Natural Computing
https://doi.org/10.1007/s11047-024-09976-3

A framework for monitored dynamic slicing of reaction systems

Linda Brodo[1] · **Roberto Bruni**[2] · **Moreno Falaschi**[3]

Accepted: 30 January 2024

Abstract
Reaction systems (RSs) are a computational framework inspired by biochemical mechanisms. A RS defines a finite set of reactions over a finite set of entities. Typically each reaction has a local scope, because it is concerned with a small set of entities, but complex models can involve a large number of reactions and entities, and their computation can manifest unforeseen emerging behaviours. When a deviation is detected, like the unexpected production of some entities, it is often difficult to establish its causes, e.g., which entities were directly responsible or if some reaction was misconceived. Slicing is a well-known technique for debugging, which can point out the program lines containing the faulty code. In this paper, we define the first dynamic slicer for RSs and show that it can help to detect the causes of erroneous behaviour and highlight the involved reactions for a closer inspection. To fully automate the debugging process, we propose to distil monitors for starting the slicing whenever a violation from a safety specification is detected. We have integrated our slicer in BioResolve, written in Prolog which provides many useful features for the formal analysis of RSs. We define the slicing algorithm for basic RSs and then enhance it for dealing with quantitative extensions of RSs, where timed processes and linear processes can be represented. Our framework is shown at work on suitable biologically inspired RS models.

Keywords Reaction systems · SOS semantics · Program slicing · Assertions · Monitors

1 Introduction

Reaction Systems (RSs) (Ehrenfeucht and Rozenberg 2007b; Brijder et al. 2011a) are a computational framework inspired by systems of living cells. Their constituents are a finite set of entities 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*. RSs have shown to be a general computational model whose application ranges from the modelling of biological phenomena (Azimi et al. 2014; Corolli et al. 2012; Azimi 2017; Barbuti et al. 2016) to molecular chemistry (Okubo and Yokomori 2016). The classical semantics of RSs is defined as a reduction system whose states are sets of entities (those produced at the previous step, possibly joined with others provided by an external context, thus modelling the interaction with the environment). Notably, as reactions exploit facilitation and inhibition mechanisms, the behaviour of RSs is generally non-monotonic, i.e., a computational effect, like the production of a given entity, is not necessarily preserved when more entities are present in the source state.

1.1 Problem statement

Several tools are already available to simulate RSs or to verify that certain properties are met. Besides our own prototype

Linda Brodo, Roberto Bruni and Moreno Falaschi have contributed equally to this work.

✉ Moreno Falaschi
moreno.falaschi@unisi.it

Linda Brodo
brodo@uniss.it

Roberto Bruni
roberto.bruni@unipi.it

[1] Dipartimento di Scienze Economiche e Aziendali, Università di Sassari, Via Muroni 25, 07100 Sassari, Italy

[2] Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy

[3] Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche, Università di Siena, Via Roma 56, 53100 Siena, Italy

Published online: 04 May 2024

BioReSolve[1] proposed in Brodo et al. (2021a, 2023b) and further extended in this paper, some notable examples are, e.g., `brsim`[2] a Basic Reaction System Simulator written in Haskell and distributed under the terms of GNU GPLv3 license (Azimi et al. 2015), its online version `WEBRSIM`[3] that makes all functionalities of `brsim` available through a friendly web interface (Ivanov et al. 2018), the GPU-based Highly Efficient REaction SYstem simulator `HERESY`[4] that exploits the large number of computational units inside GPUs to boost performance (Nobile et al. 2017), the optimised Common Lisp simulator for RSs `cl-rs`[5] presented in Ferretti et al. (2020).

However, the design of RSs for modelling some natural phenomenon is often done by domain experts to validate their hypotheses and require some degree of abstraction. Consequently, inaccuracies or false assumptions may be introduced as early as the design stage. In addition, writing reactions is an error-prone activity, and verifying their execution can be difficult even for RSs with a couple of dozens of entities and reactions. For example, if some mistake is done at the design level and some inexplicable result is observed during the simulation, then a manual inspection of the computation may be necessary in order to understand the nature of the problem.

The aim of this paper is to propose the first automatic technique to ease the debugging of RSs. In fact, we are not aware of any debugging systems available for the above RSs tools.

1.2 The approach

Slicing is the classical technique to circumscribe the most relevant parts of a program that may have caused a certain observable effect. Slicing was first introduced as a static technique by Weiser (1984). Then it was extended in Korel and Laski (1988) by introducing the so called dynamic program slicing, which supports the debugging process by selecting a portion of the program containing the faulty code. Dynamic program slicing has been applied to several programming paradigms (see Silva (2012) for a survey).

The idea explored in this paper is to tailor the slicing technique to RSs and then to automatically trigger the slicing of the computation as soon as certain events are detected. The expected benefit is to discard all irrelevant entities and reactions and to focus the attention on those directly responsible for the observed undesirable effect.

In order to trigger the slicing in a fully automated manner we adapt the monitoring framework in Aceto et al. (2021a) to the setting of RSs. To this aim, first we build on the available Labelled Transition System (LTS) semantics of RSs defined in Brodo et al. (2021a) following the well-known Structural Operational Semantics (SOS) style (Plotkin 1981, 2004). Second, we introduce a flexible modal logic of assertions over transition labels that can be used to express safety conditions. Formulas are then used to synthesize suitable monitors for the LTS semantics of RSs: they inspect the labels as the computation progresses and when a violation is detected they trigger the slicing of the trace from the faulty state.

Our slicing framework follows three main steps as in Falaschi et al. (2016) (there declined for the quite different setting of Concurrent Constraint Programming, which is a monotonic language). First the dynamics of RSs is extended to an enriched semantics that considers states paired with their computation history. Second, we take a partial computation which is considered faulty by the user, or which is determined automatically as faulty by the monitor: the fault is identified by marking a set of entities in the last state reached by the computation. The marked entities are considered responsible of the bug and the goal of the slicing is to determine the causes which led to their production. The third step is the execution of an algorithm that removes from the history any irrelevant information w.r.t. the production of marked entities.

1.3 Contribution

First, we consider a slicing algorithm for ‘closed’ RSs which are RSs where the environment provides just a set of initial entities and then no longer interact at subsequent computation steps. Then, we extend the slicing algorithm to take into account the step-by-step interaction of RSs with the environment, represented by a ‘context’, which can provide new entities at each computation step. In both cases, we show that the sliced computation is a simplification of the original one, which highlights the entities, contexts and reactions that are essential to produce the marked entities. Then we show how to define monitors to fully automate the slicing process.

Finally, we consider slicing for enhanced RSs with quantitative features, such as delays, duration and linearly constrained reactions.

We have developed a prototype implementation in Prolog, freely available online. Here the use of a declarative approach is useful to guarantee the correctness of the implementation, which closely mirrors the theoretical definitions, and also it can be easily modified to accommodate several quantitative extensions.

[1] Available at http://www.di.unipi.it/~bruni/LTSRS/

[2] Available at https://github.com/scolobb/brsim/

[3] Available at https://combio.org/portfolio/webrsim-reaction-system-simulator/

[4] Available at https://github.com/aresio/HERESY/

[5] Available at https://github.com/mnzluca/cl-rs

1.4 Organisation

In Sect. 2 we summarise the basics of RSs. In Sect. 3 we recall a more convenient process algebra for RSs (Brodo et al. 2021a). In Sect. 4 we define the slicing technique for RSs. In Sect. 5 we show a biological example to illustrate our framework and implemented tools. In Sect. 6 we present a specialised assertion language and we introduce monitors. Monitors can be derived from safety formulas in modal logic to mark automatically the entities and start the slicing. In Sect. 7 we recall the definition of extensions of RSs for modelling quantitative information, namely processes with delays and processes with quantities which can be computed by solving linear expressions. Then we show how the slicing algorithm naturally extends to delays and linear quantities. We discuss some related work in Sect. 8 and future work in Sect. 9, together with concluding remarks.

This paper is an extended version of Brodo et al. (2023a). We have added an entire new section to show that our slicing framework is flexible and that our algorithm can be easily modified to deal with previous extensions of RSs such as timed processes and linear processes, which introduce quantitative information in RSs. We have included the correctness proof of our base slicing algorithm. We have added some examples and further explanations to improve the readability of this paper and we have introduced a better and more clear notation to represent the RS computations and their sliced simplifications.

2 Reaction systems

The theory of Reaction Systems (RSs) was born in the field of Natural Computing to model the qualitative behaviour of biochemical reactions in living cells. We briefly account here for the main concepts as introduced in the classical set theoretic presentation of RSs (Brijder et al. 2011a). In the following, we use the term *entities* to denote generic molecular substances (e.g., atoms, ions, molecules) that may be present in the states of a biochemical system.

Let S be a (finite) set of entities; we call a *state* any subset $W \subseteq S$. A *reaction* in S is a triple $r = (R, I, P)$, where $R, I, P \subseteq S$ are finite, non empty sets and $R \cap I = \emptyset$. The sets R, I, P are the sets of *reactants*, *inhibitors*, and *products*, respectively. We let $rac(S)$ denote the set of all reactions over S.

Given a state $W \subseteq S$, the result of a reaction $r = (R, I, P) \in rac(S)$ on W, denoted by $res_r(W)$, is defined as follows, where $en_r(W)$ is called the *enabling predicate* of r:

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

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

From the above definition it follows that all reactants have to be present in the current state for the reaction to take place, while the presence of any of the inhibitors would block the reaction. Products are the outcome of the reaction, to be released in the next state.

A Reaction System is a pair $A = (S, A)$ where S is the set of entities, and $A \subseteq rac(S)$ is a finite set of reactions over S. Given $W \subseteq S$, the result of the reactions A on W, denoted $res_A(W)$, is just the lifting of res_r, i.e., $res_A(W) \triangleq \cup_{r \in A} res_r(W)$.

Since living cells are seen as open systems that react to environmental stimuli, the behaviour of a RS is formalised in terms of an *interactive process*. Let $A = (S, A)$ be a RS and let $n \geq 0$. An $(n+1)$-steps *interactive process* in A is a pair $\pi = (\gamma, \delta)$ s.t. $\gamma = \{C_i\}_{i \in [0,n]}$ is the *context sequence* and $\delta = \{D_i\}_{i \in [0,n]}$ is the *result sequence*, where $C_i, D_i \subseteq S$ for any $i \in [0, n]$, $D_0 = \emptyset$, and $D_{i+1} = res_A(D_i \cup C_i)$ for any $i \in [0, n-1]$. The context sequence γ represents the environment, while the result sequence δ is entirely determined by γ and the set of reactions A. We call $\tau = W_0, \ldots, W_n$ the *state sequence*, with $W_i \triangleq C_i \cup D_i$ for any $i \in [0, n]$. Note that each state W_i in τ is the union of the context C_i at step i and the result set $D_i = res_A(W_{i-1})$ from step $i-1$. Note also that the result of a computation step does not depend on the order of application of the reactions.

Example 1 We consider the simple RS $A \triangleq (S, A)$, where $S \triangleq \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and the set $A \triangleq \{r_1, r_2\}$ contains the reactions $r_1 \triangleq (\{\mathsf{a}\}, \{\mathsf{d}\}, \{\mathsf{b}, \mathsf{c}\})$ and $r_2 \triangleq (\{\mathsf{a}, \mathsf{c}\}, \{\mathsf{d}\}, \{\mathsf{d}\})$, more concisely written as $(\mathsf{a}, \mathsf{d}, \mathsf{bc})$ and $(\mathsf{ac}, \mathsf{d}, \mathsf{d})$. Then, we consider a $3-steps$ interactive process $\pi \triangleq (\gamma, \delta)$, where $\gamma \triangleq C_0, C_1, C_2 = \{\mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}\}$ and $\delta \triangleq D_0, D_1, D_2 = \emptyset, \{\mathsf{b}, \mathsf{c}\}, \{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$. The resulting state sequence is $\tau \triangleq W_0, W_1, W_2 = \{\mathsf{a}, \mathsf{b}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. In fact, it is easy to check that, e.g., $W_0 = C_0$, $D_1 = res_A(W_0) = res_A(\{\mathsf{a}, \mathsf{b}\}) = \{\mathsf{b}, \mathsf{c}\}$ because $en_{r_1}(W_0) \wedge \neg en_{r_2}(W_0)$, and $W_1 = C_1 \cup D_1 = \{\mathsf{a}, \mathsf{b}\} \cup \{\mathsf{b}, \mathsf{c}\} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

3 SOS rules for reaction systems

In order to define our automatic slicing technique, we find it convenient to exploit the algebraic syntax for RSs introduced in Brodo et al. (2021a). Inspired by process algebras such as CCS (Milner 1980), simple SOS inference rules define the behaviour of each operator. This induces a LTS semantics for RSs, where states are terms of the algebra, each transition corresponds to a step of the RS and transition labels retain some information on the entities involved at each step. Transition labels and SOS rules will allow us to pair RSs with monitors and to easily enrich state information with histories, respectively.

Definition 1 (RS processes) Let S be a set of entities. A *RS process* P is any term defined by the following grammar:

$$\mathsf{P}::=[\mathsf{M}] \qquad \mathsf{M}::=(R, I, P) \mid D \mid \mathsf{K} \mid \mathsf{M}|\mathsf{M}$$
$$\mathsf{K}::=\mathbf{0} \mid X \mid C.\mathsf{K} \mid \mathsf{K}+\mathsf{K} \mid \mathsf{rec}\ X.\ \mathsf{K}$$

where $R, I, P \subseteq S$ are non empty sets of entities, $C, D \subseteq S$ are possibly empty set of entities, and X is a process variable.

