Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

Runtime Verification for Biochemical Programs

Oana Andrei[1]

School of Computing Science, University of Glasgow, Glasgow, UK

Hélène Kirchner[2]

INRIA, France

Abstract

The biochemical paradigm is well-suited for modelling autonomous systems and new programming languages are emerging from this approach. However, in order to validate such programs, we need to define precisely their semantics and to provide verification techniques. In this paper, we consider a higher-order biochemical calculus that models the structure of system states and its dynamics thanks to rewriting abstractions, namely rules and strategies. We extend this calculus with a runtime verification technique in order to perform automatic discovery of property satisfaction failure. The property specification language is a subclass of LTL safety and liveness properties.

Keywords: Biochemical calculus, Rewriting strategies, Autonomous systems, Runtime verification.

1 Introduction

More and more complex distributed computing systems are employed in large computer networks where direct human intervention would become easily overwhelmed. Then self-managing properties for such systems became highly desirable and the autonomic computing framework [14] was designed for this purpose. Autonomous systems are initially provided with some high-level instructions from administrators and should require minimal human intervention during their functioning if none. There is crucial need for theories and formal frameworks to model computation, to define unconventional languages for programming and to establish foundations for verifying properties of these systems, for instance security-related properties.

Our work fits into the unconventional programming languages approach such as the *chemical paradigm* [11,13,7]. The chemical paradigm captures the intuition of a *collection of freely interacting atomic values* where interactions are modelled

[1] Email: Oana.Andrei@glasgow.ac.uk
[2] Email: Helene.Kirchner@inria.fr

©2011 Published by Elsevier Science B. V.

by rules and has been applied to modelling self-managing systems [7], computer networks, grid infrastructures [8], web services [16]. We extended this paradigm with structured objects instead of atomic values and defined a higher-order formalism, the *Abstract Biochemical Calculus* [5]. An autonomous system is modelled as a *biochemical program* with the structural part described by a collection of objects and behavioural part as a collection of transformations (rewrite rules) over collections of objects, both at the same level. Objects and rewrite rules are generically called *molecules* and collections of molecules are also molecules. Higher-order rewrite rules over molecules introduce new rewrite rules in the behaviour of the system. More expressive power is gained in modelling a system's evolution via *strategies* which control the application order of a set of rewrite rules.

In [3] we proposed the visual structure of *port graph*, as well as port graph rewrite rules and a rewriting relation on port graphs for modelling complex systems with dynamical topology, whose components interact in a concurrent and distributed manner. Nodes with ports represent components (or objects), while edges correspond to interactions (or communications). Then port graph rewrite rules are used for modelling the system evolution by creating new nodes and connections among them at specific points (ports), breaking connections, merging, splitting, or deleting nodes, or performing any combination of these operations. By instantiating the structure of the objects in the Abstract Biochemical Calculus with port graphs, we obtained the *Port Graph Calculus* and we showed its capabilities of modelling autonomous systems and biochemical networks in [1,4]. An implementation of the port graph calculus is provided by the PORGY environment [2].

At this stage, our aim is to improve the confidence in biochemical models of autonomous systems by ensuring that the current execution is consistent with the expected properties of the system and by detecting violations of safety properties in order to recover from problematic situations. Runtime verification fits well in this context as a verification technique that increases confidence in the correctness of the system behaviour with respect to its formal specification. We have already proposed a method for verifying invariant properties of biochemical program [5]. An invariant property is encoded as a special rule in the biochemical program modelling the system and such rule is dynamically checked at each execution step. We obtained a kind of runtime verification technique which allows the running program to detect its own structural failures.

In this paper, we enrich the abstract biochemical calculus with a verification technique where properties are expressed in linear temporal logic (LTL) [17]. We concentrate here on *safety properties* ("Something bad never happens") that include invariant properties as subclass, and *liveness properties* ("Something good will happen eventually"). We consider a set of LTL formulae built using one temporal modal operator, which is enough to express a fair amount of properties, and we use a three-valued semantics for the LTL formulae [10]. In order to illustrate our approach and the proposed concepts, in this paper we develop a simple example of service orchestration based on library loaning services.

