# Chapter 1

# Introduction

Reaction Systems (RSs) are a successful computational framework inspired by biological system. The underlying idea is that the interactions between biochemical reactions as well as the functioning of single reactions are based on the mechanisms of facilitation and inhibition.

A Reaction System consists of a set of entities and a set of reactions over them. Each reaction produces some set of entities P (called products) if enabled, meaning if a set R (called reactants) is wholly present and if a set I (called inhibitors) of entities is completely absent. The use of inhibitors induces non-monotonic behaviors that are difficult to analyze. This work aims to build software that aids in the study and analysis of certain classes of Reaction Systems.

Entities can also be provided by an external context sequence to simulate *in silico* biological experiment.

# Chapter 2

# Background

## 2.1 Reaction Systems

Reaction Systems are a qualitative model inspired by biochemical processes. The behavior is described by reactions, each of them requiring some reactants $R$ and requiring the absence of inhibitors $I$ to produce some product elements $P$. "Elements" and "entities" will be used interchangeably to refer to elements of these sets.

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

The reactions $(R, I, P)$ operate over a finite set of entities $S$, called *background set.* The theory of RSs is based on three assumptions:

- no permanency, meaning entities vanish unless sustained by a reaction;
- no counting, meaning the exact quantity of each entity is irrelevant;
- threshold nature of resources, meaning if an entity is present, it is present for all possible reactions.

**Definition.** Let $T$ be a finite set.

1. Let $a$ be a reaction. Then $a$ is enabled by $T$, denoted by $en_a(T)$ if $R_a \subseteq T$ and $I_a \cap T = \emptyset$. The result of $a$ on $T$, denoted by $res_a(T)$, is defined by: $res_a(T) := P_a$ if $en_a(T)$ and $res_a(T) := \emptyset$ otherwise.

2. Let $A$ be a finite set of reactions. The result of $A$ on $T$, denoted by $res_A(T)$, is defined as $res_A(T) := \bigcup_{a \in A} res_a(T)$.

Note that by virtue of the second assumption (2) if two reactions $a, b \in A$, with both $a$ and $b$ enabled by T, then even if $R_a \cap R_b \neq \emptyset$, still both $P_a \subseteq res_A(T)$ and $P_b \subseteq res_A(T)$. Both reactions can use $R_a \cap R_b$ to produce their products. This would not be allowed in other models such as Petri nets [6], a common model of concurrent systems.

Let $rac(S)$ be the set of all the reactions in $S$.

**Definition** (Reaction System)**.** A Reaction System (RS) is a pair $\mathcal{A} = (S, A)$ such that $S$ is a finite set and $A \subseteq rac(S)$ is a finite set of reactions in S.

The set $S$ is called the *background set* of $\mathcal{A}$; its elements, called *entities*, represent molecular entities (e.g. atoms, ions, molecules) that may be present in the state of the system modeled by $\mathcal{A}$. The set $A$ is called the *set of reactions* of $\mathcal{A}$. Since $S$ is finite, so is $A$.

the behavior of a RS is formalized through the notion of an interactive process.

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

Note that $C_i$ and $D_i$ do not have to be disjointed.

$W_0 = C_0$is called the initial state of $\pi$. If $C_i \subseteq D_i$ for all $i \in [1, n]$ then $\pi$ is called context-independent. For context-independent interactive process, we can take $C_i = \emptyset$ for all $i = [1, n]$ without changing the state sequence.

In a context-independent state sequence $\tau = W_0, \ldots, W_i, W_{i+1}, \ldots, W_n$, during the transition from $W_i$ to $W_{i+1}$ all entities from $W_i - res_{\mathcal{A}}(W_i)$ will not persist. This reflects the assumption of no permanency (1). Thus, if $\tau$ is not context-independent, an entity from a current state can also be sustained by the context $C_{i+1}$.

**Definition** (Sequence Shift)**.** Let $\gamma = \{C_i\}_{i\in[0,n]}$ a context sequence. Given a positive integer $k \leq n$, let $\gamma^k := \{C_{i+k}\}_{i\in[0,n-k]}$.

### 2.1.1 Example: A binary counter

A reaction system can act as a cyclic n-bit counter in which external signals trigger increment or decrement operations. To build the counter, let $n > 0$ be an integer and define the background set as $\{p_0, p_1, \ldots, p_{n-1}\} \cup \{dec, inc\}$.

Five sets of reactions describe a binary counter-like behavior:

$$
\begin{aligned}
a_j &= (\{p_j\}, dec, inc, p_j), && \forall j \in [0, n] \\
b_j &= (\{inc, p_0, p_1, \ldots, p_{j-1}\}, dec, p_j, p_j), && \forall j \in [0, n] \\
c_{j,k} &= (\{inc, p_k\}, dec, p_j, p_k), && \forall j, k \text{s.t.} 0 \leq j < k < n \\
d_j &= (\{dec\}, inc, p_0, p_1, \ldots, p_j, p_j), && \forall j \in [0, n] \\
e_{j,k} &= (\{dec, p_j, p_k\}, inc, p_k), && \forall j, k \text{s.t.} 0 \leq j < k < n
\end{aligned}
$$

where reactions $a$ cause the bits to be restrained in the next state if there is no operation, reactions $b$ implement the increment operation by flipping the least significant zero to one, reactions $c$ let the more significant bits remain, reactions $d$ implements the decrement operation by flipping to one the bits when there is no one at a lower position, and reactions $e$ let the more significant bits remain.

The complete RS $\mathcal{B}_n$ is defined as follows: $\mathcal{B}_n = (S_n, B_n)$ where $S_n = \{p_0, p_1, \ldots, p_{n-1}\} \cup \{dec, inc\}$ and $B_n = \{a_j, b_j, d_j | 0 \leq k < n\} \cup \{c_{j,k}, e_{j,k} | 0 \leq l < k < n\}$.

To illustrate the system in action consider the sequence of contexts: $C_0 = \{p1, p3\}, C_1 = \emptyset, C_2 = \{inc\}, C_3 = \{inc\}, C_4 = \{dec\}, C_5 = \{dec, inc\}$. This gives the result sequence $\delta = \emptyset.\{p1, p3\}.\{p1, p3\}.\{p0, p1, p3\}.\{p2, p3\}.\{p0, p1, p3\}.\emptyset$ and state sequence $\tau = \{p1, p3\}.\{p1, p3\}.\{p1, p3, inc\}.\{p0, p1, p3, inc\}.\{p2, p3, dec\}.\{p0, p1, p3, dec, inc\}.\emptyset$ that in binary representation is $\{1010\}.\{1010\}.\{1010\}.\{1011\}.\{1100\}.\{1011\}.\{0000\}$ by ignoring *inc* and *dec*.

## 2.2 SOS rules for reaction systems

The behavior of a RS could be defined as a discrete time interactive process: a finite context sequence describes the entities provided by the environment at each step, the current state is determined by the union of the entities coming from the environment with those produced from the previous step and the state sequence is determined by applying all and only the enabled reactions to the set of entities available in the current state.

Given the context sequence, the semantics of RSs is uniquely determined and can be represented as a finite, deterministic and labeled or unlabeled transition system. RS have had defined a Labeled Transition System (LTS) semantics as seen in[3].

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

| | | | |
|---|---|---|---|
| $P$ | ::= | $[M]$ | mixture process |
| $M$ | ::= | $(R, I, P)$ | reaction |
| | \| | $D$ | set of entities |
| | \| | $K$ | context process |
| | \| | $M \vert M$ | parallel composition |
| $K$ | ::= | $0$ | nil context |
| | \| | $X$ | process variable |
| | \| | $C.X$ | set of entities followed by context |
| | \| | $K + K$ | non deterministic choice |
| | \| | $\texttt{rec}X.K$ | recursive operator |

Where

| | | |
|---|---|---|
| $R, I, P$ | $\subseteq$ | $S$ non empty sets of entities |
| $C, D$ | $\subseteq$ | $S$ possibly empty set of entities |
| $X$ | | a process variable |

An RS process $P$ embeds a mixture process $M$ obtained as he parallel composition of some reactions $(R, I, P)$, some set of currently present entities $D$ (possibly the empty set $\emptyset$), and some context process $K$. For brevity sake $M_1|M_2|M_3 = \prod_{i\in\{1,2,3\}} M_i$. A process context $K$ is a possibly nondeterministic and recursive system: the nil context **0** stops the computation; the prefixed context $C.K$ provides the entities in $C$ and then uses $K$ as the next context; the non deterministic choice $K_1 + K_2$ allows the context to behave either as $K_1$ or $K_2$; X is a process variable; and $\texttt{rec}X.K$ is the recursive operator of process algebras. For brevity sake $K_1 + K_2 + K_3 = \sum_{i\in\{1,2,3\}} K_i$.

**Definition** (RSs as RS processes)**.** Let $\mathcal{A} = (S, A)$ be a RS, and $\pi = (\gamma, \delta)$ an $n$-step interactive process in $\mathcal{A}$ 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 $[\![\mathcal{A}, \pi]\!]_i$ is defined as follows:

$$[\![\mathcal{A}, \pi]\!]_i := \left[\prod_{a\in A} a|D_i|K_{\gamma^i}\right]$$

where the context process $K_{\gamma^i} := C_i.C_{i+1}.\ldots.C_n.\mathbf{0}$ is the sequentialization of the entities offered by $\gamma^i$.