While in principle reactions, entities and contexts could be seen as components to be handled separately (and in the implementation is more convenient to do so), we mix them together to provide a compositional account of their interactions. A RS process P embeds a *mixture* process M obtained as the parallel composition of some reactions (R, I, P), some set of current entities D (possibly the empty set), and some *context* processes K. We write $\prod_{i\in I} \mathsf{M}_i$ for the parallel composition of all M_i with $i \in I$.

A context process K is a possibly non-deterministic and recursive system: the nil context $\mathbf{0}$ stops the computation; the prefixed context $C.\mathsf{K}$ makes the entities C available to the reactions at the current step, and then leaves K be the context offered at the next step; the non-deterministic choice $\mathsf{K}_1 + \mathsf{K}_2$ allows the context to behave as either K_1 or K_2; X is a process variable, and $\mathsf{rec}\ X.\ \mathsf{K}$ is the usual recursive operator of process algebras. Choice and recursion can combine in-breadth (different evolutions) and in-depth (evolutions of different length, possibly infinite) analysis. The ability to compose contexts in parallel is useful, e.g., to handle different entities with different strategies or to reduce combinatorial explosion at the specification level. For example, letting $\mathsf{K}^?_{\mathsf{a}} \triangleq \mathsf{rec}\ X.\ \mathsf{a}.X + \emptyset.X$ be the context that can recursively offer a or not at any step, the process $\mathsf{K}^?_{\mathsf{a}}|\mathsf{K}^?_{\mathsf{b}}|\mathsf{K}^?_{\mathsf{c}}$ can recursively offer any combination of entities a, b and c.

We say that P and P' are structurally equivalent, written $\mathsf{P} \equiv \mathsf{P}'$, when they denote the same term up to the laws of commutative monoids (unit, associativity and commutativity) for parallel composition $\cdot|\cdot$, with $\emptyset$ as the unit, and the laws of idempotent and commutative monoids for choice $\cdot+\cdot$, with $\mathbf{0}$ as the unit. We also assume $D_1|D_2 \equiv D_1 \cup D_2$ for any $D_1, D_2 \subseteq S$.

Definition 2 (RSs as RS processes) Let $A = (S, A)$ be a RS, and $\pi = (\gamma, \delta)$ an $(n+1)$-steps interactive process in A, with $\gamma = \{C_i\}_{i\in[0,n]}$ and $\delta = \{D_i\}_{i\in[0,n]}$. For any step $i \in [0, n]$, the corresponding RS process $[\![A, \pi]\!]_i$ is defined as follows:

$$[\![A, \pi]\!]_i \triangleq \left[\prod_{r\in A} r \mid D_i \mid \mathsf{K}_{\gamma^i}\right]$$

where the context $\mathsf{K}_{\gamma^i} \triangleq C_i.C_{i+1}.\cdots.C_n.\mathbf{0}$ is the serialisation of the entities offered by γ^i (the shifting of γ at the i-th step). We write $[\![A, \pi]\!]$ as a shorthand for $[\![A, \pi]\!]_0$.

Example 2 The encoding of the RS $A = (S, A)$, in Example 1, is as follows:

$$\mathsf{P}_0 \triangleq [\![A, \pi]\!] = [(\mathsf{a}, \mathsf{d}, \mathsf{bc}) \mid (\mathsf{ac}, \mathsf{d}, \mathsf{d}) \mid \emptyset \mid \mathsf{K}_\gamma]$$
$$\equiv [(\mathsf{a}, \mathsf{d}, \mathsf{bc}) \mid (\mathsf{ac}, \mathsf{d}, \mathsf{d}) \mid \mathsf{ab.ab.ab.}\mathbf{0}]$$

where $\mathsf{K}_\gamma = \{\mathsf{a}, \mathsf{b}\}.\{\mathsf{a}, \mathsf{b}\}.\{\mathsf{a}, \mathsf{b}\}.\mathbf{0}$ is written more concisely as $\mathsf{ab.ab.ab.}\mathbf{0}$. Note that $D_0 = \emptyset$ is inessential and can be discarded thanks to structural congruence.

A transition label ℓ is a tuple $\langle (D, C) \rhd R, I, P\rangle$ with $D, C, R, I, P \subseteq S$. The sets D, C record the entities available in the current state of the system, where C is provided by the context and D is the result from the previous step; the set R records entities whose presence is assumed (either acting as reactants or as inhibitors); the set I records entities whose absence is assumed (either acting as inhibitors or as missing reactants); and the set P records the products of enabled reactions.

The operational semantics of RS processes is defined by the SOS rules in Fig. 1.

The process $\mathbf{0}$ has no transition. The rule (*Ent*) makes available the entities in the (possibly empty) set D, then reduces to $\emptyset$. The rule (*Cxt*) says that a prefixed context process $C.\mathsf{K}$ makes available the entities in the set C and then reduces to K. The rule (*Rec*) is the classical rule for recursion. The rules (*Suml*) and (*Sumr*) select a move of either the left or the right component, resp., discarding the other process. The rule (*Pro*) executes the reaction (R, I, P) (its reactants, inhibitors, and products are recorded in the label), which remains available at the next step together with newly produced entities P. The rule (*Inh*) applies when the reaction (R, I, P) should not be executed; its label records the causes for which the reaction is disabled: possibly some inhibiting entities $(J \subseteq I)$ are present or some reactants $(Q \subseteq R)$ are missing, with $J \cup Q \neq \emptyset$, as at least one cause is needed. The rule (*Par*) puts two processes in parallel by pooling their labels and joining all the set components of the labels. The sanity check $\ell_1 \frown \ell_2$ is required to guarantee that reactants and inhibitors are consistent (see definition below, where we let $W_i \triangleq D_i \cup C_i$ for brevity):

$$\langle (D_1, C_1) \rhd R_1, I_1, P_1\rangle \frown \langle (D_2, C_2) \rhd R_2, I_2, P_2\rangle$$
$$\triangleq (W_1 \cup W_2 \cup R_1 \cup R_2) \cap (I_1 \cup I_2) = \emptyset$$

In the conclusion of rule (*Par*) we write $\ell_1 \cup \ell_2$ for the component-wise union of labels (see definition below, where the notation $X_{1,2} \triangleq X_1 \cup X_2$ is used):

$$\langle (D_1, C_1) \rhd R_1, I_1, P_1\rangle \cup \langle (D_2, C_2) \rhd R_2, I_2, P_2\rangle$$
$$\triangleq \langle (D_{1,2}, C_{1,2}) \rhd R_{1,2}, I_{1,2}, P_{1,2}\rangle$$

Fig. 1 SOS semantics of the RS processes

$$\frac{}{D \xrightarrow{\langle (D,\emptyset)\rhd \emptyset,\emptyset,\emptyset\rangle} \emptyset}\ (Ent) \qquad \frac{}{C.\mathsf{K} \xrightarrow{\langle (\emptyset,C)\rhd \emptyset,\emptyset,\emptyset\rangle} \mathsf{K}}\ (Cxt)$$

$$\frac{\mathsf{K}_1 \xrightarrow{\ell} \mathsf{K}_1'}{\mathsf{K}_1+\mathsf{K}_2 \xrightarrow{\ell} \mathsf{K}_1'}\ (Suml) \qquad \frac{\mathsf{K}_2 \xrightarrow{\ell} \mathsf{K}_2'}{\mathsf{K}_1+\mathsf{K}_2 \xrightarrow{\ell} \mathsf{K}_2'}\ (Sumr) \qquad \frac{\mathsf{K}[^{\mathsf{rec}\ X.\mathsf{K}}/_X] \xrightarrow{\ell} \mathsf{K}'}{\mathsf{rec}\ X.\ \mathsf{K} \xrightarrow{\ell} \mathsf{K}'}\ (Rec)$$

$$\frac{}{(R,I,P) \xrightarrow{\langle (\emptyset,\emptyset)\rhd R,I,P\rangle} (R,I,P) \mid P}\ (Pro) \qquad \frac{J \subseteq I \quad Q \subseteq R \quad J \cup Q \neq \emptyset}{(R,I,P) \xrightarrow{\langle (\emptyset,\emptyset)\rhd J,Q,\emptyset\rangle} (R,I,P)}\ (Inh)$$

$$\frac{\mathsf{M}_1 \xrightarrow{\ell_1} \mathsf{M}_1' \quad \mathsf{M}_2 \xrightarrow{\ell_2} \mathsf{M}_2' \quad \ell_1 \frown \ell_2}{\mathsf{M}_1 \mid \mathsf{M}_2 \xrightarrow{\ell_1 \cup \ell_2} \mathsf{M}_1' \mid \mathsf{M}_2'}\ (Par) \qquad \frac{\mathsf{M} \xrightarrow{\langle (D,C)\rhd R,I,P\rangle} \mathsf{M}' \quad R \subseteq D \cup C}{[\mathsf{M}] \xrightarrow{\langle (D,C)\rhd R,I,P\rangle} [\mathsf{M}']}\ (Sys)$$

Finally, the rule (*Sys*) requires that all the processes of the systems have been considered, and also checks that all the needed reactants are actually available in the system ($R \subseteq D \cup C$). In fact this constraint can only be met on top of all processes. The check that inhibitors are absent ($I \cap (D \cup C) = \emptyset$) is embedded in rule (*Par*). In the following we assume transitions $\mathsf{P} \xrightarrow{\langle (D,C)\rhd R,I,P\rangle} \mathsf{P}'$ guarantee that any instance of the rule (*Inh*) is applied in a way that maximises the sets J and Q (see Brodo et al. 2021a).

Example 3 Let us consider the RS process $\mathsf{P}_0 \triangleq [\mathsf{M} \mid \mathsf{ab.ab.ab.}\mathbf{0}]$ from Example 2, where we let $\mathsf{M} \triangleq (\mathsf{a, d, bc}) \mid (\mathsf{ac, d, d})$ for brevity. The process P_0 has a unique outgoing transition:

$$\mathsf{P}_0 = [\mathsf{M} \mid \mathsf{ab.ab.ab.}\mathbf{0}] \xrightarrow{\langle (\emptyset,\mathsf{ab})\rhd \mathsf{a,cd,bc}\rangle} [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{ab.ab.}\mathbf{0}].$$

Letting $\mathsf{P}_1 \triangleq [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{ab.ab.}\mathbf{0}]$, there is also a unique outgoing transition from P_1:

$$\mathsf{P}_1 = [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{ab.ab.}\mathbf{0}] \xrightarrow{\langle (\mathsf{bc,ab})\rhd \mathsf{ac,d,bcd}\rangle} [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{d} \mid \mathsf{ab.}\mathbf{0}].$$

Finally, letting $\mathsf{P}_2 \triangleq [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{d} \mid \mathsf{ab.}\mathbf{0}]$, there is also a unique outgoing transition from P_2:

$$\mathsf{P}_2 = [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{d} \mid \mathsf{ab.}\mathbf{0}] \xrightarrow{\langle (\mathsf{bcd,ab})\rhd \mathsf{d},\emptyset,\emptyset\rangle} [\mathsf{M} \mid \mathbf{0}].$$

As the process $\mathsf{P}_3 \triangleq [\mathsf{M} \mid \mathbf{0}]$ contains the stopping context $\mathbf{0}$, it has no outgoing transition.

The following theorem from Brodo et al. (2021a) shows how the set-theoretic dynamics of a RS matches the SOS semantics of its RS process.

Theorem 1 *(see Brodo et al. 2021a) Let $A = (S, A)$ be a RS, and $\pi = (\gamma, \delta)$ an $(n+1)$-steps interactive process in A, with $\gamma = \{C_i\}_{i\in[0,n]}$, $\delta = \{D_i\}_{i\in[0,n]}$, and let $\mathsf{P}_i \triangleq [\![A, \pi]\!]_i$ for any $i \in [0, n]$. Then, for any $i \in [0, n-1]$:*

1. *if $\mathsf{P}_i \xrightarrow{\langle (D,C)\rhd R,I,P\rangle} \mathsf{P}$ then $D = D_i$, $C = C_i$, $P = D_{i+1}$ and $\mathsf{P} \equiv \mathsf{P}_{i+1}$;*
2. *there exist $R, I \subseteq S$ such that $\mathsf{P}_i \xrightarrow{\langle (D_i,C_i)\rhd R,I,D_{i+1}\rangle} \mathsf{P}_{i+1}$.*

4 Slicing RS computations

Dynamic slicing is a technique that helps to debug a program by simplifying a partial execution trace, thus depurating it from parts which are irrelevant to finding the bug. It can also help to highlight parts of the program which have been wrongly ignored by the execution.

4.1 SOS rules for the slicing computation

The slicing algorithm, presented in the next section, needs a slightly different state configuration that includes the whole past state sequence $W_i = C_i \cup D_i$. To carry the relevant information, we enrich the state configuration [M] by prefixing it with the list of the previous result and context sets, written $\frac{D_0}{C_0}\frac{D_1}{C_1}\cdots\frac{D_n}{C_n}[\mathsf{M}]$.

First, we extend the syntax of RS processes as follows:

$$\mathsf{P} ::= [\mathsf{M}] \mid \tfrac{D}{C}\mathsf{P}$$

The next step consists in enriching the operational semantics to deal with histories. We add the new rule (*Hist*) and modify the (*Sys*) inference rule in Fig. 1 as below:

$$\frac{\mathsf{P} \xrightarrow{\ell} \mathsf{P}'}{\frac{D}{C}\mathsf{P} \xrightarrow{\ell} \frac{D}{C}\mathsf{P}'}\ (Hist) \qquad \frac{\mathsf{M} \xrightarrow{\langle (D,C)\rhd R,I,P\rangle} \mathsf{M}' \quad R \subseteq D \cup C}{[\mathsf{M}] \xrightarrow{\langle (D,C)\rhd R,I,P\rangle} \frac{D}{C}[\mathsf{M}']}\ (Sys)$$

The formal Definition 2 is thus updated to carry on the history of the computation.

