ S1 | S ]) : -
     resultallreactions ([ X | L ], R, S1),
     computeWithContext (S1,  [  ], R, S).
computeWithContext ([ X | L ], [C | Cs], R, [S1 | S ]) : -
     resultallreactions ([ X | L ], R, S1),
     union (S1, C, S2), computeWithContext (S2, Cs, R, S).
```

Let us see a simple example.

▶ **Example 7.** Let us define the context and the reactions of a reaction system as follows:

```
context([ [e1, e2], [e3, e2, e5] ]).
reactionSet([([e1,e2],[e3],[e2,e3]),  ([e5], [e4], [e1,e4])]).
```

We can execute the following query (starting from an empty initial state):

```
? - computation ( [ ], L ).
```

We get:

```
L = [ [e2, e3], [e1, e4], [ ] ].
```

that is the next state is `[e2, e3]`, then `[e1, e4]` and the final state `[ ]`.

3.3 Stopping unlimited computations with memoization

A problem which may arise during a computation in a reaction system is that a loop can be created easily either directly in one reaction or with dependencies between different reactions. For instance consider the reaction `[[a],[b],[a]]`. If we start with the initial state `[a]`, then an infinite sequence `[a]`, `[a]`, `[a]`, ... will be generated, unless the context introduces the inhibitor `b` at some stage of the computation.

We have extended our interpreter by using a technique in the style of "memoization". So we have defined a predicate `unlimitedComputationContextIndependentMemoized` which keeps track of the states of the computation generated, and as soon as a state of the computation is repeated (i.e. it appears identical in a previous step), the computation is stopped and the finite sequence of states until the current one is returned.

4 Reaction systems: a biological example

In this section we present the encoding of a reaction system example taken from [12], that regards the *lac* operon mechanism in the reaction system formalism. Therefore, we preliminary introduce the most essential notions about the *lac* operon.

4.1 The *lac* operon

An *operon* is a functioning unit of DNA containing a cluster of genes under the control of a single promoter (i.e. a sequence of DNA to which proteins bind in order to initiate transcription). The *lac* operon is involved in the metabolism of lactose in *Escherichia coli* cells (i.e. a bacteria which lives in the intestines of mammals and birds and which is needed

to digest food). It is composed by three adjacent structural genes (plus some regulatory components): *lacZ*, *lacY* and *lacA* encoding for two enzymes Z and A, and a transporter Y, involved in the digestion of the lactose. The main regulations are:

- The DNA sequence, called *promoter*, is recognized by a RNA polymerase (i.e. an enzyme that synthesizes RNA from a DNA template) to iniziate the transcription of the genes *lacZ*, *lacY* and *lacA*;
- The gene *lacI* encodes for a repressor protein I;
- A DNA segment, called the *operator* (OP), obstructs the RNA polymerase functionality when the repressor protein I is bound to it forming I-OP;
- A short DNA sequence, called the CAP-binding site, when it is bound to the complex composed by the protein CAP and the signal molecule $cAMP$, acts as a promoter for the interaction between the RNA polymerase and the promoter.

The functionality of the *lac* operon is based on the integration of two control mechanisms of which, one is mediated by lactose, while the other one is mediated by glucose.

1. In the first control mechanism, an effect of the absence of the lactose is that I can bind the operator sequence preventing in this way the lac operon expression. If lactose is available, I is unable to bind the operator sequence, and then the *lac* operon can be potentially expressed.
2. In the second control mechanism, in the absence of glucose, the molecule $cAMP$ and the protein CAP increase the lac operon expression, thanks to the fact that the binding between the molecular complex $cAMP$-CAP and the CAP-binding site increases.

Therefore, to sum up, the condition that promotes the operon gene expression is the presence of lactose and the absence of glucose.

4.2 The Reaction System formalization

The reaction system for the *lac* operon is defined as $A_{lac} = (S, A)$, where the set S represents the main biochemical components involved in the considered genetic system and the reaction set A contains the biochemical reactions involved in the regulation of the *lac* operon expression. $S = \{lac, Z, Y, A, lacI, I, I\text{-}OP, cya, cAMP, crp, CAP, cAMP\text{-}CAP, lactose, glucose\}$ and A consists of the following 10 reactions:

$$
\begin{array}{lcllcl}
a_1 & = & (\{lac\}, \{...\}, \{lac\}), & a_6 & = & (\{cya\}, \{...\}, \{cAMP\}), \\
a_2 & = & (\{lacI\}, \{...\}, \{lacI\}), & a_7 & = & (\{crp\}, \{...\}, \{crp\}), \\
a_3 & = & (\{lacI\}, \{...\}, \{I\}), & a_8 & = & (\{crp\}, \{...\}, \{CAP\}), \\
a_4 & = & (\{I\}, \{lactose\}, \{I\text{-}OP\}), & a_9 & = & (\{cAMP, CAP\}, \{glucose\}, \{cAMP\text{-}CAP\}), \\
a_5 & = & (\{cya\}, \{...\}, \{cya\}), & a_{10} & = & (\{lac, cAMP\text{-}CAP\}, \{I\text{-}OP\}, \{Z, Y, A\}).
\end{array}
$$

where $\{...\}$ stands for any dummy inhibitor. Observe that reactions a_1, a_2, a_5, a_7 are necessary to grant the permanency of the genes in the system; while reactions a_4, a_9, a_{10} can only be enabled if the current state of the system does not include the inhibitor elements specified in each reaction. In more details, reaction a_4 can be applied only in the absence of lactose, reaction a_9 in the absence of glucose, and reaction a_{10} when repressor I is not bound to the operator OP.

The *lac* operon expression is based on which substrates the environment provides. In order to translate this situation in the *lac* operon reaction system, we need to evaluate what happens to the system when the context provides both glucose and lactose, only glucose, only lactose, or none of them. To do this, we define a default context (DC) that mimics the real biological system in which the genomic elements plus their encoded proteins are

normally present; hence DC is composed by those entities that are always present in the system $DC = \{lac, lacI, I, cya, cAMP, crp, CAP\}$, whereas the lactose and the glucose are given non-deterministically by the context.

4.3 The Reaction System encoding in Prolog

We now show the encoding of the considered reaction systems in Prolog. We note that, by definition, the set of inhibitors should not be empty, but in this example most of the triples have empty inhibitors. Thus, we add a dummy inhibitor gp, i.e. a new constant that does not appear in any reaction, if the set of inhibitors is empty. If we add the new inhibitor gp in all sets of inhibitors in the rules, then we can use it to force the termination of a computation, when the context introduces it. This is useful, as reactions such as "$([a], [\ ], [a])$" may cause an infinite loop. Notice that since a reaction system has a finite background set, we can prove the following property:

▶ **Proposition 8.** *If a reaction system enters in an infinite loop then the infinite computation has the form* $W_0, W_1, \ldots\ W_m \ldots W_k \ldots$, *where* $m < k$, *and* $W_m = W_k$.

This means that the subsequence $W_m, \ldots, W_k$ will then be repeated iteratively.

Let us now consider the program presented in Section 3, where we replace the rules for predicates `reactionset` and `context` by the following ones:

```
reactionset([([lac], [gp], [lac]), ([laci],[gp],[laci]),
      ([laci],[gp],[ig]), ([ig],[lactose, gp],[iop]),
      ([cya],[gp],[cya]), ([cya],[gp], [camp]),
      ([crp],[gp],[crp]), ([crp],[gp],[cap]),
      ([camp, cap],[glucose, gp],[campcap]),
      ([lac, campcap], [iop, gp],[z,y,a])]).

context([[lac,laci,ig,cya,camp,crp,cap],
        [glucose], [glucose], [glucose],
        [glucose], [glucose], [glucose, lactose],
        [glucose, lactose], [glucose, lactose],
        [glucose, lactose], [glucose, lactose],[gp]]).
```

By executing the following query:

```
? - computation ( [  ], L ).
```

We get:

```
L =  [[lac, laci, ig, iop, cya, camp, crp, cap, campcap],
      [[lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, iop, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap],
      [lac, laci, ig, cya, camp, crp, cap], [  ] ]
```

In the following section, we will present some extensions of the basic framework and a possible implementation in our interpreter.

5 Extensions of the basic framework: modifications to the context

In this section we want to show that our interpreter is flexible and can be exploited as a rapid prototyping tool for implementing prototypes of extensions of the basic framework of reaction systems. We will illustrate this characteristic by defining first a very simple nondeterministic extension of reaction systems, and then showing how to implement a recent extension of reaction systems to a network of them [6].

5.1 Non-deterministic context

Reaction systems are deterministic. However, the evolution of a computation interacting with a context depends on such interaction. So, recently some work has focused on extending the context behaviour to make it more expressive. For instance [10] has designed an extension of the context based on process algebras which allows for non deterministic and even recursive contexts. Here we propose a much simpler extension, by adding a nondeterministic operator to the context.

The implementation of non-deterministic finite transition systems provides an instructive insight into the role of context in interactive processes. Let's modify our program and add a non-determinism operator in context. In this way, the context instead of being made from a sequence of lists $S_1, S_2, \ldots$, will be a list in which each element of the list is a list of lists from which it can be chosen not deterministically. For example $(S_{11} + S_{12} + \cdots + S_{1k_1})(S_{21} + S_{22} + S_{23} + \cdots + S_{k_2}) \ldots$ and the system chooses one of these lists at each step in a completely non-deterministic way. If our context sequence was $(S_{11})(S_{21} + S_{22})(S_{31} + S_{32})$, then the possible (context) sequences generated in a non-deterministic way would be S_{11}-S_{21}-S_{31} or S_{11}-S_{21}-S_{32} or S_{11}-S_{22}-S_{31} or S_{11}-S_{22}-S_{32}.

The nondeterministic choice on each step of the context is performed by the predicate `chooseContext/2`, which chooses randomly one of the contexts in the list.

We modify our program in the following way:

```
contextND([[[a1,a2],[a3,a2,a5]],[[a2,a3,a,4],[a1,a2,a5],[a3]]]).

chooseContext(PossibleContext, ChoosenContext):-
        length(PossibleContext, Length),
        random(0,Lenght,Index),
        nth0(Index,PossibleContext,ChoosenContext).

context([],[]).
context([L|OtherList],[Cc|Cot]):- chooseContext(L,Cc),
context(OtherList,Cot).
computation(InitialState,L):- reactionSet(R), contexND(C),
         context(C,[C0|Cs]), union(InitialState,C0,SC),
         computeWithContext(SC,Cs,R,L).
```

We have defined the predicate *chooseContext* that selects a random element from a list of lists. The new context is a list whose elements are lists in which to choose a list. The new context is given by *contextND*. To create the context we defined the *context* predicate. Finally, we modified the *computation* predicate.

We notice that we will not add these modifications to our interpreter. These modifications could be useful to model easily non deterministic systems, with a *don't care* kind of non-determinism which is typical of concurrent systems [24]. Don't care nondeterminism means that only one of the possible choices is chosen, and the other alternatives are discarded. For an

extension of our interpreter which exploits a non deterministic context with the typical *don't know* non determinism of logic programming please see [10]. Don't know nondeterminism means that all possible choice alternatives are tried.

In the following section we show that our interpreter can be extended to model two interacting reaction systems.

5.2 Interaction of two Reaction Systems

We start by discussing first a simple extension to a network made by two reaction systems which "cooperate". We illustrate how it is possible to program two reaction systems encodings, in such a way that the entities that usually come from the context of one reaction system will be provided instead from the other reaction system.
A slight modification of our program allows us to consider two reaction systems in which the output of the first reaction system becomes the context of the other.

We can define two separate unit predicates for the reactions in the two RS, `reactionsetF/1` and `reactionsetS/1`. Then, we have to modify the `computation/2` predicate so that in the case of the first reaction system we provide the context via the context predicate; while for the second reaction system we use the list obtained from the computation of the first reaction system.

```
reactionsetF([([a1,a2],[a3],[a2,a3]),([a5],[a4],[a1,a4])]).
reactionsetS([([a1,a3],[a4],[a1,a3]),([a3,a5],[a4],[a2,a4])]).

computationF(InitialState,L):-reactionsetF(R),context([C0|Cs]),
       union(InitialState,C0,SC),computeWithContext(SC,Cs,R,L).