The paper is organised as follows. In the next section, we give an overview of the abstract biochemical calculus, syntax and semantics. Section 3 discusses the Kripke structure associated to a biochemical program and the set of temporal formulae we

consider. In Section 4, we formalise runtime verification in this context and extend the calculus to perform simultaneously computation and verification steps. Further perspectives are drawn in Section 5.

2 The Abstract Biochemical Calculus

The Abstract Biochemical Calculus [5] is based on two main formalisms: the ρ-calculus (also called the rewriting calculus) [12] and the γ-calculus [6]. It extends the chemical model of the γ-calculus by embedding higher-order capabilities of the rewriting calculus and by considering an abstract structure $\mathcal{OBJ}$ for the basic chemical objects of the γ-calculus. The structure $\mathcal{OBJ}$ is required to allow modelling connections between objects as well as creating and removing such connections. The result is an abstract biochemical calculus based on rewriting structured molecules, called the $\rho_{\langle\mathcal{OBJ}\rangle}$-calculus and presented in the rest of this section.

2.1 Syntax

The syntax of the Abstract Biochemical Calculus is given in Fig. 1. In the following we detail each class of entities.

The structure of objects $\mathcal{OBJ}$ is an abstract one; for instance objects may be terms, records, graphs, *etc.* We only require that for any object $O \in \mathcal{OBJ}$, we can retrieve the set of its variables denoted by $\mathcal{V}ar(O)$, we can select a sub-object, and that there always exists a procedure for matching objects w.r.t. a given congruence. We assume that a matching algorithm $\mathcal{S}ol$ for $\mathcal{OBJ}$ as well as a structural congruence relation $\equiv$ on objects are given as parameters of the calculus. Following the intuition of chemical solutions, *collections of objects* are built using an associative and commutative juxtaposition operator $\bullet$. We also define an object stk to be used later as failure object.

The collection of objects can be transformed using *rules* of the form $L \Rightarrow R$ where L and R are objects such that $\mathcal{V}ar(R) \subseteq \mathcal{V}ar(L)$. Objects and rules are grouped together in *molecules* which are bags built using the same associative-commutative (AC) juxtaposition operator $\bullet$ as for objects. The structural congruence relation on molecules is defined as the union of structural congruence relation on $\mathcal{OBJ}$ and the AC properties of the juxtaposition operator on objects and molecules.

Transformations of molecules are defined as *higher-order abstractions* $M \Rrightarrow K$. A rule is a first-order abstraction that abstracts only on (collections of) objects while the general higher-order abstractions act on objects as well as on rules. Hereafter,

(Objects)	$\mathcal{O}$	$::=$	$\mathsf{stk} \mid \mathcal{OBJ} \mid \mathcal{X} \mid \mathcal{O}\bullet\mathcal{O}$
(Molecules)	$\mathcal{M}$	$::=$	$\mathcal{O} \mid \mathcal{O} \Rightarrow \mathcal{O} \mid \mathcal{M}\bullet\mathcal{M} \mid \mathcal{Y}$
(Configurations)	$\mathcal{K}$	$::=$	$\mathcal{M} \mid \mathcal{M} \Rrightarrow \mathcal{K} \mid (\mathcal{M} \Rrightarrow \mathcal{K})@\mathcal{K} \mid \mathcal{K}\bullet\mathcal{K}$
(Biochemical program)	$\mathcal{P}$	$::=$	$[\mathcal{K}]$

Fig. 1. The syntax of the Abstract Biochemical Calculus

we use the symbol $\Rrightarrow$ to denote either $\Rightarrow$ or $\Rrightarrow$. The application operator @ takes an abstraction and a molecule to build a *reactive molecule*. Higher-order abstractions, also called *strategies*, may contain in their right-hand side reactive molecules.

Collections of molecules and abstractions are called *configurations*. Then all molecules and abstractions describing the structure and behaviour of the system to be modelled are put in a configuration to obtain a *biochemical program*, which we also refer to as (system) state. We introduce the program entity in order to distinguish between global and local configurations by making explicit the square brackets for the former ones. This distinction is needed in the formalisation of the calculus.