**Definition** (Label)**.** A label is a tuple $\langle W \triangleright R, I, P\rangle$ with $W, R, I, P \subseteq S$.

In a transition label $\langle W \triangleright R, I, P\rangle$, $W$ records the set of entities currently in the system (produced in the previous step or provided by the context), $R$ records the set of entities whose presence is assumed (either because they are needed as reactants on an applied reaction or because their presence prevents the application of some reaction), $I$ records the set of entities whose absence is assumed, and $P$ records the set of entities produced by the applied reactions.

**Definition** (Operational Semantics)**.** The operational semantics of processes is defined by the set of SOS inference rules in figure 2.1.

$$\frac{}{D \xrightarrow{\langle D \,\triangleright\, \emptyset,\emptyset,\emptyset\rangle} \emptyset}\ (\text{Ent}) \qquad \frac{}{C.K \xrightarrow{\langle C \,\triangleright\, \emptyset,\emptyset,\emptyset\rangle} K}\ (\text{Ctx}) \qquad \frac{K\left[\mathtt{rec}X.K/X\right] \xrightarrow{\langle W\triangleright R,I,P\rangle} K'}{\mathtt{rec}X.K \xrightarrow{\langle W \,\triangleright\, R,I,P\rangle} K'}\ (\text{Rec})$$

$$\frac{K_1 \xrightarrow{\langle W\triangleright R,I,P\rangle} K_1'}{K_1 + K_2 \xrightarrow{\langle W \,\triangleright\, R,I,P\rangle} K_1'}\ (\text{Suml}) \qquad \frac{K_2 \xrightarrow{\langle W\triangleright R,I,P\rangle} K_2'}{K_1 + K_2 \xrightarrow{\langle W \,\triangleright\, R,I,P\rangle} K_2'}\ (\text{Sumr})$$

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

$$\frac{M_1 \xrightarrow{\langle W_1\triangleright R_1,I_1,P_1\rangle} M_1' \quad M_2 \xrightarrow{\langle W_2\triangleright R_2,I_2,P_2\rangle} M_2' \quad (W_1 \cup W_2 \cup R_1 \cup R_2) \cap (I_1 \cup I_2)}{M_1|M_2 \xrightarrow{\langle W_1\cup W_2 \,\triangleright\, R_1\cup R_2,I_1\cup I_2,P_1\cup P_2\rangle} M_1'|M_2'}\ (\text{Par})$$

$$\frac{M \xrightarrow{\langle W\triangleright R,I,P\rangle} M' \qquad R \subseteq W}{[M] \xrightarrow{\langle W \,\triangleright\, R,I,P\rangle} [M']}\ (\text{Sys})$$

Figure 2.1: SOS semantics of the reaction system process

$K\,[^{\texttt{rec}X.K}/_X]$ denotes the process obtained by replacing in $K$ every free occurrence of the variable $X$ with its recursive definition $\texttt{rec}X.K$. The rule (Pro), executes the reaction $(R, I, P)$ (its reactants, inhibitors, and products are recorded the label), which remains available at the next step together with $P$. The rule (Inh) applies when the reaction $(R, I, P)$ should not be executed; it records in the label the possible 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 for explaining why the reaction is not enabled.

## 2.3 Positive Reaction Systems

A particular kind of Reaction Systems are those without inhibitors. Such reactions are called positive and can be simply written as pairs $(R, P)$ and are equivalent to $(R, \emptyset, P)$. One can always encode any standard RS $\mathcal{A} = (S, A)$ into an equivalent one without inhibitors. In order to track the absence of entities, a new "negative" entity is added for each original one. In any meaningful state $W = D \cup C$ there will always be either one between $a$ and $\bar{a}$, but never both. As a consequence, for any entity $a \in S_C$, we must assume that the context will provide either $a$ or $\bar{a}$. Define $\mathbf{S} := S \uplus \bar{S}$ and $\bar{S} := \{\bar{a} | a \in S\}$. A subscript $D$ or $C$ will be used to differentiate between entities related to reaction products and related to the context.

**Definition** (State consistency)**.** A set $\mathbf{W} \subseteq \mathbf{S}$ is non-contradictory if for all entities $a \in S$ it holds that $\{a, \bar{a}\} \nsubseteq \mathbf{W}$. A non-contradictory state $\mathbf{W} \subseteq \mathbf{S}$ is consistent if, for any entity $a \in S$, either $a \in \mathbf{W}$ or $\bar{a} \in \mathbf{W}$ holds.

**Definition** (Positive RS)**.** A Positive RS is a Reaction System $\mathcal{A}^+ = (\mathbf{S}, A)$ that satisfies the following conditions:

1. Each reaction $r$ in $A$ is positive, i.e., $r = (\mathbf{R}, \emptyset, \mathbf{P})$ for some non-contradictory sets $\mathbf{R}$ and $\mathbf{P}$.

2. Consistency preservation: for any consistent state $\mathbf{W}$, the result set $res_{\mathcal{A}^+}(\mathbf{W})$ must be consistent.

If one assumes that the initial state $\mathbf{D}_0 \subseteq \mathbf{S}_D$ is a non-contradictory set and that the sets $\mathbf{C}_0, \ldots, \mathbf{C}_n \subseteq \mathbf{S}_C$ provided by the context are non-contradictory sets, the second condition of 2 guarantees that all result states traversed by the computation will be consistent as well.

### 2.3.1 From RSs to Positive RSs

For each standard RS $\mathcal{A} = (S, A)$ it is possible to construct a Positive RS $\mathcal{A}^+ = (\mathbf{S}, A^+)$ that exactly mimic the behavior of $\mathcal{A}$. The reactions in $A^+$ can be split in two categories: $A^+_{pos}$ that simply embeds the original reactions $A$ and $A^+_{neg}$ whose reactions serve for negative entities bookkeeping.

For each reaction $(R, I, P) \in A$ there will be one positive reaction $(R \cup \bar{I}, P) \in A^+_{pos}$. Extra reactions are needed to track the absence of the products of the original reactions. They will be produced whenever no reaction in $A$ that produces $a$ is enabled. For this purpose, assume to collect all reactions in $A$ that are capable of producing $a$: to ensure that none of them are enabled, we must make sure that, for each one, at least one reactant is absent or at least one inhibitor is present.

**Definition** (Prohibiting set)**.** Let $\mathcal{A} = (S, A)$ be RS and $a \in S_D$ one of its entities. A non-contradictory set $\mathbf{T} \subseteq \mathbf{S}$ is a prohibiting set for $a$ if for any reaction $(R, I, P \cup a) \in A$ we have that $\mathbf{T} \cap (I \cup \bar{R}) \neq \emptyset$. Denote the set of prohibiting sets for $a$ with $Proh_{\mathcal{A}}(a)$.

**Definition** (Encoding RSs into PRSs)**.** Let $\mathcal{A} = (S, A)$ be a RS, its encoding into a Positive RS is obtained by considering $\mathcal{A}^+ := (\mathbf{S}, A^+)$ whose set of positive reactions $A^+ := A^+_{pos} \cup A^+_{neg}$ is defined as follows:

$$A^+_{pos} := \left\{(R \cup \bar{I}, P) | (R, I, P) \in A\right\}$$
$$A^+_{neg} := \bigcup_{a \in S} \{(\mathbf{T}, \bar{a}) | \mathbf{T} \in Proh_{\mathcal{A}}(a)\}$$

The resulting $\mathcal{A}^+$ satisfies the two conditions from Definition  and thus is a Positive RS.

The states of the new Positive RS are in bijection with the states of the old system and can be proven that the two systems compute exactly the same states at each step.

### 2.3.2 Minimization

The procedure of converting a RS into a Positive RS can produce a system with many redundant reactions. The following rules are used to minimize the reactions after they are computed:

1. The reaction $r_1 = (R_1, I_1, P)$ can be omitted if a reaction $r_2 = (R_2, I_2, P)$ such that $R_2 \subseteq R_1$ and $I_2 \subseteq I_1$ is present.

2. If both reactions $r_1 = (R \cup \{a\}, I, P)$ and $r_2 = (R, I \cup \{a\}, P)$ are present, they can be replaced by $r = (R, I, P)$.

In general one can apply a minimization process to both standard and Positive RS and derive a simplified version of the original system with fewer reactions.

### 2.3.3 Slicing

In the context of programming, dynamic slicing is a technique that helps a user to debug a program by simplifying a partial execution trace, by pruning parts which are irrelevant and highlighting parts of the program wich are responsible for the production of an error. In the case of RSs, the goal is to highlight how a subset of the elements in a state were originated. This include the reactants and reactions that were responsible for producing them.

**Algorithm 1** Trace Slicer

**Input:**- a reaction system $\mathcal{A}$
- a trace $T = \frac{D_0}{C_0} \xrightarrow{N_1} \cdots \xrightarrow{N_m} \frac{D_m}{C_m}$
- a marking $D_\sigma \subseteq D_m$

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

$D'_m := D_\sigma$
**for** $i = \{m, m-1, \ldots, 1\}$ **do**
  $D'_{i-1} := \emptyset$
  $C'_{i-1} := \emptyset$
  $N'_i := \emptyset$
  **for** $r_j = (R_j, I_j, P_j) \in N_i$ such that $(D'_i \cap P_j \neq \emptyset)$ **do**
    $N'_i := N'_i \cup \{j\}$
    $C'_{i-1} := C'_{i-1} \cup (R_j \cap S_C)$
    $D'_{i-1} := D'_{i-1} \cup (R_j \cap S_D)$
  **end for**
**end for**

Starting from the pair $\frac{D_\sigma}{C_m}$ denoting the user's marking and proceeding backwards, apply iteratively a slicing step that deletes from the partial computation all information not related to $D_\sigma$. The sliced trace will contain only the subsets of entities and reactions which are necessary for deriving the marked entities.