/* new predicate to calculate the second reaction system */
computation(InitialStateF,InitialStateS, L):- reactionsetS(R),
       computationF(InitialStateF,[C0|L1]), context([C0|L1]),
       union(InitialState,C0,SC),computeWithContext(SC,Cs,R,L).
```

In the following section we show that our interpreter can be extended to model networks of reaction systems. We have enclosed this extension in the interpreter available online.

5.3 Networks of Reaction Systems

Here we illustrate how to extend our interpreter for modeling networks of reaction systems as introduced in [6]. In [6] the context has its own structure: the context for a reaction system originates from a network of reaction systems. Such a network is formalized as a graph where nodes represent reaction systems, and where each reaction system contributes to defining the context of all its neighbours. Thus, as the context for a reaction system is given by a network of reaction systems communicating with it, the interaction between two reaction systems that we have introduced in Section 5.2 can be seen as a special case of the definition of a network of reaction systems. In the basic model of [6] reported here, all edges function as communication channels and states of reaction systems residing at nodes are synchronized according to a global clock.
We start by introducing the general notions of *centralized network* of reaction systems and *interactive network process*. In the network of RS that we will define, the j-th RS will be denoted by $\mathcal{A}^j = (S^j, A^j)$. $\mu : V \to \mathcal{F}$ is a *location function*, which assigns RS to nodes. So, for $v_j \in V$, $\mu(v_j) = \mathcal{A}^j$. The set of incoming neighbours of a node v in a graph, namely those nodes for which there is an edge connecting them to v, is denoted by $in(v)$. The following two definitions are from paper [6].

▶ **Definition 9.** *A centralized network of reaction systems is a tuple $\mathcal{N} = (G, \mathcal{F}, \mu)$, where $G = (V, E, v_0)$ is a finite centralized graph such that $in(v_0) \neq \emptyset$, $\mathcal{F}$ is a nonempty finite set of reaction systems, and $\mu : V \to \mathcal{F}$ is a location function, assigning reaction systems to nodes.*

The following definition formalises the notion of a computation of length n for an *interactive network process*, which is given by a vector of individual interactive processes of the reaction systems in the network nodes. The computation starts from an initial given distribution (C_0^j, D_0^j). Roughly speaking C_k^j represents the context for RS j at step k of computation, and D_k^j represents the state of RS j at step k of computation. Thus, for any node v_j, for each subsequent step i of the process associated with such a node π^j, the component D_i^j is obtained by applying enabled reactions from A^j to the current state, while the component C_i^j is given by the union of the results produced, at the previous step, by the incoming neighbours of v_j. It is finally made an intersection with S^j to filter out entities which are not in the background set of $\mathcal{A}^j$.

▶ **Definition 10.** *Let $\mathcal{N} = (V, E, v_0, \mathcal{F}, \mu)$ be a centralized network of reaction systems with $|V| = m + 1$ for some $m \geq 0$. For $n \in \mathbb{N}^+$, an interactive (n-step) network process is a tuple $\Pi = (\pi^0, \ldots, \pi^m)$, where, for $j \in \{0, \ldots, m\}$, $\pi^j = (\gamma^j, \delta^j)$ and $\gamma^j = (C_0^j, \ldots, C_n^j)$, $\delta^j = (D_0^j, \ldots, D_n^j)$, are such that:*

1. $C_k^j = S^j \cap \bigcup\{D_{k-1}^i | v_i \in in(v_j)\}$, *for* $k \in \{1, \ldots, n\}$,
2. $D_k^j = res_{A^j}(D_{k-1}^j \cup C_{k-1}^j)$ *for* $k \in \{1, \ldots, n\}$.
3. *If* $in(v_j) = \emptyset$, *then* $C_0^j = \emptyset$.

We now illustrate the implementation of the network of Reaction Systems. The new implementation proceeds for a limited number of steps K, where K is a number given at the beginning when it is requested by the program, or until an empty state is encountered. The complete derivation of the Reaction System 1 is calculated (it was called 0 in the previous definition). The reaction systems of the network corresponds to the nodes of the network and are numbered by positive integers 1, 2, 3, and so on. At the beginning, the overall number of Reaction Systems in the network must be given as input. In the following we present an example consisting of two reaction systems, but the program is valid for an arbitrary finite number of nodes. As output we obtain the final state of all the Reaction Systems in the network, and the complete computation of the reaction system 1. It is sufficient to invoke *main(F, D)*, so that the program calculates F and D, i.e. the overall final state F of the network and the complete derivation D for reaction system 1. We do not restrict the set of computed values to the background of the node. It would be easy to add such a restriction. For the sake of simplicity we assume that all RSs in the network have the same background set.

The edges of the network are represented by a predicate `network/1` introducing a list of pairs of the form `[m,n]` meaning that there is an arc from node `m` to node `n`. The predicate `search(N,Net,S0,S1)` looks for all pairs `[N1,N]` in `Net` and returns `S1` = union of the states in position `N1` of `S0`, thus computing the context for `N` in the network of RS.

The predicate `initialStates/1` is defined by a unit fact which introduces a list of list defining the initial states of the nodes in the network. So list in position k corresponds to the initial state of node k.

Let us see a fragment of one example. For more details please refer to the interpreter online.