Example 2.1 We illustrate the calculus with a toy example of a loaning service model provided by a university library for students in the Departments of Mathematics and Computer Science. The entities of this model are students as *users*, libraries as *service providers*, textbooks as *resources* (or services), and a global clock:

- **student** $S(id, dpt, state)$, where id is a unique student identity, dpt takes a value of M or CS, and $state$ takes a value of 1 for requesting, 2 for pending, 3 for reading;
- **library** $L(dpt, SID, BID)$, where SID and BID are sets of student and book identities respectively;
- **book** $B(b, sid, tick)$, where b is the book identity, and sid takes the value of the identifier of the student who borrowed it at non-nullary time stamp $tick$, otherwise nil if $tick$ is 0.

We model each entity as a node with ports in a port graph [4], where some ports have states or maximum arity greater than one. A library node has the port SID where several students from the same department may connect and the port BID where all book nodes from the same department are connected, a student node can be connected with a book node, and the two libraries are connected. Let $\equiv$ denote the isomorphism relation on port graphs. The juxtaposition operator $\bullet$ is not represented for port graph objects.

The library service example has three basic actions illustrated as port graph rewrite rules in Fig. 2:

BReq: a student *requests* a book at the departmental library within one time unit;

BBrw: a student *borrows* an available book from the library within one time unit;

BRet: a student returns a book instantaneously to the departmental library after holding it for at most 5 time units;

DBRet: a student returns a book instantaneously to the library of another department after holding it for at most 3 time units.

2.2 Semantics of Biochemical Programs

The evaluation mechanism of the calculus relies on solving the fundamental problem of *matching* a pattern to a molecule, i.e., finding an injective morphism between the pattern and the molecule. Let $\mathcal{S}ol$ be a function that returns the set of substitutions

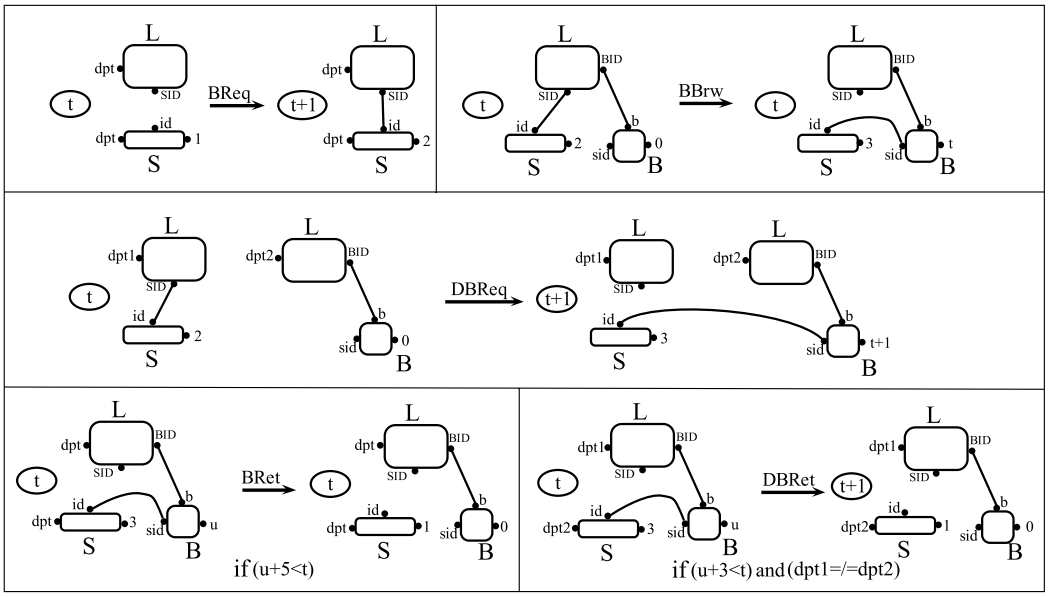

Fig. 2. Rewrite rules describing the actions in the library loaning service

σ solutions of a given matching problem between two molecules M and M', denoted $M \ll M'$. A submatching problem $M \prec\!\!\!\!\prec M'$ is a matching problem from M to a sub-molecule M'' of M'.