Definition 3 (RS processes with history) Let $A = (S, A)$ be a RS, and $\pi = (\gamma, \delta)$ an $(n+1)$-steps interactive process in A, with $\gamma = \{C_i\}_{i\in[0,n]}$ and $\delta = \{D_i\}_{i\in[0,n]}$. For any step $i \in [0, n]$, the corresponding new process configuration

$[\![A, \pi]\!]_i$ is defined as follows:

$$[\![A, \pi]\!]_i \triangleq \frac{D_0}{C_0}\frac{D_1}{C_1}\cdots\frac{D_{i-1}}{C_{i-1}}\left[\prod_{r\in A} r \mid D_i \mid \mathsf{K}_{\gamma^i}\right].$$

Example 4 Let us revisit the computation in Example 3 to take histories into account. The corresponding sequence of transitions is thus:

$$\mathsf{P}_0 \xrightarrow{\langle(\emptyset,\mathsf{ab})\triangleright\mathsf{a},\mathsf{cd},\mathsf{bc}\rangle} \frac{\emptyset}{\mathsf{ab}}\mathsf{P}_1 \xrightarrow{\langle(\mathsf{bc},\mathsf{ab})\triangleright\mathsf{ac},\mathsf{d},\mathsf{bcd}\rangle} \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\mathsf{P}_2 \xrightarrow{\langle(\mathsf{bcd},\mathsf{ab})\triangleright\mathsf{d},\emptyset,\emptyset\rangle} \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\frac{\mathsf{bcd}}{\mathsf{ab}}\mathsf{P}_3.$$

where we recall that $\mathsf{P}_0 \triangleq [\mathsf{M} \mid \mathsf{ab.ab.ab.}\mathbf{0}]$, $\mathsf{P}_1 \triangleq [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{ab.ab.}\mathbf{0}]$, $\mathsf{P}_2 \triangleq [\mathsf{M} \mid \mathsf{b} \mid \mathsf{c} \mid \mathsf{d} \mid \mathsf{ab.}\mathbf{0}]$ and $\mathsf{P}_3 \triangleq [\mathsf{M} \mid \mathbf{0}]$ with $\mathsf{M} \triangleq (\mathsf{a}, \mathsf{d}, \mathsf{bc}) \mid (\mathsf{ac}, \mathsf{d}, \mathsf{d})$.

4.2 Slicing algorithms

In the following we assume that the reactions are numbered consecutively by positive integer numbers, and denote the j-th reaction in the RS by the notation r_j. As a matter of notation, please notice that each history $\frac{D_0}{C_0}\cdots\frac{D_m}{C_m}$ computed in m steps by Definition 3, defines a *trace* $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$ on which we perform the slicing computation, where N_i is the set of reactions applied in the i-th computation step. Here each reaction is simply represented by its numeric position in the list of reactions, i.e., $N_i = \{j \mid en_{r_j}(D_{i-1} \cup C_{i-1})\}$ for any $i \in [1, m]$. Abusing the notation, in the following we write $r_j \in N$ whenever $j \in N$.

Example 5 The trace that corresponds to the computation in Example 4 is just:

$$\frac{\emptyset}{\mathsf{ab}} \xrightarrow{\{r_1\}} \frac{\mathsf{bc}}{\mathsf{ab}} \xrightarrow{\{r_1,r_2\}} \frac{\mathsf{bcd}}{\mathsf{ab}}.$$

Our slicing technique consists of three main steps.

Enriched Semantics (Step S1). The slicing process requires some extra information from the execution of the processes. More precisely, (1) at each operational step we need to highlight the reactions that have been applied; and (2) we need to determine the part of the context which adds to the previous state the entities which are necessary to produce the marked entities in the following state. For solving (1) and (2), in Sect. 4.1 we have introduced an enriched semantics that records computation sequences. We need to keep track of the state sequence of the computation for the slicing process, by keeping separated the produced entities D_i in a computation step from the context C_i.

Marking the state (Step S2). Let us suppose that the final configuration in a partial computation is $\frac{D_m}{C_m}$. The user selects a subset $D_{sliced} \subseteq D_m$ that may explain the (wrong) behaviour of the program. In Sect. 6 we describe an assertion language and monitors to automatize the selection.

Trace Slice (Step S3). Starting from the user's selection $\frac{D_{sliced}}{\emptyset}$, we define a backward slicing step. Roughly, this step allows us to eliminate from the execution trace all the information not related to D_{sliced}. Starting from this sliced final state and proceeding backwards we can compute for each computation step the information which is relevant to produce the marked elements in the final state. At the generic step $i \in [1, m]$ we assume the marked entities $\frac{D'_i}{C'_i}$ are already available and infer the entities $\frac{D'_{i-1}}{C'_{i-1}}$ to be marked at step $i-1$. Notably, the set C'_i is irrelevant for computing $\frac{D'_{i-1}}{C'_{i-1}}$ (only the component D'_i matters).

Input: - a trace $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$
- a marking $D_{sliced} \subseteq D_m$

Output: a sliced trace $\frac{D'_0}{C'_0} \xrightarrow{N'_1} \cdots \xrightarrow{N'_m} \frac{D'_m}{C'_m} = \frac{D'_{sliced}}{\emptyset}$

```
1  begin
2      let D'_m = D_sliced ∧ C'_m = ∅
3      for i = m to 1 do
4          let D'_{i-1} = ∅ ∧ C'_{i-1} = ∅ ∧ N'_i = ∅
5          for j ∈ N_i where r_j = (R_j, I_j, P_j), such that (D'_i ∩ P_j ≠ ∅) do
6              let N'_i = N'_i ∪ {j}
7              if en_{r_j}(D_{i-1}) then
8                  let D'_{i-1} = D'_{i-1} ∪ R_j
9              else
10                 let C'_{i-1} = C'_{i-1} ∪ (R_j \ D_{i-1})
11                 let D'_{i-1} = D'_{i-1} ∪ (R_j ∩ D_{i-1})
12             end
13         end
14     end
15 end
```

Algorithm 1: Trace Slicer for context dependent computations

Input: - a trace $D_0 \xrightarrow{N_1} \cdots \xrightarrow{N_m} D_m$
- a marking $D_{sliced} \subseteq D_m$

Output: a sliced trace $D'_0 \xrightarrow{N'_1} \cdots \xrightarrow{N'_m} D'_m = D_{sliced}$

```
1  begin
2      let D'_m = D_sliced
3      for i = m to 1 do
4          let D'_{i-1} = ∅ ∧ N'_i = ∅
5          for j ∈ N_i where r_j = (R_j, I_j, P_j), such that D'_i ∩ P_j ≠ ∅ do
6              let N'_i = N'_i ∪ {j}
7              let D'_{i-1} = D'_{i-1} ∪ R_j
8          end
9      end
10 end
```

Algorithm 2: Trace Slicer for context independent computations

4.2.1 Marking algorithms

Let us explain how the slicing Algorithm 1 works.

Our algorithm returns a sliced trace which contains only the (usually rather small) subset of the entities which are nec-

essary for deriving the marked entities. Let us now describe it informally. Let us consider first the more complex case of context dependent computations. First of all the user has to indicate the subset D_{sliced} of the entities in the last state of the computation D_m that she wants to mark. Then the backward slicing process can start. Now let us consider the iteration i of the slicer. Marking the relevant information in previous state $\frac{D_{i-1}}{C_{i-1}}$ requires analyzing the rules which have been applied at step $i-1$. So, if $r_j \in N_{i-1}$ and $r_j = (R_j, I_j, P_j)$ then we need to check if r_j produces at least one entity which is marked in the next state. If this is the case, then j is added to the set of marked rules. Then it is necessary to check if the context C_{i-1} was essential for applying rule r_j, or if all necessary entities were already included in D_{i-1}. Thus it is necessary to compute the entities in C_{i-1} which are missing in D_{i-1} in order for rule r_j to be enabled, and those entities are marked in (added to) context C'_{i-1}. The elements in R_j are added to the marked entities in D'_{i-1}. For the computations which are context independent the part of transformation which is related to the context can clearly be eliminated (see Algorithm 2).

Example 6 Let us consider the trace in Example 5:

$$\frac{\emptyset}{\mathsf{ab}} \xrightarrow{\{r_1\}} \frac{\mathsf{bc}}{\mathsf{ab}} \xrightarrow{\{r_1,r_2\}} \frac{\mathsf{bcd}}{\mathsf{ab}}.$$

and suppose we let $D_{sliced} = \{\mathsf{d}\}$, i.e., we mark the entity d in the target state. The slicing algorithm returns the sliced trace:

$$\frac{\emptyset}{\mathsf{a}} \xrightarrow{\{r_1\}} \frac{\mathsf{c}}{\mathsf{a}} \xrightarrow{\{r_2\}} \frac{\mathsf{d}}{\emptyset}.$$

In the following we will sometimes illustrate the sliced trace as embedded in the original trace, by boxing the sliced entities and reactions:

$$\frac{\emptyset}{\boxed{\mathsf{a}}\mathsf{b}} \xrightarrow{\{\boxed{r_1}\}} \frac{\mathsf{b}\boxed{\mathsf{c}}}{\boxed{\mathsf{a}}\mathsf{b}} \xrightarrow{\{r_1,\boxed{r_2}\}} \frac{\mathsf{bc}\boxed{\mathsf{d}}}{\mathsf{ab}}.$$

The following proposition states what is kept or removed by the slicing algorithm at each step.

Proposition 2 *Let $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$ be a context dependent computation and let $\frac{D'_0}{C'_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D'_m}{C'_m}$ be the sliced trace corresponding to a given marking $D_{sliced} = D'_m \subseteq D_m$ (with $C'_m = \emptyset$). Then, we have all of the following:*

1. *$\forall i \in [1,m]$, $r_j = (R_j, I_j, P_j) \in N'_i$ iff $r_j \in N_i$ and there exists $e \in D'_i$ s.t. $e \in P_j$.*
2. *$\forall i \in [0, m-1]$, $e \in D'_i$ iff there exists $r_j = (R_j, I_j, P_j) \in N'_{i+1}$ such that $e \in D_i \cap R_j$.*
3. *$\forall i \in [0, m-1]$, $e \in C'_i$ iff there exists $r_j = (R_j, I_j, P_j) \in N'_{i+1}$ such that $e \in (C_i \cap R_j) \setminus D_i$.*

Proof The proof is by induction on the number n of computation steps.

Base case $n = 1$)

Let us first prove property (1).

Let the marked subset of D_1 be the set $D'_1 \subseteq D_1$ as defined in line 2 of Algorithm 1. Then, by line 5, for a reaction $r_j = (R_j, I_j, P_j) \in N_1$, we have that: $r_j \in N'_1$ (by line 6) iff $D'_1 \cap P_j \neq \emptyset$ (by line 5) iff there exists $e \in D'_1$ s.t. $e \in P_j$. Thus, property (1) holds.

Let us now prove property (2).

In Algorithm 1, D'_0 is initialised to the empty set in line 4. Thus, we have that an entity $e \in D'_0$ iff (by line 7 we have two cases) (a) r_j is enabled on D_0, which means that $R_j \subseteq D_0$, hence $D'_0 = D'_0 \cup R_j = D'_0 \cup (D_0 \cap R_j)$, and the property holds, or (b) if $\neg en_{r_j}(D_0)$ then $D'_0 = D'_0 \cup \{D_0 \cap R_j\}$ and hence the property holds.

Let us now prove property (3).

By line 4, $C'_0 = \emptyset$. Then a new entity e can be added to C'_0 only by line 10 and only if the conditions in line 5 holds but not the one in line 7. Hence, by line 5, $j \in N_0$, with $r_j = (R_j, I_j, P_j)$, and, by line 6, $r_j \in N'_1$. By line 10, we derive that $C'_0 = C'_0 \cup (R_j \setminus D_0)$. Now let us consider $e \in R_j \setminus D_0$. By line 5, r_j is enabled in this (current) first step, hence, by definition, $e \in R_j \subseteq D_0 \cup C_0$. By line 10 $\neg en_{r_j}(D_0)$, and hence $R_j \setminus D_0 \subseteq C_0$ must be non empty and $R_j \setminus D_0 \subseteq C_0 \cap R_j$ and hence for all $e \in R_j \setminus D_0 \subseteq C_0 \cap R_j$ we have that $e \in C'_0$ (by line 10), and $e \notin D_0$, and the property holds.

Inductive case $n > 1$)

We start by considering the step $\frac{D_{n-1}}{C_{n-1}} \xrightarrow{N_n} \frac{D_n}{C_n}$. In the first iteration of the **for** statement in line 5 in Algorithm 1, thus with $i = n-1$, we can show that properties (1-3) hold. Thus we prove that property (1) holds for $i = n$, and properties (2) and (3) hold for $i = n-1$.

Let us first prove property (1).

Let the marked subset of D_n be the set $D_{sliced} = D'_n \subseteq D_n$ as defined in line 2 of Algorithm 1. Then, by line 5, for a reaction $r_j = (R_j, I_j, P_j) \in N_n$, we have that: $r_j \in N'_n$ (by line 6) iff $D'_n \cap P_j \neq \emptyset$ (by line 5) iff there exists $e \in D'_n$ s.t. $e \in P_j$. Thus, property (1) holds for the last computation step.