```
reaction(1,[([lac],[ a],[cya]),([lacI],[a ],[lac2]),
       ...([lac2],[a ],[lac3]),([cya],[ a],[cya3])]).
reaction(2,[([lac],[ a],[cya1,cya]),([lacI],[a ],[lac2]),
       ...([lac2],[a ],[lac3]),([cya],[ a],[cya,cya2])]).

network([[2,1]]).

computeOneStep(N,S,S2):- computeState(S,S0,1),
  network(Net), computeContext(S0,Net,N,S1),
  unionList(S0,S1,S2).

computeState([],[],K).
computeState([S|Ss],[S1|S1s],K):-reaction(K,R),
  resultallreactions(S,R,S1), K1 is K+1,
  computeState(Ss,S1s,K1).

computeContext(S,Net,N,S0):- computeContext1(S,Net,N,S0,1).

computeContext1(S,Net,N,[],N1):-N<N1.
computeContext1(S,Net,N,[S1|S0],N1):-N1=<N,
  search(N1,Net,S,S1),N2 is N1+1,
  computeContext1(S,Net,N,S0,N2).

initialStates([[lac],[lac]]).
```

An example of execution follows:

```
| ?- main(F,D).
Give me the number of Reaction Systems in the Network
(a positive integer, followed by a dot) 2.
Give me the maximun number of computation steps
(a positive integer, followed by a dot) 5.