Definition 2.2 The *structural congruence on configurations* is the least equivalence relation on configurations such that it includes the structural congruence relation on molecules and the following equations:

$$\textbf{(Application)}\ (M \Rrightarrow K)@M' \equiv \varsigma(K)\ \textbf{if}\ \varsigma \in \mathcal{S}ol(M \prec\!\!\!\!\prec M')$$

$$\textbf{(Application Fail)}\ (M \Rrightarrow K)@M' \equiv \mathsf{stk}\ \textbf{if}\ \mathcal{S}ol(M \prec\!\!\!\!\prec M') = \emptyset$$

$$\textbf{(Stuck)}\ \mathsf{stk} \bullet K \equiv K$$

If the matching problem between the left-hand side of the abstraction $M \Rrightarrow K$ and the molecule M' has solutions, the equation **(Application)** is used and one substitution is chosen from the solution set and applied on the right-hand side of $M \Rrightarrow K$; otherwise, if the matching problem has no solution, the application fails and it is reduced to the failure object stk according to equation **(Application Fail)**. A stk object is deleted from a juxtaposition of configurations according to **(Stuck)**.

Definition 2.3 A configuration is in *normal form* if it is irreducible with respect to the equations **(Application)**, **(Application Fail)** and **(Stuck)** ordered from left to right. A program $[K]$ is in *normal form* if configuration K is in normal form.

The following evaluation rule defines the operational semantics of biochemical programs, more specifically, the *interaction* between abstractions and molecules:

$$\textbf{(Interaction)}\ [K' \bullet (M \Rrightarrow K) \bullet M'] \longrightarrow [K' \bullet \varsigma(K)]$$
$$\textbf{if}\ \varsigma \in \mathcal{S}ol(M \prec\!\!\!\!\prec M')\ \textit{and}\ [K' \bullet (M \Rrightarrow K) \bullet M']\ \textit{is in normal form}$$

The interaction evaluation rule chooses an abstraction $M \Rrightarrow K$, a molecule M' in the current program and a substitution ς solution of the matching problem $M \prec\!\!\!\!\prec M'$, and then replaces $M \Rrightarrow K$ and M' with the configuration obtained from applying the substitution ς to K. The variable K' captures the context in which the rule $M \Rrightarrow K$ and the molecule M' interact, thanks to the AC properties of $\bullet$.

We define a new evaluation rule on biochemical programs $[K] \longrightarrow_{\equiv} [K']$ as syntactic sugar for $[K] \longrightarrow [K'']$ and $[K''] \equiv [K']$ with $[K']$ in normal form.

2.3 Strategies

It is important to emphasise that we can model a non-deterministic (and possibly non-terminating) behaviour for the application of abstractions in a biochemical program, or introduce some control to compose or to choose the rules to apply. For this purpose we introduce *strategies* in the calculus by expressing them as higher-order abstractions. Strategies may be used in various ways: to sequentialise or order the application of several abstractions, to exploit failure information, to define persistent rules that are not consumed when applied, to implement case analysis and iteration. More details can be found for instance in [1,5].

Most strategies are encoded as abstractions of the form $Y \Rrightarrow K$, where Y is a molecule variable and it corresponds to a higher-order function, i.e. to a lambda expression $\lambda Y.K$. When $Y \Rrightarrow K$ is applied to a molecule M, we obtain a reactive molecule $(Y \Rrightarrow K)@M$ which is structurally congruent to K where each occurrence of Y has been replaced by M.