Since the algorithm 1 can only capture dependencies related to reactants, but ignores the ones related to inhibitors, converting the RS into a Positive RS makes possible the tracking of the absence of entities via negative entities. Minimizing the Positive RS reduces the noise in the output and is thus desirable.

## 2.4 Bisimulation

Given two distinct RS processes, the natural question to ask would be if their simulation is the same, or at least behaves the same. Bisimulation is one such relation, defined in terms of coinductive games, of fixed point theory and of logic. Bisimulation equivalence aims to identify transitions systems with the same branching structure, and wich thus can simulate each other in a stepwise manner.

**Definition** (Transition System[2])**.** A transition system $TS$ is a tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S$ is a set of states,
- $Act$ is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- $AP$ is a set of atomic propositions,

- $L : S \to 2^{AP}$ is a labeling function.

$TS$ is called finite if $S$, $Act$, and $AP$ are finite.

The intuitive behavior of a transition system can be described as follows: the transition system start in some initial state $s_0 \in I$ and evolves according to the transition relation $\to$. Given $s$ as the current state, then a transition $s \xrightarrow{\alpha} s'$ is selected nondeterministically and taken, meaning the action $\alpha$ is performed and the transition system evolves from state $s$ into the state $s'$. The labeling function $L$ relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state $s$. It intuitively stands for exactly those aotmic propositions $\alpha \in AP$ which are satisfied by state $s$.

**Definition** (Bisimulation Equivalence[2])**.** Let $TS_i = (S_i, Act_i, \to_i, I_i, AP, L_i), i \in \{1, 2\}$, be transition systems over AP. A bisimulation for $(TS_1, TS_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that:

- $\forall s_1 \in I_1(\exists s_2 \in I_2.(s_1, s_2) \in \mathcal{R})$ and $\forall s_2 \in I_2(\exists s_1 \in I_1.(s_1, s_2) \in \mathcal{R})$
- for all $(s_1, s_2) \in \mathcal{R}$ it holds:
  1. $L_1(s_1) = L_2(s_2)$
  2. if $s_1' \in Post(s_1)$ then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$
  3. if $s_2' \in Post(s_2)$ then there exists $s_1' \in Post(s_1)$ with $(s_1', s_2') \in \mathcal{R}$.

$TS_1$ and $TS_2$ are bisimulation-equivalent (bisimilar), denoted $TS_1 \; TS_2$, if there exists a bisimulation $\mathcal{R}$ for $(TS_1, TS_2)$.

Where $Post(s)$ is the set of successors of $s$ defined as

$$Post(s) := \bigcup_{\alpha \in Act} \left\{ s' \in S \middle| s \xrightarrow{\alpha} s' \right\}$$

An intuitive way to see bisimulation is by framing it as a game between an attacker and a defender: the attacker wants to disprove the equivalence between two processes $s$ and $t$, the latter tries to show that $s$ and $t$ are equivalent. Each turn the attacker picks one process and one transition $s \xrightarrow{\alpha} s'$, the defender must reply by picking one transition $t \xrightarrow{\alpha} t'$ of the other process with exactly the same label $\alpha$. The game ends either with the attacker winning by finding a transition with no equivalent one in the other process or with the attacker losing by having no transitions available.

### 2.4.1 Algorithms for evaluating bisimulation

Follows a definition of a partition, used extensively in the following algorithms:

**Definition** (Partition)**.** A partition of $S$ is a set $\{B_0, \ldots, B_k\}, k \geq 0$ of non-empty subsets of $S$ such that:

- $B_i \cap B_j = \emptyset$, for all $0 \leq i < j \leq k$,
- $S = B_0 \cup B_1 \cup \ldots \cup B_k$.

An equivalence relation over the set of states $S$ can be represented as a partition of the states. The sets $B_i$ are called blocks.

Let $\pi$ and $\pi'$ be two partitions of $S$. $\pi'$ is a refinement of $\pi$ if for each block $B' \in \pi'$ there exists some block $B \in \pi$ such that $B' \subseteq B$.

**The algorithm of Kanellakis and Smolka[1]** Given a transition system $T = (S, Act, \rightarrow, I, AP, L)$, let $\pi = \{B_0, \ldots, B_k\}, k \geq 0$ be a partition of the set of states $S$. The algorithm due to Kanellakis and Smolka is based on the notion of splitter.

**Definition** (Splitter)**.** A splitter for a block $B_i \in \pi$ is a block $B_j \in \pi$ such that, for some action $\alpha \in Act$, some states in $B_i$ have $\alpha$-labeled transitions whose target is a state in $B_j$ and others do not.

Intuitively, thinking of blocks as representing approximations of equivalence classes of processes with respect to strong bisimilarity, the existence of a splitter $B_j$ for a block $B_i$ in the current partition indicates that we have a reason for distinguishing two groups of sets of states in $B_i$, namely those that afford an $\alpha$-labeled transition leading to a state in $B_j$ and those that do not. Therefore $B_i$ can be split by $B_j$ with respect to action $\alpha$ into the two new blocks:

$$B_i^1 = \left\{ s \middle| s \in B_i \text{ and } s \xrightarrow{\alpha} s' \text{ for some } s' \in B_j \right\} \text{ and}$$
$$B_i^2 = B_i \setminus B_i^1.$$

This splitting results in the new partition:

$$\pi' = \left\{ B_o, \ldots, B_{i-1}, B_i^1, B_i^2, B_{i+1}, \ldots, B_k \right\}$$

which is a refinement of $\pi$.

The algorithm of Kanellakis and Smolka iterate the splitting of some blocks $B_i$ by some blocks $B_j$ with respect to some action $\alpha$ until no further refinement of the current partition is possible. The resulting partition coincides with bisimilarity over the input labeled transition systems when the initial partition $\pi_{\text{initial}}$ is chosen to be equal to $S$ and is called the coarsest stable partition.

**Definition** (Stable Partition and Coarsest Stable Partition)**.** A set $B \subseteq S$ is stable with respect to a set $I \subseteq S$ if either $B \subseteq pre(I)$ or $B \cap pre(I) = \emptyset$.

A partition $\pi$ is stable with respect to a set $I$ if each block $B \in \pi$ is stable with respect to $I$.

A partition $\pi$ is stable with respect to a partition $\pi'$ if $\pi$ is stable with respect to each block $B' \in \pi'$. A partition $\pi$ is stable if it is stable with respect to itself.

The coarsest stable refinement of a partition $\pi_{\text{initial}}$ is a stable partition that is refined by any other stable partition that refines $\pi_{\text{initial}}$.

Note that $B \subseteq S$ is stable with respect to a block $C \subseteq S$ if and only if $C$ is not a splitter for $B$.

Follows the pseudocode for the algorithm of Kanellakis and Smolka[1].

The algorithm uses the function `split`$(B, \alpha, \pi)$ which given a partition $\pi$, a block $B \in \pi$ and an action $\alpha$, splits $B$ with respect to each block in $\pi$ and action $\alpha$.

```
1: π := S
2: changed := true
3: while changed do
4:     changed := false
5:     for each block B ∈ π do
6:         for each action α do
7:             sort the α-labeled transitions from states in B
8:             if split(B, α, π) = {B1, B2} ≠ {B} then
9:                 refine π by replacing B with B1 and B2
10:                changed := true
11:            end if
12:        end for
13:    end for
14: end while
```

```
1: procedure SPLIT(B, α, π)
2:     choose some state s ∈ B
3:     B1, B2 := ∅
4:     for each state t ∈ B do
5:         if s and t can reach the same set of blocks in π via α-labeled transitions then
6:             B1 := B1 ∪ {t}
7:         else
8:             B2 := B2 ∪ {t}
9:         end if
10:    end for
11:    if B2 is empty then
12:        return {B1}
13:    else
14:        return {B1, B2}
15:    end if
16: end procedure
```

**Theorem** (Kanellakis and Smolka)**.** When applied to a finite labeled transition system with $n$ states and $m$ transitions, the algorithm of Kanellakis and Smolka computes the partition corresponding to bisimilarity in time $O(n \cdot m)$. □

Proof of correctness relies on the fact that when *changed* is false, there is no splitter for any of the blocks in $\pi$. Moreover, if we denote by $\pi_i$ the partition after the $i$-iteration of the main loop, we have $\subseteq \ _i \subseteq \pi_i$. Thus the algorithm terminates with $\pi = $ .

**The algorithm of Paige and Tarjan[4]** Performance of the algorithm by Kanellakis and Smolka can be significantly improved through the use of more complex data structures. Paige and Tarjan proposed an algorithm that utilizes information about previous splits to make future splits more efficient. A simple algorithm over a one symbol alphabet is presented, followed by an algorithm that converts any LTS into a one symbol LTS.