D = [[cya1,cya],[cya3,cya,cya2],[cya3,cya,cya2], [cya3,cya,cya2],[cya3,cya,cya2]]
F = [[cya3,cya,cya2],[cya,cya2]]
```

This model of communicating reaction systems can enable the study of the behaviour of one reaction system in relation to other ones. This way, the lac operon system can be connected with the two systems producing the lactose and the glucose, and therefore the presence of these two entities in the lac operon system can be regulated by realistic mechanisms.

6 Conclusions and future work

In this paper we have recalled the framework of Reaction Systems introduced by A. Ehrenfeucht and G. Rozenberg [16]. Then we have described our preliminary implementation of this framework in Prolog. We have then shown that our interpreter is flexible and suitable for rapid prototyping and implementing extensions of the basic framework. It allows to make indefinitely long computations, computations limited to a maximum of k steps, and we have also introduced a kind of memoization mechanism based on accumulators for stopping a computation when a state gets repeated. The user can choose her preferences. Thus, we have shown how to implement an extension of RS for modeling nondeterministic contexts with don't care non determinism, and two interacting RS, and then we have implemented the recently introduced networks of reaction systems [6]. Our interpreter is freely available

online. As a future work we plan to improve the implementation to make it more efficient by using constraint logic programs, by exploiting finite domains, and CLP(SET) [14], and more user friendly, also by interfacing it to graphical tools for showing the computations in our framework. We also plan as a future work to study how to exploit the structures which have been defined for representing efficiently enormous numbers of states in model checking, in order to improve the evaluation of reaction systems. Some work has already been done in [22]. We also want to study the application of static analysis techniques [2, 5, 4] to RS.

References

1 S. Azimi, C. Gratie, S. Ivanov, and I. Petre. Dependency graphs and mass conservation in reaction systems. *Theoretical Computer Science*, 598:23—-39, 2015. doi:10.1016/j.tcs.2015.02.014.

2 R. Barbuti, R. Gori, F. Levi, and P. Milazzo. Investigating dynamic causalities in reaction systems. *Theor. Comput. Sci.*, 623:114–145, 2016. doi:10.1016/j.tcs.2015.11.041.

3 A. Bernini, L. Brodo, P. Degano, M. Falaschi, and D. Hermith. Process calculi for biological processes. *Natural Computing*, 17(2):345–373, 2018. doi:10.1007/s11047-018-9673-2.

4 C. Bodei, L. Brodo, and R. Focardi. Static evidences for attack reconstruction. In *Proc. of Programming Languages with Applications to Biology and Security*, volume 9465 of *Lecture Notes in Computer Science*, pages 162–182. Springer, 2015. doi:10.1007/978-3-319-25527-9_12.

5 C. Bodei, L. Brodo, R. Gori, F. Levi, A. Bernini, and D. Hermith. A static analysis for brane calculi providing global occurrence counting information. *Theor. Comput. Sci*, 696:11–51, 2017. doi:10.1016/j.tcs.2017.07.008.

6 P. Bottoni, A. Labella, and G. Rozenberg. Networks of reaction systems. *International Journal of Foundations of Computer Science*, 31:53–71, 2020. doi:10.1142/S0129054120400043.

7 R. Brijder, A. Ehrenfeucht, M. Main, and G. Rozenberg. A tour of reaction systems. *International Journal of Foundations of Computer Science*, 22(07):1499–1517, 2011. doi:10.1142/S0129054111008842.