Let us now prove property (2).

In Algorithm 1, D'_{n-1} is initialised to the empty set in line 4. Thus, we have that an entity $e \in D'_{n-1}$ iff (by line 7 we have two cases) (a) r_j is enabled on D_{n-1}, which means that $R_j \subseteq D_{n-1}$. Hence $D'_{n-1} = D'_{n-1} \cup R_j = D'_{n-1} \cup (D_{n-1} \cap R_j)$, and the property holds. (b) if $\neg en_{r_j}(D_{n-1})$

then $D'_{n-1} = D'_{n-1} \cup (D_{n-1} \cap R_j)$ and hence the property holds.

Let us now prove property (3).

By line 4, $C'_{n-1} = \emptyset$. Then a new entity e can be added to C'_{n-1} only by line 10 and only if the conditions in line 5 holds but not the one in line 7. Hence, by line 5, $j \in N_n$, with $r_j = (R_j, I_j, P_j)$, and, by line 6, $r_j \in N'_n$. By line 10, we derive that $C'_0 = C'_0 \cup (R_j \setminus D_0)$. Now let us consider $e \in R_j \setminus D_{n-1}$. By line 5, r_j is enabled in this (current) first step, hence, by definition, $e \in R_j \subseteq D_{n-1} \cup C_{n-1}$. By line 7 $\neg en_{r_j}(D_{n-1})$, and hence $R_j \setminus D_{n-1} \subseteq C_{n-1}$ must be non empty and $R_j \setminus D_{n-1} \subseteq C_{n-1} \cap R_j$ and hence for all $e \in R_j \setminus D_{n-1} \subseteq C_{n-1} \cap R_j$ we have that $e \in C'_{n-1}$ (by line 10), and $e \notin D_{n-1}$, and the property holds for the last computation step. Now we consider the computation $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_{n-1}} \frac{D_{n-1}}{C_{n-1}}$ w.r.t. the marking D'_{n-1} computed in the last computation step (see line 7 of Algorithm 1). By line 5 the set R_j in line 10 is a subset of D_{n-1}, and hence D'_{n-1} is a subset of D_{n-1} and determines a marking D_{sliced} for the trace $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_{n-1}} \frac{D_{n-1}}{C_{n-1}}$, then the property follows by the inductive hypothesis, as the number of steps for the remaining computation is $n - 1 < n$. □

Proposition 2 addresses context independent computations when contexts are empty.

5 Implementation

In this section we show how to check a biological model by our slicing methodology. The implementation is available on-line,[6] with a small manual to use it. It extends the tool BioReSolve,[7] which already provided a friendly environment for simulation, analysis and verification of RSs. The tool has been developed and tested under SWI-Prolog and exploits a few library predicates for efficiency reasons. DCG Grammar rules are used to ease the writing of RS specifications. Its features include the possibility to simulate single traces or generate the whole graph of the LTS (where user defined predicates can be used to color each node depending on its content so to improve readability), to verify modal logic formulas, to check bisimulation based equivalences between different RSs, to deal with quantitative aspects of RSs, such as delays and duration for produced entities and linear handling of concentration levels (see Brodo et al. 2021a, 2021c for more details).

For a computation which does not use assertions the interpreter gives the users some choices: (1) whether they want to make a context independent computation; (2) the possibility to specify the maximum number m of computation steps.

The interpreter will show the corresponding trace $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$, emphasizing the elements D_i, C_i, N_i. Then the users have to provide the entities that they wantsto mark in D_m. Finally, the interpreter will compute the corresponding sliced computation and present it. Let us see one example for illustrating our tool.

Example 7 We consider a RS defined in Barbuti et al. (2021), to model a network for gene regulation. These networks represent the interactions among genes regulating the activation of specific cell functions. The RS models a fragment of the network for controlling the process of differentiation of T helper (Th) lymphocytes, which play a fundamental role in the immune system. We introduced on purpose one wrong reaction in the RS model that can be found in our tool website[8] and performed some experiments. For instance we made a context free computation, starting from the initial state containing only the entity `ifngammah`. The computation, limited to 6 steps, produced the following sequence of states.

```
[ [ifngammah], [ifngammarh], [stat1h],
    [socs1,socs2], [tbeth],
  [ifngammah,socs1,tbeth],
    [ifngammah,ifngammarm,socs1,tbeth] ]
```

Now we wanted to focus on the molecule `tbeth` in the last computation state, and hence we used our slicer, marking `tbeth`. The sliced trace in outcome was the following:

```
[ [ifngammah], [ifngammarh], [stat1h],
   [socs1], [tbeth],
  [socs1,tbeth], [tbeth] ]
```

with the following sequence of reaction numbers applied in the steps:

```
[ [5], [8], [19], [18], [27,29], [18,27] ]
```

The sliced sequence can now easily be interpreted. Clearly `[tbeth]` was produced as a result of the application of four reaction rules in a sequence, those with numbers `[5],[8],[19],[18]`. Then reaction `[27]` reintroduced `tbeth` in each following step. So, the sliced sequence produced a much simpler trace, with an easy interpretation. It is now immediate to see that `[tbeth]` was introduced by rule `[18]`, due to the reactant `[socs1]`, which is recognised as an error, because `[tbeth]` should be introduced by `[stat1h]`. Thus the user can now correct reaction `[18]` for `[stat1h]`.

[6] http://www.di.unipi.it/~bruni/LTSRS/slicingBioReSolve.zip

[7] http://www.di.unipi.it/~bruni/LTSRS/

[8] http://www.di.unipi.it/~bruni/LTSRS/wrongspec.pl

6 A logical framework for the slicing algorithm

In this section we try to further automate the slicing process whenever the user can describe a property to be checked along a computation: the computation is stopped as soon as the current state S does not satisfy the property. Then slicing starts from S, which is automatically marked.

To specify properties we reuse the simple assertion language introduced in Brodo et al. (2021a), which is tailored on the labels of the LTS generated by SOS semantics in Sect. 3. Then, we rely on a fragment of the recursive extension of the Hennessy-Milner Logic, called sHML (Aceto et al. 2021a), to formally express properties to be verified along a RS process execution. We exploit the monitor technique from Aceto et al. (2020) to check if the current state of the RS process execution respects or not the required property. To this aim, we apply the translation from sHML formula to monitors given in Aceto et al. (2021a). Some modifications are required as in the original proposal sHML logic works on action names, whereas we work with logic assertions.

6.1 The assertion language

The labels of our LTS carry on a large amount of information about the activity executed during each transition, our assertion is a formula that predicates on those labels. Hereafter, we assume that the context can be non-deterministic.

Example 8 Here follows an example of some properties which we may verify:

Has the reaction (ab, c, b) been applied ?

Has the entity a played both as reactant and as product ?

Definition 4 [Assertion Language] Given a set of entities S, assertions F on S are built from the following syntax, where $E \subseteq S$ and $Pos \in \{\mathcal{D}, \mathcal{C}, \mathcal{W}, \mathcal{R}, \mathcal{I}, \mathcal{P}\}$:

$$\mathsf{F} ::= \mathbf{tt} \mid E \subseteq Pos \mid ? \in Pos \mid \mathsf{F} \vee \mathsf{F} \mid \mathsf{F} \wedge \mathsf{F} \mid \neg\mathsf{F}$$

Pos distinguishes different positions in the labels: $\mathcal{D}$ stands for entities produced in the previous transition, $\mathcal{C}$ for entities provided by the context, $\mathcal{W}$ for their union, $\mathcal{R}$ for reactants, $\mathcal{I}$ for inhibitors, and $\mathcal{P}$ for products. An assertion $E \subseteq Pos$, checks the membership of a subset of entities E in a given Pos, $? \in Pos$ is a test of non-emptiness of Pos, $\mathsf{F}_1 \vee \mathsf{F}_2$ denotes a disjunction, $\mathsf{F}_1 \wedge \mathsf{F}_2$ is a conjunction, $\neg\mathsf{F}$ is a negation.

Definition 5 [Satisfaction of Assertion] Let P be a RSs process, let $\ell = \langle (D, C) \rhd R, I, P \rangle$ be a transition label, and F be an assertion. We write $\ell \models \mathsf{F}$ (read as the transition label ℓ satisfies the assertion F) if and only if the following hold, where $\mathsf{select}(\ell, Pos)$ extract the desired component from the label ℓ:

$$\begin{array}{ll}
\ell \models E \subseteq Pos & \text{iff } E \subseteq \mathsf{select}(\ell, Pos) \\
\ell \models ? \in Pos & \text{iff } \mathsf{select}(\ell, Pos) \neq \emptyset \\
\ell \models \mathsf{F}_1 \vee \mathsf{F}_2 & \text{iff } \ell \models \mathsf{F}_1 \vee \ell \models \mathsf{F}_2 \\
\ell \models \mathsf{F}_1 \wedge \mathsf{F}_2 & \text{iff } \ell \models \mathsf{F}_1 \wedge \ell \models \mathsf{F}_2 \\
\ell \models \neg\mathsf{F} & \text{iff } \ell \not\models \mathsf{F}
\end{array}$$

$$\mathsf{select}(\langle (D, C) \rhd R, I, P \rangle, Pos) \triangleq \begin{cases} D & \text{if } Pos = \mathcal{D}, \\ C & \text{if } Pos = \mathcal{C}, \\ D \cup C & \text{if } Pos = \mathcal{W}, \\ R & \text{if } Pos = \mathcal{R}, \\ I & \text{if } Pos = \mathcal{I}, \\ P & \text{if } Pos = \mathcal{P} \end{cases}$$

Example 9 Some assertions matching the queries listed in Example 8 are:

- $\mathsf{F}_1 \triangleq \{\mathsf{ab}\} \subseteq \mathcal{R} \wedge \{\mathsf{c}\} \subseteq \mathcal{I}$, while $\mathsf{F}'_1 \triangleq \neg\mathsf{F}_1$ is verified if (ab, c, b) is not applied,
- $\mathsf{F}_2 \triangleq \{\mathsf{a}\} \subseteq \mathcal{R} \wedge \{\mathsf{a}\} \subseteq \mathcal{P}$.

6.2 Monitors

Differently from Aceto et al. (2020), in our context monitors check if transition labels satisfy a given property. A process monitor stops when a verdict is reached, thus we omit $\mathbf{0}$. The $\mathbf{no}_s$ verdict is equipped with a set of entities, $s \subseteq S$, used as markers for the slicing.

Definition 6 A monitor process is defined by the grammar:

$$m, n \in \mathsf{Mon} ::= \mathbf{no}_s \mid \mathbf{yes} \mid \mathsf{F}.m \mid m + n \mid m \otimes n \mid \mathsf{rec}\, X.m \mid X$$

where X comes from a countably infinite set of monitor variables, and the set $s \subseteq S$.

The syntax of a monitor is similar to that of a context process: actions are replaced by properties to be verified by the process action. A 'verdict' can be **yes** or $\mathbf{no}_s$ for acceptance or rejection respectively. Sum $m + n$ is used to provide monitors with different behaviours, while $m \otimes n$ is used to compose monitors in parallel.

The semantics is in Fig. 2, where the symmetric rules are omitted. The set of transition labels is composed by the set of the formulas of the assertion language in Definition 4 plus a special silent action τ. The verdicts do nothing.

Definition 7 A monitored system is a monitor $m \in \mathsf{Mon}$ and a process $p \in \mathsf{Proc}$ that run side-by-side, denoted $m \lhd p$. The behaviour of a monitored system is defined by the inference rules in Table 1.

If a monitored system $m \lhd p$ reaches a verdict like $\mathbf{no}_s \lhd q$, then a violation is detected and q is the state where the slicing starts by marking the entities in the set s.

Fig. 2 SOS semantics of the monitors

$$\frac{}{\mathsf{F}.m \xrightarrow{\mathsf{F}} m}\ (Pro) \qquad \frac{}{\mathbf{yes} \otimes m \xrightarrow{\tau} m}\ (Ver1) \qquad \frac{}{\mathbf{no}_s \otimes m \xrightarrow{\tau} \mathbf{no}_s}\ (Ver2) \qquad \frac{m \xrightarrow{\tau} m'}{m \otimes n \xrightarrow{\tau} m' \otimes n}\ (Par1)$$

$$\frac{m_1 \xrightarrow{\mathsf{F}} m_1'}{m_1 + m_2 \xrightarrow{\mathsf{F}} m_1'}\ (Sum) \qquad \frac{m[^{\mathsf{rec}\ X.m}/_X] \xrightarrow{\mathsf{F}} m'}{\mathsf{rec}\ X.\ m \xrightarrow{\mathsf{F}} m'}\ (Rec) \qquad \frac{m \xrightarrow{\mathsf{F}_1} m' \qquad n \xrightarrow{\mathsf{F}_2} n'}{m \otimes n \xrightarrow{\mathsf{F}_1 \wedge \mathsf{F}_2} m' \otimes n'}\ (Par2)$$

Table 1 Monitored systems

$$\frac{p \xrightarrow{\ell} p' \qquad m \xrightarrow{\mathsf{F}} m' \qquad \ell \vDash \mathsf{F}}{m \lhd p \xrightarrow{(\ell,\mathsf{F})} m' \lhd p'}\ (Exec) \qquad \frac{m \xrightarrow{\tau} m' \quad p}{m \lhd p \xrightarrow{\tau} m' \lhd p}\ (\tau)$$

6.3 Monitorability

Typically we are interested in verifying if certain assertions are met along a process execution. Writing the corresponding monitors is an error prone task. Following Aceto et al. (2021a), we prefer to write property specifications as formulas in a fragment of the (recursive) Hennessy-Milner logic, called sHML. The syntax and semantics of sHML are reported in Table 2. For example, given a certain assertion F (see Definition 4) we can write the sHML formula $\mathsf{max}\ X.([\mathsf{F}].X \wedge [\neg\mathsf{F}].\mathbf{ff})$ to specify that "the computation should always exhibit transition labels satisfying F and stops as soon as a violation is detected". Of course, other properties can be required, e.g. that F_1 and F_2 are satisfied in alternation. Following the monitor synthesis in Aceto et al. (2021b), and recalled in Table 3, we obtain that, after some logical simplifications, a monitor implementing the previous formula is: $\mathsf{m} = \mathsf{rec}\ X.(\neg\mathsf{F}.X + \mathsf{F}.\mathbf{no})$. While it may seem that monitors closely resemble sHML formulas, we argue that the box modality $[\mathsf{F}].\phi$ is much more convenient to write and to manage than the sum $\mathsf{F}.\mathsf{m}(\phi) + \neg\mathsf{F}.\mathbf{yes}$.