The Paige-Tarjan algorithm is based on the following observation. Let $B$ be stable with respect to $S$, and let $S$ be partitioned into $S_1$ and $S_2$. Then, if $B \cap S = \emptyset$, $B$ is stable with respect to both $S_1$ and $S_2$. Otherwise $B$ can be split into three blocks:

$$\begin{aligned} B_1 &= B \setminus pre(S_2), \\ B_{12} &= B \cap pre(S_1) \cap pre(S_2), \\ B_2 &= B \setminus pre(S_1). \end{aligned}$$

The improvement in complexity that the Paige-Tarjan algorithm provides over the algorithm by Kanellakis and Smolka stems from the fact that three-way splitting can be performed in time proportional to the size of the smaller of the two blocks $S_1, S_2$.

Two types of splitter can be identified: simple and compound splitters.

*simple splitters* are used to split blocks in $\pi$ into two disjoint subsets as done in the algorithm of Kanellakis and Smolka.

**Definition** (Simple splitting)**.** Let $\pi$ be a partition and let $B$ be a set of states in $S$. Define $\texttt{split}(B, \pi)$ as the following procedure:

For each block $B' \in \pi$ such that $B'$ is not stable with respect to $B$, replace $B'$ by the blocks

$$\begin{aligned} B'_1 &= B' \cap pre(B) \quad \text{and} \\ B'_2 &= B' \setminus pre(B). \end{aligned}$$

$B$ is a splitter for $\pi$ when $\texttt{split}(B, \pi) \neq \pi$, in which case $\pi$ is refined with respect to $B$ and $\texttt{split}(B, \pi)$ is the partition that results from that refinement.

Some useful properties follow:

**Lemma.** [1]

1. Stability is preserved by refinement: if $\pi$ refines $\pi'$ and $\pi'$ is stable with respect to a set of states $I$, then so is $\pi$.

2. Stability is preserved by union: if $\pi$ is stable with respect to sets $I_1$ and $I_2$, then $\pi$ is also stable with respect to $S_1 \cup S_2$.

3. Assume that $B \subseteq S$. Let $\pi_1$ and $\pi_2$ be two partitions of $S$ such that $\pi_1$ refines $\pi_2$. Then $\texttt{split}(B, \pi_1)$ refines $\texttt{split}(B, \pi_2)$.

4. Assume that $B, B' \subseteq S$. Let $\pi$ be a partition of $S$. Then

$$\texttt{split}(B, \texttt{split}(B', \pi)) = \texttt{split}(B', \texttt{split}(B, \pi))$$

□

In order to implement the algorithm efficiently, it is useful to reduce the problem to that of considering a labeled transition system without deadlocked states, meaning without states with no outgoing edge. This can be done easily by preprocessing the initial partition $\pi_{intial}$ by splitting each block $B \in \pi_{initial}$ into:

$$B_1 = B \cap pre(S) \quad \text{and}$$
$$B_2 = B \setminus pre(S).$$

$B_2$ will never be split again by the refinement algorithm. Therefore run the refinement algorithm starting from the partition $\pi'_{initial} = \{B_1 | B \in \pi_{initial}\}$.

In order to find splitters efficiently, some additional information is kept. The algorithm maintains another partition $X$ such that

- $\pi$ is a refinement of $X$ and
- $\pi$ is stable with respect to $X$.

Initially $X = \{S\}$. Follows a general outline of the algorithm.

---

1: **while** $\pi \neq X$ **do**
2: Find a block $I \in X \setminus \pi$
3: Find a block $B \in \pi$ such that $B \subseteq S$ and $|B| \leq \frac{|I|}{2}$
4: Replace $I$ withing $X$ with the two sets $B$ and $I \setminus B$
5: Replace $\pi$ with $\texttt{split}(I \setminus B, \texttt{split}(B, \pi))$.
6: **end while**

---

The efficiency of the above algorithm relies on the heuristic for the choice of the block $B$ at line 3 and on the use of three-way splitting to implement line 5 efficiently.

Suppose that we have a partition $\pi$ that is stable with respect to a set of states $I$ that is a union of some of the blocks in $\pi$. Assume also that $\pi$ is refined first with respect to a non-empty set $B \subset I$ and then with respect to $I \setminus B$. Two properties can be observed:

- Refining $\pi$ with respect to $B$ splits a block $D \in \pi$ into two blocks $D_1 = D \cap pre(B)$ and $D_2 = D \setminus pre(B)$ if, and only if, $D$ is not stable with respect to $B$.

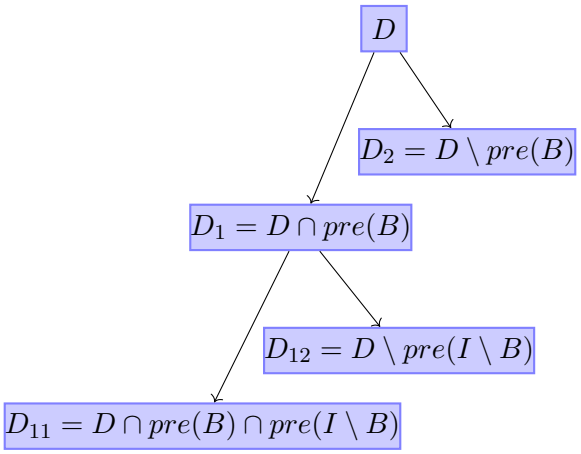


Figure 2.2: Three-way splitting of a block.

- Refining further $\texttt{split}(B, \pi)$ with respect to $I \setminus B$ splits the block $D_1$ into two blocks $D_{11} = D_1 \cap pre(S \setminus B)$ and $D_{12} = D_1 \setminus D_{11}$ if, and only if, $D_1$ is not stable with respect to $S \setminus B$.

A block $I$ is simple if it is also a block of $\pi$ and is compound otherwise. Note that a compound block $I$ contains at least two blocks of $\pi$. A compound block can be partitioned into $B$ and $I \setminus B$ in such a way that both of the properties 2.4.1 hold. If $\pi$ is stable with respect to $I$, either $D \subseteq pre(I)$ or $D \cap pre(I) = \emptyset$. If $D$ is not stable with respect to $B$, it holds that $D \nsubseteq pre(B)$ and $D \cap pre(B) \neq \emptyset$. Therefore, $D \subseteq pre(I)$. One can thus infer that $D_{12} = D_1 \setminus D_{11} = D_1 \cap (pre(B) \setminus pre(I \setminus B))$, which is the crucial observation underlying the implementation of the algorithm.

Depicted in 2.2 the result of a three-way split of a block $D$.

The time performance of the algorithm by Paige and Tarjan relies on the following observations:

- Each state in the input labeled transition system is in at most $\log(n+1)$ blocks $B$ used as refining sets, since each is at most half the size of the previous one.
- A refinement step with respect to a block $B$, as shown by Paige and Tarjan[5], can be implemented in time $O\left(|B| + \sum_{b \in B} |pre(b)|\right)$ by means of use of appropriate data structures.

The algorithm has thus an $O(m \log n)$ time bound.

**log-space reduction of bisimilarity checking over a one-letter action set** In order to apply the previous algorithm to a LTS with arbitrary number of labels, there needs to be first a reduction to an equivalent LTS with only one symbol as label. Assume two given processes $P$ and $Q$ over an LTS $T$ with the set of actions $\{\alpha_1, \alpha_2, \ldots, \alpha_l\}$. Let $T'$ be the modified LTS which contains all the process of $T$ together with some additional ones defined in the following way: for every transition $P_1 \xrightarrow{\alpha_i} P_2$ in $T$ we add into $T'$

- two transitions $P_1 \to P_{(P_1,\alpha_i,P_2)}$ and $P_{(P_1,\alpha_i,P_2)} \to P_2$ where $P_{(P_1,\alpha_i,P_2)}$ is a newly added state, and
- a newly added path of length $i$ from $P_{(P_1,\alpha_i,P_2)}$.

Finally for every process $P$ in $T$ we create in $T'$ a newly added path of length $l+1$ starting from $P$. A small optimization can be added by sorting the frequency of labels and thus creating the lowest possible number of auxiliary nodes for each label.