In Fig. 3 we recall the encoding of some strategies as higher-order abstractions in the $\rho_{\langle \mathcal{OBJ} \rangle}$-calculus, where T, T_1, T_2, T_3 range over abstractions. The strategies id and $\mathtt{fail}$ represent the identity and failure strategies. The $\mathtt{seq}(T_1, T_2)$ strategy applies sequentially the two abstractions T_1 and T_2, while $\mathtt{first}(T_1, T_2)$ tries first to apply T_1 and in case of application failure it applies the second abstraction T_2. The application of $\mathtt{not}(T)$ fails if the application of T_1 does not fail, otherwise it does nothing. The strategy $\mathtt{ifThenElse}(T_1, T_2, T_3)$ applies the first strategy: if it does not fail, it applies the second strategy, else it applies the third strategy; it fails if both applications of T_2 and T_3 fail.

$$\mathsf{id} \triangleq Y \Rrightarrow T \qquad \mathtt{fail} \triangleq Y \Rrightarrow \mathsf{stk} \qquad \mathtt{seq}(T_1, T_2) \triangleq Y \Rrightarrow T_2@(T_1@Y)$$

$$\mathtt{first}(T_1, T_2) \triangleq Y \Rrightarrow (T_1@Y)\bullet(\mathsf{stk} \Rrightarrow (T_2@Y))@(T_1@Y)$$

$$\mathtt{not}(T) \triangleq Y \Rrightarrow \mathtt{first}(\mathsf{stk} \Rrightarrow Y, Y' \Rrightarrow \mathsf{stk})@(T@Y)$$

$$\mathtt{ifThenElse}(T_1, T_2, T_3) \triangleq Y \Rrightarrow \mathtt{first}(\mathsf{stk} \Rrightarrow T_3@Y, Y' \Rrightarrow T_2@Y)@(T_1@Y)$$

Fig. 3. Strategy operators encoded as higher-order abstractions

A strategy T can be applied as long as it is does not fail using the recursive strategy: $\mathtt{repeat}(T) \triangleq \mu X.\mathtt{first}(\mathtt{seq}(T, X), \mathsf{id})$. The recursion operator μ is encoded using the fixed-point combinator of the λ-calculus following the procedure done for encoding iterators in the ρ-calculus [12].

In the **(Interaction)** evaluation rule, a strategy (or a rule) $M \Rrightarrow K$ is consumed by a non-failing interaction with a molecule M'. The *persistent* strategy combina-

tor defined by $T! \triangleq \mu X.\mathtt{seq}(T, \mathtt{first}(\mathsf{stk} \Rightarrow \mathsf{stk}, Y \Rrightarrow Y \bullet X))$, when applied to a molecule, applies T to the molecule and $T!$ is replicated; if the application of T fails, then the failure stk is returned.

Example 2.4 We extend the library service model with an action **DBReq** depicted in Fig. 2 which allows a library to dispatch a book request to the other library if the latter has available books resulting in a student borrowing a book from the other department's library. A library can dispatch a request only when it has no book available by considering a strategy which gives a higher priority to **BBrw** than **DBReq**; therefore we replace the rules **BBrw** and **DBReq** by $\mathtt{seq}(\mathbf{BBrw}, \mathbf{DBReq})$.

The initial state of the biochemical program modelling the library service is a juxtaposition of student nodes in requesting state, two connected library nodes – one for each department, book nodes available for each library, and the persistent strategies **BReq**!, $\mathtt{seq}(\mathbf{BBrw}, \mathbf{DBReq})!$, **BRet1**! and **BRet2**!.

Rewrite rules and strategies used in biochemical programs are indeed essential concepts to express evolution of autonomous systems. The same concepts can be reused for developing verification techniques for autonomous systems.

A first approach for embedding verification in models of autonomous systems, described in [5], consists in expressing an invariant property of the system as an abstraction with identical sides, $M \Rrightarrow M$, and relying on the matching process for testing the presence of a molecule M. The failure of the invariant is handled by a failure port graph Error that does not allow the execution to continue. The strategy verifying such an invariant is $\mathtt{first}(M \Rrightarrow M, X \Rightarrow \mathsf{Error})!$. We can also express the unwanted occurrence of a molecule M using the strategy $(M \Rightarrow \mathsf{Error})!$. In both cases above, instead of yielding the failure Error signalling that a property of the system is not satisfied, the problem can be "repaired" by associating to each property the necessary rules or strategies to be inserted in the system in case of failure. Such ideas open a wide field of possibilities for combining runtime verification and self-healing capabilities and we explore them further in Sect. 4.