Example 10 Let P_0 be the process in Examples 3–4. We want to study a computation where all the visited states satisfy the following assertion: $\mathsf{F} \triangleq \{\mathsf{b}\} \subseteq \mathcal{C} \wedge \{\mathsf{d}\} \nsubseteq \mathcal{R}$. Then we need a sHML formula of the format described above: $\phi \triangleq \mathsf{max}\ X.([\mathsf{F}].X \wedge [\neg\mathsf{F}]\mathbf{ff}_{\{\mathsf{d}\}})$, where the idea is to flag the presence of the entity d as a fault to understand why it has been produced. The corresponding process monitor is thus: $\mathsf{m} \triangleq \mathsf{rec}\ X.(\mathsf{F}.X + \neg\mathsf{F}.\mathbf{no}_{\{\mathsf{d}\}})$. The execution of the monitored process $\mathsf{m} \lhd \mathsf{P}_0$ proceeds by applying the rules in Table 1.

$$\frac{\mathsf{P}_0 \xrightarrow{\langle(\emptyset,\mathsf{ab})\rhd \mathsf{a,cd,bc}\rangle} \frac{\emptyset}{\mathsf{ab}}\mathsf{P}_1 \qquad \mathsf{m} \xrightarrow{\mathsf{F}} \mathsf{m} \qquad \langle(\emptyset,\mathsf{ab})\rhd \mathsf{a,cd,bc}\rangle \vDash \mathsf{F}}{\mathsf{m} \lhd \mathsf{P}_0 \xrightarrow{(\langle(\emptyset,\mathsf{ab})\rhd \mathsf{a,cd,bc}\rangle,\mathsf{F})} \mathsf{m} \lhd \frac{\emptyset}{\mathsf{ab}}\mathsf{P}_1}\ (Exec)$$

The next step is derived by using rule (*Exec*) again:

$$\frac{\frac{\emptyset}{\mathsf{ab}}\mathsf{P}_1 \xrightarrow{\langle(\mathsf{bc,ab})\rhd \mathsf{ac,d,bcd}\rangle} \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\mathsf{P}_2 \qquad \mathsf{m} \xrightarrow{\mathsf{F}} \mathsf{m} \qquad \langle(\mathsf{bc,ab})\rhd \mathsf{ac,d,bcd}\rangle \vDash \mathsf{F}}{\mathsf{m} \lhd \frac{\emptyset}{\mathsf{ab}}\mathsf{P}_1 \xrightarrow{(\langle(\mathsf{bc,ab})\rhd \mathsf{ac,d,bcd}\rangle,\mathsf{F})} \mathsf{m} \lhd \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\mathsf{P}_2}\ (Exec)$$

The computation ends after applying the (*Exec*) rule:

$$\frac{\frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\mathsf{P}_2 \xrightarrow{\langle(\mathsf{bcd,ab})\rhd \mathsf{d},\emptyset,\emptyset\rangle} \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\frac{\mathsf{bcd}}{\mathsf{ab}}\mathsf{P}_3 \qquad \mathsf{m} \xrightarrow{\neg\mathsf{F}} \mathbf{no}_{\{\mathsf{d}\}} \qquad \langle(\mathsf{b,\ c})\rhd \mathsf{bc},\emptyset,\emptyset\rangle \vDash \neg\mathsf{F}}{\mathsf{m} \lhd \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\mathsf{P}_2 \xrightarrow{(\langle(\mathsf{bcd,ab})\rhd \mathsf{d},\emptyset,\emptyset\rangle,\neg\mathsf{F})} \mathbf{no}_{\{\mathsf{d}\}} \lhd \frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\frac{\mathsf{bcd}}{\mathsf{ab}}\mathsf{P}_3}\ (Exec)$$

The computation stops and $\frac{\emptyset}{\mathsf{ab}}\frac{\mathsf{bc}}{\mathsf{ab}}\frac{\mathsf{bcd}}{\mathsf{ab}}\mathsf{P}_3$ is a starting point for backward slicing on $\{\mathsf{d}\}$.

Example 11 Given the assertion $\mathsf{F} = \{\mathtt{ifngammah}, \mathtt{tbeth}\} \nsubseteq W$, the formula $\phi \triangleq \mathsf{max}\ X.([\mathsf{F}].X \wedge [\neg\mathsf{F}]\mathbf{ff}_{\{\mathtt{tbeth}\}})$ would automatize the slicing in Example 7, by selecting `tbeth` with the monitor, if it is found present.

Let us now consider a complex biological example in which there is a continuous interaction with the environment, represented by the sequence of the provided contexts.

Example 12 This example is due to Nobile et al. (2017), where a RS was introduced to replicate one of the experiments in Helikar and et al. (2013) related to a dynamical model of ErbB receptor signal transduction in human mammary epithelial cells. The non-receptor tyrosine kinase Src and receptor tyrosine kinase epidermal growth factor receptor (EGFR/ErbB1) have been established as collaborators in cellular signaling and their combined disregulation plays key roles in human cancers, including breast cancer. The RS contains 6,720 reactions and a sequence of 1,000 contexts.

We installed a monitor looking for the introduction of an EFGR lysosome, which is essential for a bifurcation in the pathway of our experiment. We discovered that the lysosome is introduced after 11 computation steps. Then we marked the lysosome in the last state of the computation of length 11, and computed the slicing.

The resulting sliced sequence contains states with at most a couple of dozens of entities, and the context of larger size contains 17 entities. Moreover, in the sliced sequence the biologist can identify other entities that can be used for further slicing simplifications. We report here an excerpt of the sliced sequence. The first and last three states of the *sliced computation* are:

Table 2 Syntax and semantics for the sHML

Syntax	$\phi, \psi \in \text{sHML} ::= \mathbf{tt} \mid \mathbf{ff}_s \mid [\mathsf{F}].\phi \mid \phi \wedge \psi \mid \max X.\phi \mid X$		
Semantics	$[\![\mathbf{tt}, \rho]\!] \triangleq P$	$[\![\mathbf{ff}_s, \rho]\!] \triangleq \emptyset$	
	$[\![[\mathsf{F}].\phi, \rho]\!] \triangleq \{p \mid \forall q.\ p \xrightarrow{\ell} q\ , \ell \models \mathsf{F} \text{ and } q \in [\![\phi, \rho]\!]\}$	$[\![\phi \wedge \psi, \rho]\!] \triangleq [\![\phi, \rho]\!] \cap [\![\psi, \rho]\!]$	
	$[\![\max X.\phi, \rho]\!] \triangleq \bigcup\{\mathsf{P} \mid \mathsf{P} \subseteq [\![\phi, \rho[X \mapsto \mathsf{P}]]\!]\}$	$[\![X, \rho]\!] \triangleq \rho(X)$	

where ρ is a set of formula definitions, and P is the whole set of processes.

Table 3 Rules for deriving a process monitor

$\mathsf{m}(\mathbf{ff}_s) \triangleq \mathbf{no}_s$	$\mathsf{m}([\mathsf{F}].\phi) \triangleq \mathsf{F}.\mathsf{m}(\phi) + \neg\mathsf{F}.\mathbf{yes}$	$\mathsf{m}(\max X.\phi) \triangleq \mathsf{rec}\, x.\mathsf{m}(\phi)$	
$\mathsf{m}(\mathbf{tt}) \triangleq \mathbf{yes}$	$\mathsf{m}(\phi \wedge \psi) \triangleq \mathsf{m}(\phi) \otimes \mathsf{m}(\psi)$	$\mathsf{m}(X) \triangleq x$	

```
[ [alpha_ir,alpha_qr,alpha_sr,ap2,arf,cas,
    clathrin,egfr_free,grb2,hip1r,ilk,
   myosin,pak,pdk1,pi4k,pi5k,pip2_45,pten,rho,
   rhok,rin],
  ...
  [egfr_egfr_egf_mvb,egfr_egfr_egf_pm,egfr_ub,
   escrt_i,rab5],
  [alix,egfr_egfr_egf_mvb,eps15,escrt_iii,rab7],
  [egfr_egfr_egf_lysosome] ]
```

The last states of the original computation consisted of about 100 different molecules each, containing molecules which are unrelated to the introduction of the marked entities. On the other hand the sliced computation gives a sequence containing less than 33 entities in each state, which is easier to inspect, and can be used for further slicing derivations. Moreover all elements in the sequence are essential to derive the marked entities. We show the sequence of sliced contexts, which emphasizes the entities provided by the contexts which are really useful/necessary for the sliced computation:

```
[ [alpha_il,alpha_ql,alpha_sl,calm,cdc42,egf,
   egfr_contr,erk,fak,gak,pa,pip2_34,
   pip3_345,pp2a,ras,src,stress],
  [alpha_sl,cdc42,egf,fak,gak,pa,pip2_34,
  pip3_345,rho,src,stress],
  [cdc42,egfr_contr,fak,gak,pip3_345,rho,
   src,stress],
  [cdc42,egfr_contr,fak,pip3_345,ras,src],
  [cdc42,egf,egfr_contr,pip3_345,src],
  [egf,fak,pip3_345,src],
  [egf,egfr_contr,pip3_345,src],
  [egf],
  [],
  [] ]
```

Each context in the original computation adds at every step 30 entities, most of which after the first computation step become irrelevant, while only 17 were used in the first sliced step. In the last two computation steps the context does not contribute at all to the computation of the marked entity. Concerning the applied reactions, in each sliced step at most 33 are used. In the last three steps of the computation the relevant reactions are just the following ones:

```
[1558,1574,2456,3276,5573]
[91,1556,3008,3279,5599]
[1554]
```

Thus, it is possible to know exactly, within the 6,720 reactions, the ones which are relevant to compute the marked entities, as well as the relevant entities introduced by the contexts. Then it is possible to proceed to identify a bug in the model following the same strategy outlined in Example 7. In practice the biologists analyze the simplified sliced sequence by using their knowledge of the intended model, or they can express the properties to be monitored automatically.

We performed our experiments on a MacBook Pro with OSX 11.7, 2,6GHz Intel i7 6 core, with 16GB RAM. The slicing for the small Example 7 was executed in 9 ms, while the big Example 12 took 6000 ms. Considering the number of reactions (about 300 times more) and the fact that the context is absent in the first example and it is pretty large in the second one we can notice that the computation time increases linearly with the number of reactions. This is a result that we might expect also theoretically by an analysis of the algorithm, which depends directly on the number of reactions and the size of the context. We are working on an optimisation of the implementation by recording in the history also the list of the reactions applied in the computation.

7 Slicing enhanced reaction systems

Reaction Systems have been designed as a qualitative model. In fact, their theory is based on assumptions such as *no permanency*, i.e., any entity vanishes unless it is sustained by a reaction in the same way as a living cell would die for lack of energy, without chemical reactions; *no counting*, i.e., the quantity of entities that are present in a cell is not taken into

account; and *threshold nature of resources*, i.e., we assume that either an entity is available for all reactions, or it is not available at all.

More recently, different versions of RSs were also studied in several papers (Ehrenfeucht and Rozenberg 2007a; Brijder et al. 2011b; Meski et al. 2016; Ehrenfeucht et al. 2017; Bottoni et al. 2019, 2020; Koutny et al. 2021) to account for certain aspects of biological system in a more accurate way and increase their predictive power about natural phenomena. In particular, the work in Brodo et al. (2021c, 2023b) show that the process algebraic approach described in Sect. 3 allows to enhance RS models with several quantitative features in a modular way.

In this section we show that the basic slicing algorithm can be rather naturally extended to deal with quantitative extensions of RSs, like those including delays and linear processes. The extension is rather direct in the case of delays. Linear processes require some more elaborate changes due to the peculiarity of this type of RSs.

7.1 Delays, durations and timed processes

In Biology it is well known that reactions occur with different frequencies. For example, since enzymes catalyse reactions, many reactions are more frequent when some enzymes are present, and less frequent when such enzymes are absent. Moreover, reactions describing complex transformations may require time before releasing their products. To capture these dynamical aspects in our framework by preserving the discrete and abstract nature of RS, in Brodo et al. (2021c, 2023b) we have proposed a discretisation of the *delay* between two occurrences of a reaction by using a scale of natural numbers, from 0 (smallest delay, highest frequency) up to n (increasing delay, lower frequency).