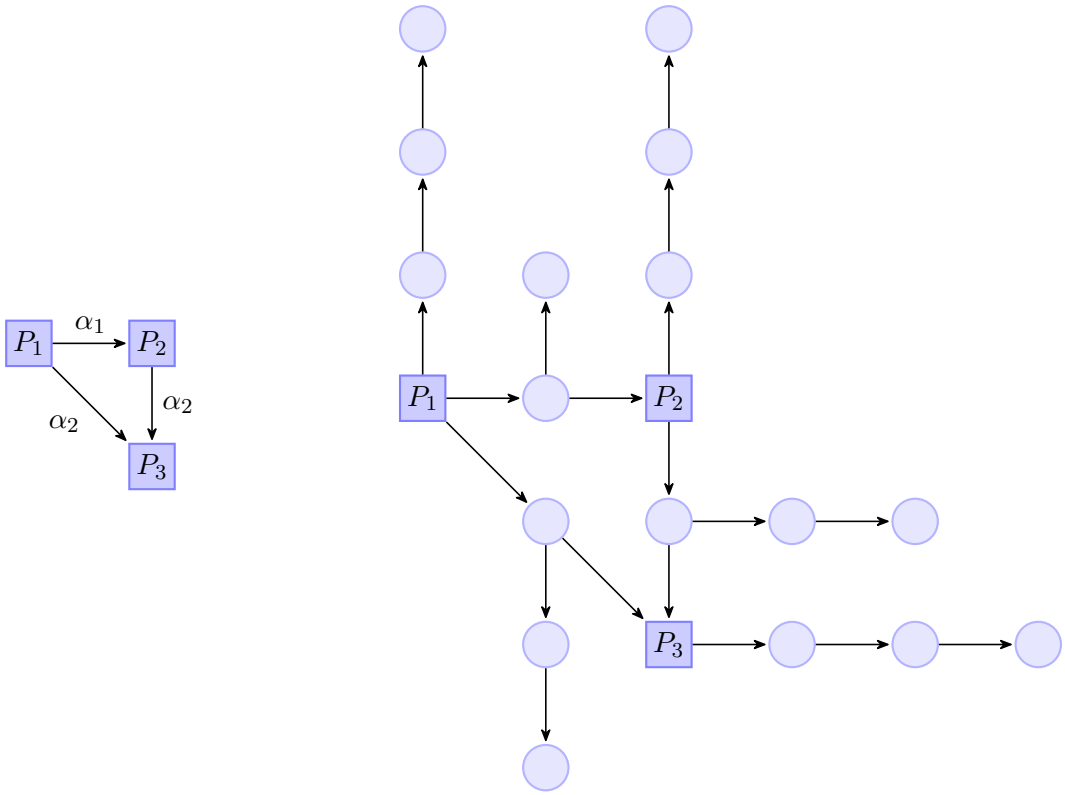


Figure 2.3: Example of reduction

# Chapter 3

# Design

Two sub-problems where identified during the design: simulating the behavior of Reaction Systems, RS processes and other operations on LTS, and interacting with the user in a intuitive manner. The programming language chosen was Rust, since it offered good performance and ease of development. Two Git repositories are provided: github.com/elvisrossi/ReactionSystems and github.com/elvisrossi/ReactionSystemsGUI.

The ReactionSystems project follows a modular architecture and clear design principles to mirror the theoretical model; it implements procedures over RS as pure rust functions and is structured as a library. It also provides a crude Command Line Interface for some of the functions provided. The code is organized in workspaces in order to reduce compilation time and aid code reuse. In the second Git repository a native and web application is implemented in Rust and in webassembly generated from Rust code. The web application consists of only static files and as such may be served by a simple HTTP server.

## 3.1 ReactionSystems

The design is structured to faithfully implement the reaction system formalism while remaining flexible. It provides a foundation that matches theoretical definitions (ensuring correctness) and supports further expansion (such as adding optimization, visualization, or integration with other tools) by maintaining a clean separation between the model representation and the execution logic. Since the language Rust supports object-oriented programming via traits, but lacks generic inheritance, the design of the basic building blocks of RSs are designed around this limitation. Usually a basic trait is provided for each of them and an extension of the trait is implemented for all structures that implement the basic trait.

Since it is not practical for a user to specify the structures in Rust, a syntax for the basic structures has been specified. This syntax tries to remain as much as possible compatible with ones from previous software. To develop the parser, LALRPOP was chosen as the parser generator framework. LALRPOP code is transpiled to Rust code

via macros and then compiled to machine code.

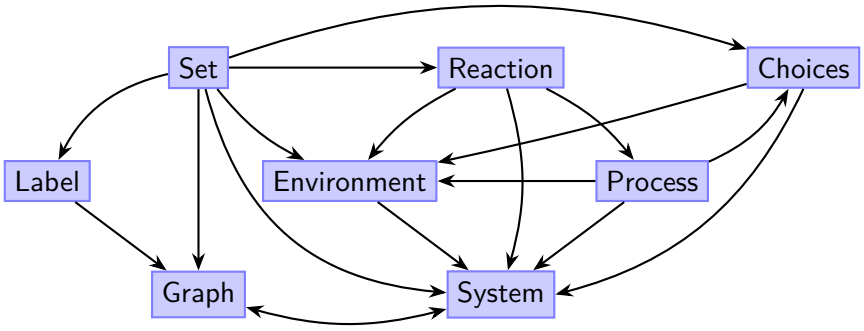


Figure 3.1: Basic structures and relationships between them

### 3.1.1 Entities and Translator

Entities are the most basic data structure that a RS need to keep track of. They don't have a specified interface and are instead treated only in sets.

Positive elements are also defined and have a state, either `Positive` or `Negative`.

Since internally entities are represented as integers, a structure that keeps track of assignment between strings and integer is provided (`Translator`). This poses a problem with the default methods for formatting available in Rust, since for the trait `Display` and `Debug` only the structure itself can be used to generate the string. The trait `PrintableWithTranslator` and the structure `Formatter` solve this issue by incorporating the `Translator` into the struct. `Display` is then implemented on the generic structure `Translator`.

### 3.1.2 Set

The common procedures required for all sets are:

- `is_subset`$(a, b) \to$ `bool`, which should return true if $a \subseteq b$;
- `is_disjoint`$(a, b) \to$ `bool`, which should return true if $a \cap b = \emptyset$;
- `union`$(a, b) \to$ `set`, which should return $a \cup b$;
- `push`$(a, b)$, which should replace $a$ with $a \cup b$ in place;
- `intersection`$(a, b) \to$ `set`, which should return $a \cap b$;
- `subtraction`$(a, b) \to$ `set`, which should return $a \setminus b$;
- `len`$(a) \to$ `int`, which should return the number of elements in $a$;
- `is_empty`$(a) \to$ `bool`, which should return true if $a$ has no elements, false otherwise;
- `contains`$(a, e :$ `entity`$) \to$ `bool`, which should return $e \in a$;
- `add`$(a, e :$ `element`$)$, which should add the element $e$ to $a$ in place.

Some other procedures are required for ease of use:

- `extend`$(a, b^?)$, which should extend $a$ with $a \cup b$ if $b$ is a non-null value, and leave $a$ unchanged otherwise, similar to `push` (3.1.2).

Two other procedures are implemented for all structs that implement the BasicSet trait:

- `iter`$(a) \to$ `iterable`, which returns an iterator over the elements of the set $a$;
- `split`$(a, \mathit{trace} :$ `[set]`$) \to$ (`[set]`, `[set]`)$^?$, which returns the prefix and the loop part of a trace.

Both normal sets and positive sets satisfy this interface, but have additional specific functions for converting between the two.

The syntax for sets is:

| | | |
|---|---|---|
| *Set* | ::= | `{`$S$`}` |
| | \| | `{}` |
| $S$ | ::= | $s$ `,` $S$ |
| | \| | $s$ |
| Where | | |
| $s$ | | is a string |

**Syntax** 3.1: Syntax for Set

| | | |
|---|---|---|
| *PositiveSet* | ::= | `{`$S$`}` |
| | \| | `{}` |
| $S$ | ::= | *state* $s$ `,` $S$ |
| | \| | *state* $s$ |
| *state* | ::= | `+` |
| | \| | `-` |
| Where | | |
| $s$ | | is a string |

**Syntax** 3.2: Syntax for Positive Set

### 3.1.3 Reaction

The methods required for all reactions are:

- `enabled`$(r, \mathit{state} :$ `set`$) \to$ `bool`, which returns true if the reaction is enabled given the entities supplied by *state*;
- `compute_step`$(r, \mathit{state} :$ `set`$) \to$ `set`$^?$, which returns the products of the reaction if it is enabled by *state*.

All reactions that satisfy the basic trait automatically implement the following methods:

- `find_loop`$(\mathit{rs} :$ `[reaction]`$, \mathit{entities} :$ `set`$, q :$ `set`$) \to$ `([set], [set])`, which finds a loop and returns the sets that make up the prefix and the loop separately;
- `find_only_loop`$(\mathit{rs} :$ `[reaction]`$, \mathit{entities} :$ `set`$, q :$ `set`$) \to$ `[set]`, which finds a loop and returns the sets that form it;
- `find_prefix_len_loop`$(\mathit{rs} :$ `[reaction]`$, \mathit{entities} :$ `set`$, q :$ `set`$) \to$ `(integer, [set])`, which finds a loop and returns the length of the prefix and the sequence of sets that compose the loop;

| | | |
|---|---|---|
| *Reaction* | ::= | `[`$s$`,` $s$`,` $s$`]` |
| | \| | `[r:` $s$`,` `i:` $s$`,` `p:` $s$`]` |

Where

| | |
|---|---|
| $s$ | is a set, see 3.1 |

**Syntax** 3.3: Syntax for Reaction

| | | |
|---|---|---|
| *PositiveReaction* | ::= | `[`$s$`,` $s$`,` $s$`]` |
| | \| | `[r:` $s$`,` `i:` $s$`,` `p:` $s$`]` |

Where

| | |
|---|---|
| $s$ | is a positive set, see 3.2 |

**Syntax** 3.4: Syntax for Positive Reaction

### 3.1.4 Process

Process structures mirror the structure of RS processes as described in Section 2.2. Since there is not much behavior that is shared between implementations and since usually they are used with pattern matching, the trait that describe a basic process is very simple.