8 L. Brodo. On the expressiveness of pi-calculus for encoding mobile ambients. *Mathematical Structures in Computer Science*, 28(2):202–240, 2018. doi:10.1017/S0960129516000256.

9 L. Brodo, R. Bruni, and M. Falaschi. Enhancing reaction systems: a process algebraic approach. In M. Alvim, K. Chatzikokolakis, C. Olarte, and F. Valencia, editors, *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy*, volume 11760 of *Lecture Notes in Computer Science*, pages 68–85. Springer Berlin, 2019. doi:10.1007/978-3-030-31175-9_5.

10 L. Brodo, R. Bruni, and M. Falaschi. SOS rules for equivalences of reaction systems. In *Pre-proceedings of the 28th Int. workshop on Functional and Logic Programming (WFLP 2020)*, 2020. arXiv:2009.01001.

11 L. Brodo and C. Olarte. Symbolic semantics for multiparty interactions in the link-calculus. In *Proc. of SOFSEM'17*, volume 10139 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2017. doi:10.1007/978-3-319-51963-0_6.

12 L. Corolli, C. Maj, F. Marinia, D. Besozzi, and G. Mauri. An excursion in reaction systems: From computer science to biology. *Theoretical Computer Science*, 454:95–108, 2012. doi:10.1016/j.tcs.2012.04.003.

13 J. Demongeot, M. Noual, and S. Sené. On the number of attractors of positive and negative boolean automata circuits. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 782–789, 2010. doi:10.1109/WAINA.2010.141.

14 A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, 2000. doi:10.1145/365151.365169.

15 A. Ehrenfeucht, J. Kleijn, M. Koutny, and G. Rozenberg. Qualitative and quantitative aspects of a model for processes inspired by the functioning of the living cell. In Evgeny Katz, editor, *Biomolecular Information Processing: From Logic Systems to Smart Sensors and Actuators*, pages 323–331. Wiley, 2012. doi:10.1002/9783527645480.ch16.

16 A. Ehrenfeucht and G. Rozenberg. Reaction systems. *Fundamenta Informaticae*, 76:1–18, 2006. URL: https://content.iospress.com/articles/fundamenta-informaticae/fi75-1-4-15.

17 A. Ehrenfeucht and G. Rozenberg. Reaction systems: a formal framework for processes based on biochemical interactions. *Electronic Communications of the EASST*, 26:1–10, 2010. doi:10.1007/978-3-642-02424-5_3.

18 C. Ferretti, A. Leporati, and L. Manzoni. The many roads to the simulation of reaction systems. *Fundamenta Informaticae*, 171(1-4):175–188, 2020. doi:10.3233/FI-2020-1878.

19 J. Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334(1-3):3–33, 2005. doi:10.1016/j.tcs.2004.11.021.

20 H-J. Kreowski and G. Rozenberg. Graph surfing by reaction systems. In Lambers L. and Weber J., editors, *Graph Transformation. ICGT 2018.*, volume 10887 of *Lecture Notes in Computer Science*, pages 45–62. Springer Berlin, 2018. doi:10.1007/978-3-319-92991-0_4.

21 H-J. Kreowski and G. Rozenberg. Graph transformation through graph surfing in reaction systems. *Journal of Logical and Algebraic Methods in Programming*, 109, 2019. doi:10.1016/j.jlamp.2019.100481.

22 A. Męski, W. Penczek, and G. Rozenberg. Model checking temporal properties of reaction systems. *Information Sciences*, 313:22–42, 2015. doi:10.1016/j.ins.2015.03.048.

23 M.S. Nobile, A.E. Porreca, S. Spolaor, L. Manzoni, P. Cazzaniga, G. Mauri, and D. Besozzi. Efficient simulation of reaction systems on graphics processing units. *Fundamenta Informaticae*, 154(1-4):307—321, 2017. doi:10.3233/FI-2017-1568.

24 E. Shapiro. The family of concurrent logic languages. *ACM Computing Surveys*, 21(3):412–510, September 1989. doi:10.1145/72551.72555.