Intuitively, the notation D^n stands for making the entities D available after n time units, and we use the shorthand D for D^0, meaning that the entities are immediately available. Similarly, we can associate a delay value to the product of each reaction by writing $(R, I, P)^n$ when the product of the reaction will be available after n time units, and we write (R, I, P) for $(R, I, P)^0$. The syntax for mixture processes is thus extended as below and the operational semantics is changed accordingly:

$$\mathsf{M} ::= (R, I, P)^n \mid D^n \mid \mathsf{K} \mid \mathsf{M}|\mathsf{M}$$

Figure 3 only reports the rules that are new and those that override the ones in Fig. 1 (e.g., the semantics of context processes is unchanged). Rule (*Tick*) represents the passing of one time unit, while rule (*Ent*) makes available those entities whose delay has expired. Note that the entities D^{n+1} that appear is the transition label in rule (*Tick*) cannot be used as reactants by any reaction, because their delay is not yet zero. Rule (*Pro*) delays the product of the reaction as specified by the reaction itself, while rule (*Inh*) is used when the reaction is not enabled.

Example 13 Let us consider two RSs sharing the same entity set $S = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and the same reactions $r_1 = (\mathsf{a}, \mathsf{b}, \mathsf{b})$, $r_2 = (\mathsf{b}, \mathsf{a}, \mathsf{a})$, $r_3 = (\mathsf{ac}, \mathsf{b}, \mathsf{d})$, $r_4 = (\mathsf{d}, \mathsf{a}, \mathsf{c})$, but working with different reaction speeds. For simplicity we assume that only two speed levels are distinguished: 0 the fastest and 1 the slowest. The reaction system $\mathcal{A}_1$ provides the speed assignment $\{r_1^1, r_2, r_3, r_4^1\}$ to its reactions, while $\mathcal{A}_2$ provides the speed assignment $\{r_1, r_2^1, r_3^1, r_4\}$. We assume that the context process for both reaction systems is just $\mathsf{K} \triangleq \mathsf{ac}.\emptyset.\emptyset.\mathbf{0}$. The LTSs of their corresponding timed processes are in Fig. 4, where, for brevity we let:

$$\mathsf{M}_1 \triangleq r_1^1 \mid r_2 \mid r_3 \mid r_4^1 \qquad \mathsf{M}_2 \triangleq r_1 \mid r_2^1 \mid r_3^1 \mid r_4$$
$$\mathsf{K}_{2\emptyset} \triangleq \emptyset.\emptyset.\mathbf{0}. \qquad \mathsf{K}_{\emptyset} \triangleq \emptyset.\mathbf{0}.$$

In both cases, the execution is deterministic and the corresponding traces are, respectively:

$$\frac{\emptyset}{\mathsf{ac}} \xrightarrow{\{r_1^1, r_3\}} \frac{\mathsf{b}^1\mathsf{d}}{\emptyset} \xrightarrow{\{r_4^1\}} \frac{\mathsf{bc}^1}{\emptyset} \quad \text{and} \quad \frac{\emptyset}{\mathsf{ac}} \xrightarrow{\{r_1, r_3^1\}} \frac{\mathsf{bd}^1}{\emptyset} \xrightarrow{\{r_2^1\}} \frac{\mathsf{a}^1\mathsf{d}}{\emptyset}.$$

Inspired by Brijder et al. (2011b), we can also provide entities with a duration, i.e. entities that last a finite number of steps. To this aim we use the syntax $D^{[n,m]}$ to represent the availability of D for $m > 0$ time units starting after n time units from the current time. In Brodo et al. (2021c) we have shown that durations can be encoded using just delays as follows:

$$D^{[n,m]} \triangleq \prod_{k=n}^{n+m-1} D^k \qquad (R, I, P)^{[n,m]} \triangleq \prod_{k=n}^{n+m-1} (R, I, P)^k.$$

For example, we have $\mathsf{a}^{[2,3]} \equiv \mathsf{a}^2 \mid \mathsf{a}^3 \mid \mathsf{a}^4$ and $\mathsf{a}^{[0,1]} \equiv \mathsf{a}^0 \equiv \mathsf{a}$.

We use the name *timed processes* for processes with delays and durations. Notably, our extension is conservative, i.e., it does not change the semantics of processes without delays and durations. Therefore, the encoding of standard RSs described in Definition 2 still applies.

7.1.1 Dynamic slicing of timed processes

We now explain how the base slicing algorithm for RSs can be extended to timed processes. The corresponding pseudocode is Algorithm 3. Here we have an additional case to be considered: if a delayed entity e^k is marked in D_i, then it can derive from an entity e^{k+1} in the previous state D_{i-1} with a delay which is one tick bigger than the marked one.

Fig. 3 SOS semantics with delays and durations

$$\frac{}{D^0 \xrightarrow{\langle (D,\emptyset)\rhd\emptyset,\emptyset,\emptyset\rangle} \emptyset}\ (Ent) \qquad \frac{}{D^{n+1} \xrightarrow{\langle (D^{n+1},\emptyset)\rhd\emptyset,\emptyset,\emptyset\rangle} D^n}\ (Tick)$$

$$\frac{}{(R,I,P)^n \xrightarrow{\langle (\emptyset,\emptyset)\rhd R,I,P\rangle} (R,I,P)^n \mid P^n}\ (Pro) \qquad \frac{J \subseteq I \quad Q \subseteq R \quad J \cup Q \neq \emptyset}{(R,I,P)^n \xrightarrow{\langle (\emptyset,\emptyset)\rhd J,Q,\emptyset\rangle} (R,I,P)^n}\ (Inh)$$

Fig. 4 Execution of two timed processes (see Example 13)

$$[\mathsf{M}_1|\mathsf{K}] \xrightarrow{\langle (\emptyset,\mathsf{ac})\rhd \mathsf{ac},\mathsf{bd},\mathsf{bd}\rangle} \frac{\emptyset}{\mathsf{ac}}[\mathsf{M}_1|\mathsf{b}^1|\mathsf{d}|\mathsf{K}_{2\emptyset}] \xrightarrow{\langle (\mathsf{b}^1\mathsf{d},\emptyset)\rhd \mathsf{d},\mathsf{abc},\mathsf{c}\rangle} \frac{\emptyset}{\mathsf{ac}}\frac{\mathsf{b}^1\mathsf{d}}{\emptyset}[\mathsf{M}_1|\mathsf{b}|\mathsf{c}^1|\mathsf{K}_\emptyset]$$
$$\xrightarrow{\langle (\mathsf{bc}^1,\emptyset)\rhd \mathsf{b},\mathsf{acd},\mathsf{a}\rangle} \frac{\emptyset}{\mathsf{ac}}\frac{\mathsf{b}^1\mathsf{d}}{\emptyset}\frac{\mathsf{bc}^1}{\emptyset}[\mathsf{M}_1|\mathsf{a}|\mathsf{c}|\mathbf{0}]$$

$$[\mathsf{M}_2|\mathsf{K}] \xrightarrow{\langle (\emptyset,\mathsf{ac})\rhd \mathsf{ac},\mathsf{bd},\mathsf{bd}\rangle} \frac{\emptyset}{\mathsf{ac}}[\mathsf{M}_2|\mathsf{b}|\mathsf{d}^1|\mathsf{K}_{2\emptyset}] \xrightarrow{\langle (\mathsf{bd}^1,\emptyset)\rhd \mathsf{b},\mathsf{acd},\mathsf{a}\rangle} \frac{\emptyset}{\mathsf{ac}}\frac{\mathsf{bd}^1}{\emptyset}[\mathsf{M}_2|\mathsf{a}^1|\mathsf{d}|\mathsf{K}_\emptyset]$$
$$\xrightarrow{\langle (\mathsf{a}^1\mathsf{d},\emptyset)\rhd \mathsf{d},\mathsf{abc},\mathsf{c}\rangle} \frac{\emptyset}{\mathsf{ac}}\frac{\mathsf{bd}^1}{\emptyset}\frac{\mathsf{a}^1\mathsf{d}}{\emptyset}[\mathsf{M}_2|\mathsf{a}|\mathsf{c}|\mathbf{0}]$$

Input: - a trace $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$
- a marking $D_{sliced} \subseteq D_m$

Output: a sliced trace $\frac{D'_0}{C'_0} \xrightarrow{N'_1} \cdots \xrightarrow{N'_m} \frac{D'_m}{C'_m} = \frac{D_{sliced}}{\emptyset}$

1 **begin**
2 | **let** $D'_m = D_{sliced} \wedge C'_m = \emptyset$
3 | **for** $i = m$ *to* 1 **do**
4 | | **let** $D'_{i-1} = \emptyset \wedge C'_{i-1} = \emptyset \wedge N'_i = \emptyset$
5 | | **for** $j \in N_i$ *where* $r_j = (R_j, I_j, P_j)^{n_j}$, *such that* $(D'_i \cap P_j^{n_j} \neq \emptyset)$ **do**
6 | | | **let** $N'_i = N'_i \cup \{j\}$
7 | | | **let** $D'_{i-1} = D'_{i-1} \cup (R_j \cap D_{i-1})$
8 | | | **if** $\neg en_{r_j}(D_{i-1})$ **then let** $C'_{i-1} = C'_{i-1} \cup (R_j \backslash D_{i-1})$
9 | | **end**
10 | | **for** $s^k \in D'_i$ **do**
11 | | | **if** $s^{k+1} \in D_{i-1}$ **then let** $D'_{i-1} = D'_{i-1} \cup \{s^{k+1}\}$,
12 | | **end**
13 | **end**
14 **end**

Algorithm 3: Trace Slicer for context dependent computations for timed processes

We also note that the context only introduces entities without delay.

Example 14 Let us consider Example 13, and the evolution (computation) for M_1 in Fig. 4, whose corresponding trace is:

$$\frac{\emptyset}{\mathsf{ac}} \xrightarrow{\{r_1^1, r_3\}} \frac{\mathsf{b}^1\mathsf{d}}{\emptyset} \xrightarrow{\{r_4^1\}} \frac{\mathsf{bc}^1}{\emptyset}.$$

By marking the entity b in the last state we get the simplified sliced computation as follows:

$$\frac{\emptyset}{\mathsf{a}} \xrightarrow{\{r_1^1\}} \frac{\mathsf{b}^1}{\emptyset} \xrightarrow{\emptyset} \frac{\mathsf{b}}{\emptyset}.$$

As previously discussed, the sliced trace can be also highlighted as part of the original trace using boxes:

$$\frac{\emptyset}{\boxed{\mathsf{a}}\mathsf{c}} \xrightarrow{\{\boxed{r_1^1}, r_3\}} \frac{\boxed{\mathsf{b}^1}\mathsf{d}}{\emptyset} \xrightarrow{\{r_4^1\}} \frac{\boxed{\mathsf{b}}\mathsf{c}^1}{\emptyset}$$

This shows that only reaction r_1 is involved in the first computation step, while no reaction is involved in the second computation step, as by the rules in lines 10 and 11 in Algorithm 3 entity b^1 is marked by the presence of the value b in the last computation state and the delayed value b^1 in previous state.

7.2 Concentration levels and linear processes

Quantitative modelling of chemical reactions requires taking molecule concentrations into account. An abstract representation of concentrations that is considered in many formalisms is based on *concentration levels*: rather than representing these quantities as real numbers, a finite classification is considered (e.g., *low/medium/high*) with a granularity that reflects the number of concentration levels at which significant changes in the molecule's behaviour are observed. In classical RSs, the modelling of concentration levels would require using different entities for the same molecule (e.g., $\mathsf{a}_\mathtt{l}$, $\mathsf{a}_\mathtt{m}$, and $\mathsf{a}_\mathtt{h}$ for low, medium and high concentration of a, respectively). This may introduce some additional complexity due to the need of guaranteeing that only one of these entities is present at any time for the state to be consistent. Moreover, consistency would be put at risk whenever the context sequence could provide a molecule with a concentration level different from the one present in the result sequence (e.g., how do we handle conflicting levels $\mathsf{a}_\mathtt{l} \in D_i$ and $\mathsf{a}_\mathtt{h} \in C_i$?).

In Brodo et al. (2021c) we have enhanced RS processes by representing concentration levels with natural numbers

associated with entities and using symbolic reactions whose product concentration levels can depend linearly on reactant levels. The idea is to associate linear expressions, such as $e = m \cdot x + n$ (where $m \in \mathbb{N}$ and $n \in \mathbb{N}^+$ are two constants and x stands for a variable ranging over natural numbers),[9] to reactants and products of each reaction. As a special case, when $m = 0$ the expression e is called *ground* and it is just a positive natural number. In the definition of states we exploit ground expressions: each entity in the state is paired with a natural number representing its current concentration level. In the following we write $s(e)$ to state that the linear expression e is associated to entity s and $s(e)[k/x]$ to denote the substitution of the variable x by the actual value k in e. Substitutions are extended to set of entities R as expected, by writing $R[k/x]$ to denote the substitution of the variable x by the actual value k in all the expressions present in R. Expressions associated to reactants are used as *patterns* to match the current levels of the entities involved in the reaction. Pattern matching allows to find the largest value for the variable x (the same for all reactants) that is consistent with the concentration levels in the current state. Then, linear expressions associated to products (that can contain, again, variable x) can be evaluated to compute the concentration levels of those entities. For example, the reactant $\mathsf{a}(x+1)$ requires that the concentration level of a is at least 1 and uses x to count the additional quantity of a in the current state; then the product $\mathsf{b}(2x+2)$ can be used to produce a molecule b with twice a concentration level than that of a. Expressions can be associated also to reaction inhibitors in order to let such entities inhibit the reaction only when their concentration level is above a given threshold. However, we require inhibitor expressions to be ground.