- `concat`$(a, b) \rightarrow$ `process`, which returns a new process $a|b$ flattened with regards to parallel composition;
- `all_elements`$(a) \rightarrow$ `set`, which returns all the entities used in the process;
- `filter_delta`$(a, id : \texttt{entity}) \rightarrow \texttt{set}^?$, which returns the first rule $X = Q.rec(X)$ for any symbol $X$.

| | | |
|---|---|---|
| *Process* | ::= | `[`$P$`]` |
| $P$ | ::= | $C$`,`$P$ |
| | \| | $C$ |
| $C$ | ::= | `(`$C$`)` |
| | \| | `nill` |
| | \| | $s$`.`$C$ |
| | \| | $C$`+`$C$ |
| | \| | `?`$r$`?.`$C$ |
| | \| | `<`$i$`,`$C$`>.`$C$ |
| | \| | $x$ |
| Where | | |
| $s$ | | is a set, see 3.1 |
| $r$ | | is a reaction, see 3.3 |
| $i$ | $\in$ | $\mathbb{N}$ |
| $x$ | | is a variable name |

**Syntax** 3.5: Syntax for Process

| | | |
|---|---|---|
| *PositiveP* | ::= | `[`$P$`]` |
| $P$ | ::= | $C$`,`$P$ |
| | \| | $C$ |
| $C$ | ::= | `(`$C$`)` |
| | \| | `nill` |
| | \| | $ps$`.`$C$ |
| | \| | $C$`+`$C$ |
| | \| | `?`$pr$`?.`$C$ |
| | \| | `<`$i$`,`$C$`>.`$C$ |
| | \| | $x$ |
| Where | | |
| $ps$ | | is a positive set, see 3.2 |
| $pr$ | | is a positive reaction, see 3.4 |
| $i$ | $\in$ | $\mathbb{N}$ |
| $x$ | | is a variable name |

**Syntax** 3.6: Syntax for Positive Process

### 3.1.5 Choices

Since one RS process may have more than one possible next system when evaluating, there is a need to express all possible choices for next states. The structure choices represents all those possible continuations, associating a set with a process. The set signifies all the entities that are provided by the context by choosing that context. One particular operation called `shuffle` is needed: given two choices structures $c_1$ and $c_2$ where $c_i$ : `set` $\rightharpoonup$ `process`, $i \in \{1, 2\}$, it generates a new choices structure $c'$ such that $\forall s_1 \in \mathit{domain}(c_1).(\forall s_2 \in \mathit{domain}(c_2).\mathit{domain}(c') \ni (s_1 \cup s_2) \wedge c'(s_1 \cup s_2) =$ `concat`$(c_1(s_1), c_2(s_2)))$. Intuitively it is all the possible combinations of two parallel processes.

### 3.1.6 Environment

An environment can be thought as an association between variable names and processes. The basic interface requires the following methods:

- `get`$(a, k$ : `entity`$) \rightarrow$ `process`, which returns the process associated with the variable $k$;
- `all_elements`$(a) \rightarrow$ `set`, which returns all the entities used in any of the processes;
- `unfold`$(a, \mathit{context}$ : `process`$, s$ : `set`$) \rightarrow$ `choices`$^?$, which returns the list of choices for the context, given the process definitions environment and is used to generate the next systems with the SOS rules.

Some methods are automatically implemented for all `BasicEnvironment`: `lollipops_decomposed`, `lollipops_prefix_len_loop_decomposed`, `lollipops_only_loop_decomposed`, `lollipops_decomposed_named`, `lollipops_prefix_len_loop_decomposed_named`, and `lollipops_only_loop_decomposed_named`. They all try to find a loop and return some information about the found loop. The `_named` variants require a variable symbol for which in the environment there is an association to a process with the form $X = Q.\texttt{rec}(X)$, where $Q$ is a set.

| | | |
|---|---|---|
| *Environment* | ::= | `[`$E$`]` |
| $E$ | ::= | $x = c$`,`$E$ |
| | \| | $x = c$ |

Where

| | |
|---|---|
| $c$ | is a process, see 3.5 |
| $x$ | is a variable name |

**Syntax** 3.7: Syntax for Environment

| | | |
|---|---|---|
| *PositiveEnvironment* | ::= | `[`$E$`]` |
| $E$ | ::= | $x = pc$`,`$E$ |
| | \| | $x = pc$ |

Where

| | |
|---|---|
| $pc$ | is a positive process, see 3.6 |
| $x$ | is a variable name |

**Syntax** 3.8: Syntax for Positive Environment

### 3.1.7 System

The basic interface for systems is only the following methods:

- `to_transitions_iterator`$(sys) \rightarrow$ `iterator over (label, system)`
- `to_slicing_iterator`$(sys) \rightarrow$ `iterator over (label, system)`
- `context_elements`$(sys) \rightarrow$ `set`
- `products_elements`$(sys) \rightarrow$ `set`

The method `to_transitions_iterator` should return an iterator over all the possible evaluations of the system. Likewise `to_slicing_iterator` should return an iterator over the same outgoing edges, but with information that support the creation of a trace to be used for slicing.

The two methods `context_elements` and `products_elements` should return the set of entities that are related to the context and the one related to the reactions. Since it may be a computationally expensive calculation, the result is cached in the structures. Since there may be errors when deciding what constitutes an element belonging to the context, methods are also present to override the default values.

Other methods are implemented for all structures that satisfy the previous interface:

- `unfold`($sys$) → `choices`$^?$, which, by calling the same method of the environment, returns the list of choices for the context;
- `run`($sys$) → `[system]`$^?$, which computes the sequence of systems for the leftmost execution;
- `digraph`($sys$) → `graph`$^?$, which computes the graph generated by the execution of the system;
- `target`($sys$) → `(integer, set)`$^?$, which returns the state in one of the terminal states and the number of steps to arrive at the last state;
- `slice_trace`($sys$) → `trace`$^?$, which generates, similarly to `run`, a trace appropriate to run slicing calculations over;
- `lollipops`($sys$) → `[([set], [set])]`, similar to the method `lollipops_decomposed` provided by `environment`.

| | | |
|---|---|---|
| *System* | ::= | `Environment:`$e$<br>`Initial Entities:`$s$<br>`Context:`$c$<br>`Reactions: (`$R$`)` |
| $R$ | ::= | $r$`;`$R$ |
| | \| | $\epsilon$ |

Where

| | |
|---|---|
| $e$ | is an environment, see 3.7 |
| $s$ | is a set, see 3.1 |
| $c$ | is a Process, see 3.5 |
| $r$ | is a reaction, see 3.3 |
| $\epsilon$ | is the empty string |

**Syntax** 3.9: Syntax for System

### 3.1.8 Label

The label structure holds the information about how entities are used in the production of a system and are the labels on the edges of the graphs. Since the only use is to hold data, no meaningful method is required.

*Label* ::= `[Entities:`*s*`,`
`Context:`*s*`,`
`Reactants:`*s*`,`
`ReactantsAbsent:`*s*`,`
`Inhibitors:`*s*`,`
`InhibitorsPresent:`*s*`,`
`Products:`*s*`]`

Where
*s* is a set, see 3.1

**Syntax** 3.10: Syntax for Label

*Label* ::= `[Entities:`*ps*`,`
`Context:`*ps*`,`
`Reactants:`*ps*`,`
`ReactantsAbsent:`*ps*`,`
`Inhibitors:`*ps*`,`
`InhibitorsPresent:`*ps*`,`
`Products:`*ps*`]`

Where
*ps* is a positive set, see 3.2

**Syntax** 3.11: Syntax for Positive Environment

### 3.1.9 Graph

The project uses petgraph as graph data structure library. *petgraph* provides several graph types, but the only one used is `Graph`, since it provided the best performance during testing. The library provides methods for converting the graph structures into Dot Language and GraphML File Format. The Dot methods where found to be not powerful enough and where partially rewritten in the file dot.rs.

Custom formatting of the graphs was a key requirement, so domain specific languages are provided to customize the appearance of the generated formats. Four structures are provided:

- `NodeDisplay`, to specify the text displayed on each node;
- `EdgeDisplay`, to specify the text displayed on each edge;
- `NodeColor`, to specify the color of each node;
- `EdgeColor`, to specify the color of each edge.