Building up on this simple idea, our goal is to generalise invariant verification towards the proof of safety and liveness properties. This is natural since the evaluation rule of biochemical programs can be seen as a reduction relation in a state transition system – more specifically an Abstract Reduction System (ARS). For verification, we consider properties expressed as formulae in a three-valued linear temporal logic and interpreted over traces in the ARS associated to the program. First we show how to encode the properties into adequate rewrite strategies, which allows us to embed them in each state of an extended ARS. For that purpose, the structure of programs is enriched by guards and another reduction relation is defined on guarded programs. Then property satisfaction is checked on each transition state via the evaluation mechanism of the rewrite strategies.

3 Linear Temporal Logic for Biochemical Reductions

In this section, we first review the basic concepts underlying an Abstract Reduction System. Then we define a set of structural formulae for reasoning about the

molecular structures in configurations. Finally, we review the set of LTL formulae [17] defined over the Abstract Reduction System whose states are biochemical programs and whose transitions are determined by the interaction evaluation rule in the $\rho_{\langle\mathcal{OBJ}\rangle}$-calculus.

3.1 Abstract Reduction Systems

An *Abstract Reduction System (ARS)* [15] is a labelled oriented graph (S, s_0, R) with S a finite set of nodes called *states*, $s_0 \in S$ the initial state, and R a binary relation over $S \times S$ called *transition relation*. Let $\mathcal{A}$ be the ARS built over biochemical programs using the interaction evaluation relation. Then $(s, s') \in R$ if $s \longrightarrow_{\equiv} s'$ in the $\rho_{\langle\mathcal{OBJ}\rangle}$-calculus. We call $s \longrightarrow_{\equiv} s'$ a *reduction steps* or *transition*. If we consider a function labelling states with atomic propositions, we obtain the definition of a Kripke structure.

A computation *path* or *trace* in the ARS $\mathcal{A}$ is a path in the graph starting from the initial state s_0, i.e., an infinite non-empty sequence $s_0 s_1 s_2 \ldots$ of states in S where $s_i \longrightarrow_{\equiv} s_{i+1}$ for all $i \geq 0$. Let $Path(\mathcal{A})$ denote the set of paths of $\mathcal{A}$. A finite trace is a non-empty sequence $s_0 s_1 \ldots s_n$ with $n \geq 0$. A finite trace $u = s_0 s_1 \ldots s_n$ can be concatenated with an infinite sequence of states $w = s'_0 s'_1 \ldots$ with $s'_j \longrightarrow_{\equiv} s'_{j+1}$, for all $j \geq 0$, to obtain a new infinite trace if and only if $s_n \longrightarrow_{\equiv} s'_0$ and we say that w is a *continuation* of u. For a path $w = s_0 s_1 s_2 \ldots$, we denote by $w[i]$ the ith state of the path, namely s_{i-1}, and by w_i the suffix of the path w starting at state $w[i]$, i.e., $w_i = s_{i-1} s_i \ldots$. We say that a state s in the ARS $\mathcal{A}$ is *final* (or irreducible) if there is no state s' such that $s \longrightarrow_{\equiv} s'$. We denote a final state s by $s_{\downarrow}$.

3.2 Structural Formulae and Structural Satisfaction

Based on molecule definition and Boolean connectors, in the following we define the formulae characterising the molecular structure of states in $\mathcal{A}$.

Definition 3.1 The set of *structural formulae* $\mathcal{F}(\mathcal{M})$ is constructed inductively as follows:

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \diamond\!\!\!

The structural satisfaction problem between a molecule M and a formula M' reduces to successfully solving the matching problem $M' \ll M$. A molecule M structurally satisfies a formula $\diamond\!\!\!\!\diamond\varphi$ if M has a submolecule M' satisfying φ.

The following proposition states that the structural satisfaction is up to the structural congruence relation on states. The proof is immediate using induction over the structure of φ.