For the sake of simplicity we do not report here all the technical definitions in Brodo et al. (2021c) that formalise the computation step, but rather we try to give an intuition of how one computation step is performed. Let us consider a state W and a reaction $r = (R, I, P)$. We call *multiplicity* of r w.r.t. W the maximum value k (a natural number) that the variable x can assume so that the concentration levels of entities in $R[k/x]$ are below or equal to those in W. Of course, the enabling of the reaction also requires that the concentration levels of all inhibitors I are above those in W. The multiplicity k of the reaction determines the level $P[k/x]$ of products. As a special case, when R and P are ground[10] we let the multiplicity be 1 by default. The following example instantiates the previous description.

Example 15 Assume that we want to write a reaction that produces c with a concentration level that corresponds to the current concentration level of a (but at least one occurrence of a must be present), and that requires b not to be present at a concentration level higher than 1. Such a reaction would be $r_1 \triangleq (\mathsf{a}(x+1), \mathsf{b}(2), \mathsf{c}(x+1))$. When $C_0 \triangleq \{\mathsf{a}(3), \mathsf{b}(2)\}$ the reaction r_1 is not enabled because the concentration of the inhibitor is too high ($\mathsf{b}(2) \not< \mathsf{b}(2)$). On the contrary, if $C_0' \triangleq \{\mathsf{a}(3), \mathsf{b}(1)\}$ it holds $\mathsf{b}(1) < \mathsf{b}(2)$, and the reaction r_1 is enabled with multiplicity $k = 2$, that is the maximum value for x that satisfies $\mathsf{a}(x+1) \leq \mathsf{a}(3)$. Therefore, the concentration level of the product c is given by $\mathsf{c}(x+1)[2/x] = \mathsf{c}(2+1) = \mathsf{c}(3)$. The corresponding trace can be written:

$$\frac{\emptyset}{3\mathsf{a} + \mathsf{b}} \xrightarrow{2r_1} \frac{3\mathsf{c}}{\emptyset}$$

where the concentration levels and multiplicities are represented more concisely using standard multiset notation: we write, e.g., $3\mathsf{a} + \mathsf{b}$ for the context where the concentration level of a is 3 and that of b is 1 and $2r_1$ for the enabling of the reaction r_1 with multiplicity 2.

In the following we further exploit multiset notation, e.g.,

- By writing $W(s)$ to denote the concentration level of the entity s in the state W;
- By writing $R \subseteq W$ whenever $R(s) \leq W(s)$ for each entity $s \in S$, i.e., to denote multiset inclusion;
- By writing $s(k) \in W$ if s is present in W with a concentration level $k' \geq k$;
- By letting $W(s) = 0$ if s is not present in W;
- By writing s instead of $s(1)$.

It is worth remarking that the concentration level of an entity in the result state is the maximum concentration level of each product. Similarly, the concentration level of an entity in the current state sequence is computed as the maximum concentration level between the result state and the context. To this aim, we introduce the notation $D \sqcup C$ to combine multisets as follow:

$$(D \sqcup C)(s) \triangleq \max\{D(s), C(s)\}$$

i.e., the concentration level of s in $D \sqcup C$ is the maximum between $D(s)$ and $C(s)$.

7.2.1 Dynamic slicing of linear processes

We finally consider the extension of the base slicing algorithm to linear processes. We assume that the user may decide to mark only a subset $s(k)$ with $k < m$ of the m available s-entities (represented by notation $s(m)$) in the state, so that the slicing is concerned with how such a subset of the entities was derived. Moreover, it has to consider the *multiplicity* of

[9] To ease the presentation, we require $n \in \mathbb{N}^+$ to guarantee that e evaluates to a positive number, even when $x = 0$. Alternative choices are possible to relax this constraint.

[10] We assume that P is ground whenever R is such.

Input: - a trace $\frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$
- a marking $D_{sliced} \subseteq D_m$

Output: a sliced trace $\frac{D'_0}{C'_0} \xrightarrow{N'_1} \cdots \xrightarrow{N'_m} \frac{D'_m}{C'_m} = \frac{D_{sliced}}{\emptyset}$

```
1  begin
2      let D'_m = D_sliced
3      for i = m to 1 do
4          let D'_{i-1} = ∅ ∧ C'_{i-1} = ∅ ∧ N'_{i-1} = ∅
5          for j ∈ N_i where r_j = (R_j, I_j, P_j), k_j = N_i(j), and
           (∃s ∈ D'_i.D'_i(s) ≤ P_j[k_j/x](s)) do
6              let N'_i = N'_i ⊔ {k_j r_j}
7              for s ∈ R_j do
8                  let m = R_j[k_j/x](s)
9                  if s(m) ∈ D_{i-1} let D'_{i-1} = D'_{i-1} ⊔ {s(m)}
10                 else let C'_{i-1} = C'_{i-1} ⊔ {s(m)}
11             end
12         end
13     end
14 end
```

Algorithm 4: Trace Slicer for context dependent computations with linear processes

each reaction in a computation step, and then also how many of entities come from the context or from the result state.

The algorithm we propose is Algorithm 4, where we assume that $D_i, C_i, N_i, \ldots$ are multisets. Line 5 selects from N_i all reactions $r_j = (R_j, I_j, P_j)$ with multiplicity k_j such that their product $P_j[k_j/x]$ includes at least one entity $s \in D'_i$ with a multiplicity $P_j[k_j/x](s)$ greater or equal to the one s has in D'_i: in such cases, r_j can be responsible for the production of the marked entities and its reactants should be considered at the stage $i-1$. Line 6 updates N'_i to include r_j with its multiplicity k_j. Then for any reactant s of r_j we let m be the multiplicity of s required to enable k_j instances of r_j and update one of D'_{i-1} or C'_{i-1}.

Example 16 Let us consider Example 15. By marking $\mathsf{c}(3)$ in the final state we obtain the following sliced computation step:

$$\frac{\emptyset}{\boxed{\mathsf{a}(3)} + \mathsf{b}(1)} \xrightarrow{\boxed{2r_1}} \frac{\boxed{\mathsf{c}(3)}}{\emptyset}$$

We note that by line 10 in Algorithm 4, we derive a marking of the three a's entities in the first computation state, which is the quantity required by reaction r_1.

Let us consider one last example which illustrates the differences in the sliced computations when different quantities of entities are marked.

Example 17 Let us consider the following linear RS over $S = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ whose reactions are $r_1 = (\{\mathsf{a}(x+1), \mathsf{c}(x+1)\}, \{\mathsf{b}(3)\}, \{\mathsf{c}(x+1)\})$, $r_2 = (\{\mathsf{a}(x+4)\}, \{\mathsf{b}(2)\}, \{\mathsf{c}(x+1), \mathsf{a}(x+1)\})$ $r_3 = (\{\mathsf{c}(x+1)\}, \{\mathsf{b}(1)\}, \{\mathsf{c}(x+2)\})$, and context $\mathsf{K} = \{\mathsf{a}(5), \mathsf{c}(3), \mathsf{b}(1)\}.\{\mathsf{a}(2)\}.\emptyset.\mathbf{0}$. Then we get the following computation trace:

$$\frac{\emptyset}{5\mathsf{a}+3\mathsf{c}+\mathsf{b}} \xrightarrow{2r_1+r_2} \frac{2\mathsf{a}+3\mathsf{c}}{2\mathsf{a}} \xrightarrow{r_1+2r_3} \frac{4\mathsf{c}}{\emptyset}$$

If the user marks $\mathsf{c}(4)$ in the last state, we obtain the following sliced computation:

$$\frac{\emptyset}{2\mathsf{a}+\boxed{3\mathsf{a}+3\mathsf{c}}+\mathsf{b}} \xrightarrow{\boxed{2r_1}+r_2} \frac{2\mathsf{a}+\boxed{3\mathsf{c}}}{2\mathsf{a}} \xrightarrow{r_1+\boxed{2r_3}} \frac{\boxed{4\mathsf{c}}}{\emptyset}$$

which is, by erasing all non boxed elements:

$$\frac{\emptyset}{3\mathsf{a}+3\mathsf{c}} \xrightarrow{2r_1} \frac{3\mathsf{c}}{\emptyset} \xrightarrow{2r_3} \frac{4\mathsf{c}}{\emptyset}$$

If instead the user marks $\mathsf{c}(2)$ in the last state, we obtain the following sliced computation:

$$\frac{\emptyset}{2\mathsf{a}+\boxed{3\mathsf{a}+3\mathsf{c}+\mathsf{b}}} \xrightarrow{\boxed{2r_1+r_2}} \frac{\boxed{2\mathsf{a}+3\mathsf{c}}}{2\mathsf{a}} \xrightarrow{r_1+2r_3} \frac{\boxed{2\mathsf{c}}+2\mathsf{c}}{\emptyset}$$

We note that by selecting only two c-entities in the last state there are two reactions (r_1 and r_3) which are able to produce at least two c-entities in the last computation step and hence we obtain a sliced computation which is a kind of 'superset' of the previous one.

Let us note that Proposition 2 can be generalised to the cases of the Algorithms 3 and 4 for the quantitative extensions and can be proven following a similar pattern.

8 Related work

Dynamic program slicing has been applied to several programming paradigms, for instance to imperative programming (Korel and Laski 1988), functional programming (Ochoa et al. 2008), Term Rewriting (Alpuente et al. 2011), functional logic programming (Alpuente et al. 2014, 2016), and Constraint Concurrent Programming (Falaschi et al. 2020). RSs use term rewriting and sets manipulation for its basic computation mechanism. Thus, Alpuente et al. (2011, 2016) have a similarity with our work in the adoption of a backward style of computation of the slicer and Alpuente et al. (2016) uses assertions to stop the computation and to start the slicing process. However our framework and the one in Alpuente et al. (2011, 2016) are quite different: the former is oriented towards functional computations, while the latter considers the Maude language. In the language that they consider there are neither inhibitors, which introduce a kind of non-monotonic behaviour in the rewriting rule, nor

a notion of interactive context. We tried to apply the framework in Alpuente et al. (2011, 2016) on some examples, but we were losing too much information, and the resulting simplified computations were not informative. We have instead defined a framework which is totally specialised and suitable for RSs, we use different logics specific for RSs and we are the first to consider monitors for checking the verification process in a slicer. The automatization of the slicing has been inspired by the recent work by Aceto et al. (2020, 2021a, 2021b), here extended to exploit assertions over labels and the explicit marking of entities that are primary responsible for the fault detection. Azimi et al. (2016) defines a framework for model checking RSs. There is some similarity with our system, in the sense that both frameworks use a logic for checking properties of the underlying model. However our system considers set oriented properties specific for RSs, and analyzes one computation per execution, while the specifications considered in model checking can be more general. Slicing does not consider the full set of possible states in all possible execution traces as model checking do. The strategy is different. Model checking aims to prove if a property holds considering all possible computation states, and it provides a counterexample if it does not. Slicing returns a simplified trace for one specific computation which will help the users in their analyses. In Brodo et al. (2019, 2021b) we derived a similar LTS to the one in this paper by encoding RSs into cCNA, a multi-party process algebra (a variant of the `link`-calculus defined in Bodei et al. 2019; Brodo and Olarte 2017). In comparison with the encoding of RS in cCNA, here the SOS semantics is extended to traces and tailored for RSs, without relying on an ad hoc translation.

Our implementation introduces several novel features not covered in the literature and has been designed as a tool for verification, for slicing, as well as for rapid prototyping extensions of RSs, not just for their simulations. For ordinary interactive RSs there are already some performant simulators, such as `brsim`, written in Haskell (Azimi et al. 2015) or such as `HERESY` which is a GPU-based simulator of RSs, written using CUDA (Nobile et al. 2017). Using Prolog has had the advantage of a more flexible, safe and rapid prototyping.

Ehrenfeucht and Rozenberg (2007a) define the notion of *event* for *"extended reaction systems"* (which are not standard RSs), which roughly consists of considering a computation $W_0, \ldots, W_n$ in RS taking a subsequence of states generated by the RS, starting from a set $U \subseteq W_i$, with $0 \leq i < n$. The sets which are in the event sequence are called *modules*. Ehrenfeucht and Rozenberg (2007a) consider that two different events can merge, when one event generates one module of state W_k which is a subset of the corresponding module in the other event. The notion of event can remind our notion of sliced sequence. However there are many differences. An event corresponds to a forward computation, while we compute a slice backwards. We focus on a set of marked entities, a notion which is not considered in Ehrenfeucht and Rozenberg (2007a). Thus, we look for "a minimal module" which includes the entities of our interest and try to determine the sliced subsequence which is responsible for their introduction. On the other hand, the notion of event and of module do not make a clear separation of the role of the context for deriving the marked entities, which is essential for our framework.

The quantitative enhancements or RSs based on delays and linear processes were first studied in Brodo et al. (2021c, 2023b). Some previous work Ehrenfeucht and Rozenberg (2009) introduced a notion of time in RSs with the purpose of computing the time distance between two different states in a single state sequence. Thus, the authors define, when it exists, a positive and strictly increasing time function, called *universal* time function, for a RS that assigns a value to each state and it determines how many time units elapse between two consecutive states in an interactive process for that RS. In our approach, the passing time between two consecutive states is alway a time unit. We define a duration for each reactant as a value indicating how many time units it will be active, and a delay value n is associated to each reaction whose effect is to make available the produced reagents only after n time units. We believe that the concept of delay as we defined in Brodo et al. (2021c, 2023b) is novel in the literature on RSs.