Follows the BNF for each of the languages:

| | | |
|---|---|---|
| *NodeDisplay* | ::= | *E* |
| | \| | *E* "*s*" *NodeDisplay* |
| *E* | ::= | `Hide` |
| | \| | `Entities` |
| | \| | `MaskEntities` *S* |
| | \| | `ExcludeEntities` *S* |
| | \| | `Context` |
| | \| | `UncommonEntities` |
| | \| | `MaskUncommonEntities` *S* |

Where

| | |
|---|---|
| *S* | is a sets of entities |
| *s* | is a possibly empty string |

**Syntax** 3.12: Syntax for `NodeDisplay`

`Hide` ignores the content of the node and prints the empty string, `Entities` prints the list of entities currently available in the system, `MaskEntities` $S$ prints the list of entities masked by a specified set $S$, `ExcludeEntities` $S$ prints the list of entities except for the entities specified by the set $S$, `Context` prints the context of the system, `UncommonEntities` prints only the entities that are not shared between all the nodes in the graph, `MaskUncommonEntities` $S$ prints the entities not shared between all the nodes in the graph and masked by a specified set $S$.

| | | |
|---|---|---|
| *EdgeDisplay* | ::= | *E* |
| | \| | *E* "*s*" *EdgeDisplay* |
| *E* | ::= | `Hide` |
| | \| | `Products` |
| | \| | `MaskProducts` $S$ |
| | \| | `UncommonProducts` |
| | \| | `UncommonMaskProducts` $S$ |
| | \| | `Entities` |
| | \| | `MaskEntities` $S$ |
| | \| | `UncommonEntities` |
| | \| | `UncommonMaskEntities` $S$ |
| | \| | `Context` |
| | \| | `MaskContext` $S$ |
| | \| | `UncommonContext` |
| | \| | `UncommonMaskContext` $S$ |
| | \| | `Union` |
| | \| | `MaskUnion` $S$ |
| | \| | `UncommonUnion` |
| | \| | `UncommonMaskUnion` $S$ |
| | \| | `Difference` |
| | \| | `MaskDifference` $S$ |
| | \| | `UncommonDifference` |
| | \| | `UncommonMaskDifference` $S$ |
| | \| | `EntitiesDeleted` |
| | \| | `MaskEntitiesDeleted` $S$ |
| | \| | `UncommonEntitiesDeleted` |
| | \| | `UncommonMaskEntitiesDeleted` $S$ |
| | \| | `EntitiesAdded` |
| | \| | `MaskEntitiesAdded` $S$ |
| | \| | `UncommonEntitiesAdded` |
| | \| | `UncommonMaskEntitiesAdded` $S$ |

Where

| | |
|---|---|
| $S$ | is a sets of entities |
| *s* | is a possibly empty string |

**Syntax** 3.13: Syntax for `EdgeDisplay`

Four version of each base option is available: normal, `Mask` which masks the normal set of entities with a specified set, `Uncommon` which considers only the entities that are not shared between all edges of the graph, and `UncommonMask` which combines the two functionalities. The base options return the corresponding entities available in the label.

| | | | |
|---|---|---|---|
| *NodeColor* | ::= | `!"` $C$ `"` | |
| | \| | $E$ `||` *NodeColor* | |
| $E$ | ::= | `Entities` *op* $S$ `?"`$C$`"` | |
| | \| | `Context.Nill ?"`$C$`"` | |
| | \| | `Context.RecursiveIdentifier (` $x$ `)?"`$C$`"` | |
| | \| | `Context.EntitySet` *op* $S$ `?"`$C$`"` | |
| | \| | `Context.NonDeterministicChoice ?"`$C$`"` | |
| | \| | `Context.Summation ?"`$C$`"` | |
| | \| | `Context.WaitEntity ?"`$C$`"` | |
| *op* | ::= | `==` | or `=` |
| | \| | `<` | or $\subset$ |
| | \| | `<=` | or $\subseteq$ |
| | \| | `>` | or $\supset$ |
| | \| | `>=` | or $\supseteq$ |

Where

| | |
|---|---|
| $C$ | is a string that specifies the color of the node |
| $S$ | is a sets of entities |
| $x$ | is a variable |

**Syntax** 3.14: Syntax for `NodeColor`

The `NodeColor` structure assigns the first correct color to the node. The structure can be thought of as a list of pairs; each pair has an entry that evaluated returns true or false, and an entry that holds the desired color of the node. To find the correct color, the list is scanned until the first pair that returns true and the color is assigned. If no pair returns true, a default value is assigned, specified after `!`. The possible functions expressible by the grammar are the ones expressed by $E$ and query either the entities available or the current context.

| | | | |
|---|---|---|---|
| *EdgeColor* | ::= | `!"` *C* `"` | |
| | \| | *E* `||` *EdgeColor* | |
| *E* | ::= | `Entities` *op S* `?"`*C*`"` | |
| | \| | `Context` *op S* `?"`*C*`"` | |
| | \| | `T` *op S* `?"`*C*`"` | |
| | \| | `Reactants` *op S* `?"`*C*`"` | |
| | \| | `AbsentReactants` *op S* `?"`*C*`"` | |
| | \| | `Inhibitors` *op S* `?"`*C*`"` | |
| | \| | `PresentInhibitors` *op S* `?"`*C*`"` | |
| | \| | `Products` *op S* `?"`*C*`"` | |
| *op* | ::= | `==` | or `=` |
| | \| | `<` | or $\subset$ |
| | \| | `<=` | or $\subseteq$ |
| | \| | `>` | or $\supset$ |
| | \| | `>=` | or $\supseteq$ |

Where

| | |
|---|---|
| *C* | is a string that specifies the color of the node |
| *S* | is a sets of entities |
| *x* | is a variable |

**Syntax** 3.15: Syntax for `EdgeColor`

`EdgeColor` behaves in a similar manner as `NodeColor`, except the base structure is a `Label`, so every field is a `Set`.

### 3.1.10 Slicing Trace

Only one structure for slicing trace is provided, but is made to work with both RS and Positive RS with generics. The only method they have is `slice`(*trace*, *marking* : `set`) $\rightarrow$ `trace`$^?$ which returns, if successful, a new sliced trace.

### 3.1.11 Bisimilarity and Bisimulation

In the workspace `bisimilarity` the algorithms by Kanellakis and Smolka, and Paige and Tarjan are implemented over generic graphs. Instead of an implementation over graphs with generic parameters, the input have to implement generic traits from the `petgraph` library, making it possible to use with different types of graph, for example spare graphs or matrix graphs.

One key feature was the ability to control via a domain specific language the labels on the edges of the graphs. The developed language is able to also specify values over nodes such that nodes with equal value may be collapsed into one node with outgoing and incoming edges inherited from the original nodes. The code for the typechecking and execution is available in the library `assert`.

The language has way to define subroutines or functions, has no while loop and limited for loop construction, so that the execution always terminates.

| | | | |
|---|---|---|---|
| *Assert* | ::= | `label{`*Tree*`}` | `label` is replaced by other strings to differentiate languages |
| *Tree* | ::= | *Tree*`;`*Tree* | |
| | \| | `if` *E* `then {`*Tree*`};` | |
| | \| | `if` *E* `then {`*Tree*`}` `else {`*Tree*`};` | |
| | \| | `let` *x* `=` *E*`;` | |
| | \| | `let` *x*`.`*Qualifier* `=` *E*`;` | |
| | \| | `return` *E*`;` | |
| | \| | `for` *x* `in` *Range* `{`*Tree*`};` | |
| *E* | ::= | *unaryP*`(`*E*`)` | |
| | \| | *E*`.`*unaryS* | |
| | \| | `(`*E* *binary* *E*`)` | |
| | \| | *binaryP*`(`*E*`,`*E*`)` | |
| | \| | *Term* | |
| *Term* | ::= | `true` | |
| | \| | `false` | |
| | \| | *x* | |
| | \| | *i* | |
| | \| | *l* | |
| | \| | *set* | |
| | \| | `'`*s*`'` | element |
| | \| | `"`*s*`"` | string |
| | \| | `(`*E*`)` | |
| *Range* | ::= | `{`*E*`}` | iterate over set |
| | \| | `{`*E*`..`*E*`}` | iterate over integer range |

Where

| | | |
|---|---|---|
| *S* | | is a sets of entities, see 3.1 |
| *i* | ∈ | ℤ |
| *x* | | is a variable name |
| *l* | | is a label, see 3.1 |
| *set* | | is a set, see 3.1 |
| *s* | | is a string |

**Syntax** 3.16: Syntax for `Assert`

Continues on the next page.

| | | | |
|---|---|---|---|
| *unaryP* | ::= | `empty` | |
| | \| | `length` | |
| | \| | `tostr` | |
| | \| | `toel` | string to element |
| | \| | `Entities` | |
| | \| | `Context` | |
| | \| | `Reactants` | |
| | \| | `ReactantsAbsent` | |
| | \| | `Inhibitors` | |
| | \| | `InhibitorsPresent` | |
| | \| | `Products` | |
| | \| | `AvailableEntities` | |
| | \| | `AllReactants` | |
| | \| | `AllInhibitors` | |
| | \| | `SystemEntities` | |
| | \| | `SystemContext` | |
| | \| | `source` | source of edge |
| | \| | `target` | target of edge |
| | \| | `neightbours` | node's neighbours |
| | \| | `system` | node's system |
| *binary* | ::= | `&&` | logical and, set intersection |
| | \| | `\|\|` | logical or, set union |
| | \| | `^^` | logical xor, set xor |
| | \| | `<` | less, set inclusion |
| | \| | `<=` | less equal, set inclusion or equal |
| | \| | `>` | greater, reverse set inclusion |
| | \| | `>=` | greater equal, reverse set inclusion or equal |
| | \| | `==` | |
| | \| | `!=` | |
| | \| | `+` | |
| | \| | `*` | |
| | \| | `^` | |
| | \| | `/` | quotient |
| | \| | `%` | reminder |
| | \| | `::` | concatenation |
| *binaryP* | ::= | `substr` | logical and |
| | \| | `min` | logical or |
| | \| | `max` | logical xor |
| | \| | `commonsubstr` | less or set inclusion |
| *unaryS* | ::= | `Entities` | |
| | \| | `length` | |
| | \| | `tostr` | |
| | \| | `toel` | |

**Syntax** 3.16: Syntax for `Assert` (Continued)

The template language requires two structures to function relating to the input of the language: a type structure and a value structure. The trait `SpecialVariables` holds all the necessary functions that need to be implemented for the special variables to function. Finally the generic language can have the two functions `typecheck` and `execute` implemented.

The language is very limited and is only designed for simple algorithms since there is no scoping. Typechecking consists in only asserting acceptable types for unary and binary functions, range declaration and for all return statements to return the same type.

A version for Positive RS is also provided and reflects the previous grammar with basic types replaced with their positive versions.

### 3.1.12 Grammar and Separated Grammar

Two workspaces are provided for parsing the structures above. `Grammar` creates only one endpoint that parses a system and a list of instructions. Those instructions are then executed via the library `execution`. A simple CLI has been implemented in the workspace `analysis`, with proper error formatting for LALRPOP errors.

### 3.1.13 Experiments and Frequency

An experiment is a list of weights and a list of sets of same length. The sets are used as entities given in addition to the context entities when computing the RS. The resulting trace is then synthesized into relative frequencies. The methods offered by `Frequency` and `PositiveFrequency` are:

- `naive_frequency`($sys$ : `system`) $\to$ `frequency`$^?$, which computes the relative frequency of each entity in all traversed states, assuming the computation is finite;
- `loop_frequency`($sys$ : `system`, $symbol$ : `IdType`) $\to$ `frequency`, which computes the relative frequency of each entity in each state of the encountered loop, assuming the system stabilizes in a loop;
- `limit_frequency`($experiment$ : `[set]`, $reactions$ : `[reaction]`, $entities$ : `set`) $\to$ `frequency`$^?$, which computes the relative frequency of each entity in the states of the last loop by providing repeatedly the sets in the experiment until the system stabilizes in a loop;
- `fast_frequency`($experiment$ : `[set]`, $reactions$ : `[reaction]`, $entities$ : `set`, $weights$ : `[int]`) $\to$ `frequency`$^?$, which computes the weighted relative frequency of each entity in any of the loops.

| | | |
|---|---|---|
| *Experiment* | ::= | `Weights :` $W$ `Sets :` $S$ |
| $W$ | ::= | $i$, $W$ |
| | \| | $\epsilon$ |
| $S$ | ::= | $s$, $S$ |
| | \| | $\epsilon$ |
| Where | | |
| $s$ | | is a sets of entities, see 3.1 |
| $i$ | $\in$ | $\mathbb{Z}$ |

**Syntax** 3.17: Syntax for Experiment

## 3.2 ReactionSystemsGUI

During development of ReactionSystems, a need for a more intuitive interaction with the structures presented itself. Since the all the operations on the types where already limited and structured, a visual programming language was chosen as the best fit.

The library `egui_node_graph2` was chosen since it offered customizability, performance and ease of programming. The library unfortunately lacked compatibility with the most recent version of `egui`, so it is included as a workspace and modified to fit better the need of the project. This way a couple of bugs present in the original code could be fixed.

`egui_node_graph2` is based on the library `egui`, which is an immediate mode GUI. It differentiate itself from retained mode GUIs by having all the elements specified at every frame; this eases programming at the expense of performance. The trade-off is favorable since most of the computation will be on the algorithms over RS and the number of elements of the UI will remain small in most cases.

All the functions previously described are available as "nodes" in the GUI program. Each takes one or more inputs, colored by type, and prevents wrong types from connecting, reducing user error when connecting similarly colored types.

Since at every step all of the GUI is recalculated, a robust cache structure is needed. The cache developed keeps track of the modified nodes and only recomputes if necessary, exploiting the structure of the graph.

The library `egui_node_graph2` was also chosen for its ability to create a web application directly from Rust code. The web application is limited; there is no interaction with the file system and no true multi-threading. These limitations are imposed by WebAssembly itself, not by the transpilation from Rust.

The native application executes the expressed instructions in a separate thread and returns the result to the GUI thread to be displayed. Thus the web application may "freeze" and become unresponsive with long calculations.

Both native and web applications have the ability to save the current state and resume. The saved state is stored in the browser cache in the web application and in special directories in the native one:

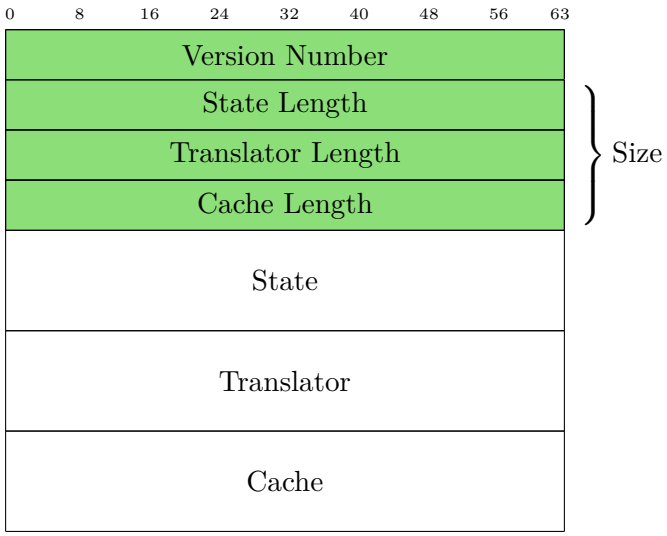


Figure 3.2: Save file structure

- Linux: `/home/UserName/.local/share/Reaction-Systems`
- macOS: `/Users/UserName/Library/Application Support/Reaction-Systems`
- Windows: `C:\Users\UserName\AppData\Roaming\Reaction-Systems\data`

The native application also has the ability to save and load the state from a file. The files have by default the extension "`.ron`". The web version has no ability to interact with the file system due to a limitation of `webassembly`.

The file structure can be seen in figure 3.2, where "state" refers to the state of the GUI, "translator" refers to the `Translator` structure used to encode entities names into fixed sized integers, and "cache" refers to the cache structure for the GUI. Version number is a little-endian `u64` that encodes the version number of the application; if different from the version of the application, a warning will be issued, but the application will try and load the state anyway. Each "length" field is a little-endian `u64` and indicates the length in bytes of the corresponding field.

The user can request the result of a computation by interacting with the button "*Set active*" under most of the windows. A panel on the right of the screen appears with the computed result. The nodes "Save string to file" and "Save SVG" instead have a button "*Write*" that writes to file the result. The node "Read a file" has an extra button "*Update file*" that reads again the file from disk since a filewatcher has not been implemented.

Since the generated graphs were often times immediately converted to DOT files and rendered to SVG, a native renderer is included that can create PNG images of the supplied graph. This reduces greatly the time switching between software to achieve the same result.

# Chapter 4

# Development

## 4.1 ReactionSystems

### 4.1.1 Entities and Translator

Entities are declared in the file element.rs and the `Translator` struct is implemented in the file translator.rs.

Entities have type `IdType` and are represented as `u32`. Representing arbitrarily named entities with integers has the immediate benefit of faster code execution, but need additional support for the encoding and decoding. Also it does not permit easy merging of different systems. This is because two elements with the same string might be assigned to a different integer and would need to be re-encoded. The ReactionSystemsGUI solves this problem by having only one `Translator` class for all entities and systems.

Positive RS have the property that if all the entities are declared in the initial state, in all subsequent states the entities will all be defined either positive or negative. This property can be exploited in the representation of a Positive RS, however the implementation disregards this fact and simply assigns either positive or negative to each positive entity.

The struct `Translator` is formed by two maps, one from strings to `IdType` and the inverse, and by a counter for the last used id. It is essential for this class to be serializable, so that the state of ReactionSystemsGUI might save it when necessary. The struct is also used to form the structure `Formatter`, which is used to format all structures that implement `PrintableWithTranslator`.

For example the implementation of `PrintableWithTranslator` for `Set` is the following:

```
1 impl PrintableWithTranslator for Set {
2     fn print(
3         &self,
4         f: &mut fmt::Formatter,
5         translator: &Translator,
6     ) -> fmt::Result {
```

```
7           write!(f, "{{")?;
8           let mut it = self.iter().peekable();
9           while let Some(el) = it.next() {
10              if it.peek().is_none() {
11                  write!(f, "{}", Formatter::from(translator, el))?;
12              } else {
13                  write!(f, "{}, ", Formatter::from(translator, el))?;
14              }
15          }
16          write!(f, "}}")
17      }
18  }
```

The structure `Translator` is only borrowed because it is never modified when printing, so only one is needed for all of the print. On lines 11 and 13 instead of directly printing `el`, we first construct another `Formatter` struct and require only for that struct to implement `std::fmt::Display`. This gives modularity and flexibility to the display system.

### 4.1.2 Set

The structure `set` is a key component for all functions in the library. It is implemented as a binary tree set. Binary trees were chosen instead of hash sets for various reasons: binary trees support hashing of the whole tree, hash sets do not; the penalty for retrieval of individual elements is offset by the performance gain for set operations like union or intersection.

### 4.1.3 Tests

During the development of the library some tests were developed in order to test behavior in the changing code. They can be run with `cargo test`. Test coverage is not high, but is present in for pieces of code that might break more easily. Tests are usually present in a separate file as the structure declaration and are named "*_test.rs", so that they might be easily recognized.

In addition to automated tests, some example inputs are provided in the folder `testing`. The extension `.system` symbolizes system and associated instructions; the extension `.experiment` symbolizes an experiment, see 3.1.13.

## 4.2 ReactionSystemsGUI

## 4.3 Validation

# Chapter 5

# Conclusion

# Chapter 6

# Appendix