The work in Brijder et al. (2011b) introduces a time duration for each entity that indicates how many steps the entity lasts, instead of decaying in the next step time; the authors point out that this mechanism can also be formalised as an interacting context. The entity duration function is then applied to define entity concentrations and to formalise some theoretical results. Later on, Salomaa (2017) introduced a time duration definition for each entity similar to the one in Brijder et al. (2011b). The duration function is then exploited to state a theoretical result. As in Brijder et al. (2011b), Salomaa (2017), we specify a time duration for each entity by assigning a value indicating how many time steps it will be active. As a difference to Brijder et al. (2011b), Salomaa (2017) we define a delay value n associated with each reaction whose effect is to make available the produced reagents only after n time steps. Thus, we make explicit the time duration of each entity and the time delay each reaction takes to be completed. We also remark that in Brijder et al. (2011b), Salomaa (2017) the duration is fixed for each entity, while in our work we can define a different duration for each reaction, leading to different durations depending on the enabled reactions. We believe that our framework for slicing is a good basis to be extended also to the works in Brijder et al. (2011b), Salomaa (2017), even if they mainly seem devoted to the development of theoretical results. In Meski et al. (2016) the authors introduce an extension of RSs by considering

discrete concentrations and introducing some quantitative modelling. There, 'bags' are used to count the number of each entity which are necessary to enable a reaction. However as a difference to our system they do not consider a concept of *multiplicity* like us. Thus, whenever they apply a reaction, the number of produced entities is the one indicated explicitly by the bags of the products of the reaction. Similarly to us Meski et al. (2016) takes the maximum number of entities produced by the reactions, and the maximum considering also the contribution of the context. We believe that our slicing algorithm for linear processes might be easily instantiated to the case of the system in Meski et al. (2016).

9 Conclusions and future work

We have presented the first framework for dynamic slicing of RSs. We have defined a slicing algorithm which has been applied first to standard RSs: both closed systems or interactive (context depending) ones can be dealt with. Then we have shown that the slicer can be applied also to several quantitative extensions of RSs by defining some natural modifications of the basic algorithm. This way we have been able to define a slicer for analysing RSs which consider notions of reaction speed and delays in the activation of the entities, as well as for treating discrete concentration levels in reactions (Brodo et al. 2021c, 2023b).

The main advantage of the slicer is the possibility to automatically extract a simplified computation, with the minimal amount of information that is relevant to explain the production of marked entities. Monitors help to identify states which violate a property and to automate marking of such states, using a flexible language to specify assertions over computations. We have implemented the slicer (Algorithm 1) for RSs, including the monitors, in our working environment called BioResolve[11], in Prolog, and have applied our tool to some bioinformatic examples. We are currently working to expand our implementation to the quantitative extensions (Algorithms 3 and 4).

As future work, we plan to add forward slicing to our interpreter, and explore the advantages of combining the information that we can derive by proceeding in the two directions (backward and forward), making the analysis more precise. We also want to extend our slicing algorithm to keep trace of inhibitors, which are essential for the computation. This would provide the user with a fully detailed information about the entities which are involved in the computation of the marked elements.

Acknowledgements We thank the anonymous reviewers for their comments and suggestions that helped us to improve our paper.

[11] Available at http://www.di.unipi.it/~bruni/LTSRS/

Author Contributions The authors of this paper have contributed equally.

Funding Open access funding provided by Universitá degli Studi di Siena within the CRUI-CARE Agreement. This research has been partially supported by the Italian MUR PRIN 2022 project "MEDICA" (2022RNTYWZ), by the Next Generation EU programme project PNRR ECS00000017 - "THE - Tuscany Health Ecosystem" - Spoke 3 - CUP B62C22000680007, by the Italian MUR PRIN 2022 PNRR project *Resource Awareness in Programming: Algebra, Rewriting, and Analysis* (P2022HXNSC), by the University of Pisa PRA_2022_99 *Formal methods for the healthcare domain based on spatial information* and by the 2023-24 INdAM–GNCS project CUP_E53C22001930001.

Data availibility No Data associated in the manuscript.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Ethical approval This is not applicable.

References

Aceto L, Achilleos A, Francalanza A et al (2020) Determinizing monitors for HML with recursion. JLAMP 111:100515. https://doi.org/10.1016/j.jlamp.2019.100515

Aceto L, Achilleos A, Francalanza A et al (2021a) The best a monitor can do. In: Proc. CSL, pp 7:1–7:23. https://doi.org/10.4230/LIPIcs.CSL.2021.7

Aceto L, Achilleos A, Francalanza A et al (2021) An operational guide to monitorability with applications to regular properties. Softw Syst Model 20:335–361. https://doi.org/10.1007/s10270-020-00860-z

Alpuente M, Ballis D, Espert J et al (2011) Backward trace slicing for rewriting logic theories. In: Proc. of CADE'11. Springer-Verlag, Berlin, pp 34–48. https://doi.org/10.1007/978-3-642-22438-6_5

Alpuente M, Ballis D, Frechina F et al (2014) Using conditional trace slicing for improving Maude programs. Sci Comput Program 80:385–415. https://doi.org/10.1016/j.scico.2013.09.018

Alpuente M, Ballis D, Frechina F et al (2016) Debugging Maude programs via runtime assertion checking and trace slicing. J Log Algebr Methods Program 85:707–736. https://doi.org/10.1016/j.jlamp.2016.03.001

Azimi S (2017) Steady states of constrained reaction systems. Theor Comput Sci 701:20–26. https://doi.org/10.1016/j.tcs.2017.03.047

Azimi S, Iancu B, Petre I (2014) Reaction system models for the heat shock response. Fund Inform 131(3–4):299–312. https://doi.org/10.3233/FI-2014-1016

Azimi S, Gratie C, Ivanov S et al (2015) Dependency graphs and mass conservation in reaction systems. Theor Comput Sci 598:23–39. https://doi.org/10.1016/j.tcs.2015.02.014

Azimi S, Gratie C, Ivanov S et al (2016) Complexity of model checking for reaction systems. Theor Comput Sci 623:103–113. https://doi.org/10.1016/J.TCS.2015.11.040

Barbuti R, Gori R, Levi F et al (2016) Investigating dynamic causalities in reaction systems. Theor Comput Sci 623:114–145. https://doi.org/10.1016/j.tcs.2015.11.041

Barbuti R, Gori R, Milazzo P (2021) Encoding Boolean networks into reaction systems for investigating causal dependencies in gene regulation. Theor Comput Sci 881:3–24. https://doi.org/10.1016/j.tcs.2020.07.031

Bodei C, Brodo L, Bruni R (2019) A formal approach to open multiparty interactions. Theor Comput Sci 763:38–65. https://doi.org/10.1016/J.TCS.2019.01.033

Bottoni P, Labella A, Rozenberg G (2019) Reaction systems with influence on environment. J Membr Comput 1(1):3–19. https://doi.org/10.1007/s41965-018-00005-8

Bottoni P, Labella A, Rozenberg G (2020) Networks of reaction systems. Int J Found Comput Sci 31:53–71. https://doi.org/10.1142/S0129054120400043

Brijder R, Ehrenfeucht A, Main M et al (2011) A tour of reaction systems. Int J Found Comput Sci 22(07):1499–1517. https://doi.org/10.1142/S0129054111008842

Brijder R, Ehrenfeucht A, Rozenberg G (2011b) Reaction systems with duration. In: Kelemen J, Kelemenová A (eds) Computation, cooperation, and life: essays dedicated to gheorghe Păun on the occasion of his 60th birthday. Springer, Berlin, pp 191–202. https://doi.org/10.1007/978-3-642-20000-7_16

Brodo L, Olarte C (2017) Symbolic semantics for multiparty interactions in the link-calculus. In: Proc. of SOFSEM'17. Lecture Notes in Computer Science, vol 10139. Springer, Berlin, pp 62–75, https://doi.org/10.1007/978-3-319-51963-0_6

Brodo L, Bruni R, Falaschi M (2019) Enhancing reaction systems: a process algebraic approach. In: The art of modelling computational systems, LNCS, vol 11760. Springer, Berlin, pp 68–85. https://doi.org/10.1007/978-3-030-31175-9_5

Brodo L, Bruni R, Falaschi M (2021) A logical and graphical framework for reaction systems. Theor Comput Sci 875:1–27. https://doi.org/10.1016/j.tcs.2021.03.024

Brodo L, Bruni R, Falaschi M (2021) A process algebraic approach to reaction systems. Theor Comput Sci 881:62–82. https://doi.org/10.1016/j.tcs.2020.09.001

Brodo L, Bruni R, Falaschi M, et al (2021c) Exploiting modularity of SOS semantics to define quantitative extensions of reaction systems. In: Aranha C, Martín-Vide C, Vega-Rodríguez MA (eds) Proc. of TPNC 2021, Lecture Notes in Computer Science, vol 13082. Springer, Berlin, pp 15–32. https://doi.org/10.1007/978-3-030-90425-8_2

Brodo L, Bruni R, Falaschi M (2023a) Dynamic slicing of reaction systems based on assertions and monitors. In: Hanus M, Inclezan D (eds) Proc. of practical aspects of declarative languages–25th Int. Symp., PADL 2023, LNCS, vol 13880. Springer, Berlin, pp 107–124. https://doi.org/10.1007/978-3-031-24841-2_8

Brodo L, Bruni R, Falaschi M et al (2023) Quantitative extensions of reaction systems based on SOS semantics. Neural Comput Appl 35(9):6335–6359. https://doi.org/10.1007/s00521-022-07935-6

Corolli L, Maj C, Marini F et al (2012) An excursion in reaction systems: from computer science to biology. Theor Comput Sci 454:95–108. https://doi.org/10.1016/j.tcs.2012.04.003

Ehrenfeucht A, Rozenberg G (2007) Events and modules in reaction systems. Theor Comput Sci 376(1–2):3–16. https://doi.org/10.1016/j.tcs.2007.01.008

Ehrenfeucht A, Rozenberg G (2007) Reaction systems. Fundam. Inform 75(1–4):263–280

Ehrenfeucht A, Rozenberg G (2009) Introducing time in reaction systems. Theor Comput Sci 410(4):310–322. https://doi.org/10.1016/j.tcs.2008.09.043

Ehrenfeucht A, Kleijn J, Koutny M et al (2017) Evolving reaction systems. Theor Comput Sci 682:79–99. https://doi.org/10.1016/j.tcs.2016.12.031

Falaschi M, Gabbrielli M, Olarte C, et al (2016) Slicing concurrent constraint programs. In: Hermenegildo M, López-García P (eds) Proc. of LOPSTR 2016, LNCS, vol 10184. Springer, Berlin, pp 76–93. https://doi.org/10.1007/978-3-319-63139-4_5

Falaschi M, Gabbrielli M, Olarte C et al (2020) Dynamic slicing for concurrent constraint languages. Fundam Inform 177(3–4):331–357. https://doi.org/10.3233/FI-2020-1992

Ferretti C, Leporati A, Manzoni L et al (2020) The many roads to the simulation of reaction systems. Fundam Inform 171(1–4):175–188. https://doi.org/10.3233/FI-2020-1878

Helikar T et al (2013) A comprehensive, multi-scale dynamical model of ERBB receptor signal transduction in human mammary epithelial cells. PLoS ONE 8(4):1–9. https://doi.org/10.1371/journal.pone.0061757

Ivanov S, Rogojin V, Azimi S, et al (2018) WEBRSIM: A web-based reaction systems simulator. In: Díaz CG, Riscos-Núnez A, Paun G et al (eds) Enjoying natural computing–essays dedicated to Mario de Jesús Pérez-Jiménez on the occasion of his 70th birthday, Lecture Notes in Computer Science, vol 11270. Springer, Berlin, pp 170–181. https://doi.org/10.1007/978-3-030-00265-7_14

Korel B, Laski J (1988) Dynamic program slicing. Inf Process Lett 29(3):155–163. https://doi.org/10.1016/0020-0190(88)90054-3

Koutny M, Pietkiewicz-Koutny M, Yakovlev A (2021) Asynchrony and persistence in reaction systems. Theor Comput Sci 881:97–110. https://doi.org/10.1016/j.tcs.2020.11.040

Meski A, Koutny M, Penczek W (2016) Towards quantitative verification of reaction systems. In: Amos M, Condon A (eds) Unconventional computation and natural computation. Springer International Publishing, Cham, pp 142–154. https://doi.org/10.1007/978-3-319-41312-9_12

Milner R (1980) A calculus of communicating systems, LNCS, vol 92. Springer, Berlin. https://doi.org/10.1007/3-540-10235-3

Nobile MS, Porreca AE, Spolaor S et al (2017) Efficient simulation of reaction systems on graphics processing units. Fundam Inform 154(1–4):307–321. https://doi.org/10.3233/FI-2017-1568

Ochoa C, Silva J, Vidal G (2008) Dynamic slicing of lazy functional programs based on Redex trails. Higher Order Symbol Comput 21(1–2):147–192. https://doi.org/10.1007/s10990-008-9023-7

Okubo F, Yokomori T (2016) The computational capability of chemical reaction automata. Nat Comput 15(2):215–224. https://doi.org/10.1007/s11047-015-9504-7

Plotkin GD (1981) A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University

Plotkin GD (2004) A structural approach to operational semantics. J Log Algebr Methods Program 60–61:17–139. https://doi.org/10.1016/j.jlap.2004.05.001

Salomaa A (2017) Minimal reaction systems: duration and blips. Theor Comput Sci 682:208–216. https://doi.org/10.1016/j.tcs.2017.01.032

Silva J (2012) A vocabulary of program slicing-based techniques. ACM Comput Surv 44(3):12:1-12:41. https://doi.org/10.1145/2187671.2187674

Weiser M (1984) Program slicing. IEEE Trans Soft Eng 10(4):352–357. https://doi.org/10.1109/TSE.1984.5010248