Proposition 3.3 *If $s \models \varphi$ and $s \equiv s'$ then $s' \models \varphi$.*

3.3 Temporal Formulae

In order to express safety or liveness properties, we need a temporal logic. The set of *LTL formulae* over $\mathcal{F}(\mathcal{M})$ is inductively defined by:

$$\phi ::= \mathit{true} \mid \mathit{false} \mid \varphi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi$$

where φ ranges over the set of structural formulae $\mathcal{F}(\mathcal{M})$. Three more path operators are available as syntactic sugar: the *eventually* operator F (future) with $\mathsf{F}\phi \equiv \mathit{true}\,\mathsf{U}\,\phi$, the *always* operator G (globally) with $\mathsf{G}\phi \equiv \neg(\mathsf{F}\neg\phi)$, and the *release* operator R with $\phi_1\mathsf{R}\phi_2 = \neg(\neg\phi_1\mathsf{U}\neg\phi_2)$.

The LTL formulae are interpreted over traces in the ARS $\mathcal{A}$ and their semantics is defined inductively as follows:

$$\begin{array}{lll}
w \models \mathit{true} \quad\quad w \not\models \mathit{false} & & w \models \phi_1 \vee \phi_2 \;\; \mathit{iff} \;\; w \models \phi_1 \text{ or } w \models \phi_2 \\
w \models \varphi \quad \mathit{iff} \;\; w[0] \models \varphi & & w \models \phi_1 \wedge \phi_2 \;\; \mathit{iff} \;\; w \models \phi_1 \text{ and } w \models \phi_2 \\
w \models \neg\phi \;\; \mathit{iff} \;\; w \not\models \phi & & w \models \phi_1 \rightarrow \phi_2 \;\; \mathit{iff} \;\; w \not\models \phi_1 \text{ or } w \models \phi_2 \\
w \models \mathsf{X}\phi \;\; \mathit{iff} \;\; w_1 \models \phi & & \\
w \models \mathsf{F}\phi \;\; \mathit{iff} \;\; \exists i \geq 0 \,.\, w_i \models \phi & & w \models \mathsf{G}\phi \;\; \mathit{iff} \;\; \forall i \geq 0 \,.\, w_i \models \phi \\
w \models \phi_1\mathsf{U}\phi_2 \;\; \mathit{iff} \;\; \exists k \geq i \text{ s.t. } w_k \models \phi_2 \text{ and } \forall i \leq l < k \,.\, w_l \models \phi_1 & & \\
w \models \phi_1\mathsf{R}\phi_2 \;\; \mathit{iff} \;\; \forall k \geq 0 \text{ either } w_k \models \phi_2 \text{ or } \exists j \geq 0 \text{ s.t. } w_i \models \phi_1 \text{ and } \forall j \leq i.\; w_j \models \phi_2 & &
\end{array}$$

An LTL formula ϕ holds in a state s of $\mathcal{A}$, $\langle \mathcal{A}, s\rangle \models \phi$, if and only if for every infinite trace w with $w[0] = s$ in $\mathcal{A}$ we have $w \models \phi$. The LTL model checking problem $\mathcal{A} \models \phi$ checks if $\langle \mathcal{A}, s_0\rangle \models \phi$ where s_0 is the initial state of $\mathcal{A}$.

We consider only LTL formulae in positive normal form, i.e. negations only occur in front of structural formulae. This is possible since every operator in the LTL syntax presented above has a dual. In particular we have the following equivalences: $\neg(\mathsf{X}\phi) = \mathsf{X}(\neg\phi)$, $\neg(\mathsf{F}\phi) = \mathsf{G}(\neg\phi)$, $\neg(\mathsf{G}\phi) = \mathsf{F}(\neg\phi)$, $\neg(\phi_1\,\mathsf{U}\,\phi_2) = \neg\phi_1\,\mathsf{R}\neg\phi_2$.

4 Embedding Runtime Verification in the Abstract Biochemical Calculus

In this section we extend the reduction relation defined for the biochemical calculus with LTL formulae as guards of the computation. The aim is to check at runtime

that, after each interaction step induced by the interaction evaluation rule, a set of properties expressed as simple LTL formulae is satisfied.

In runtime verification a trace is also called *run* and a finite trace an *execution*. Monitoring a property at runtime can only be formally defined over finite traces. The LTL_3 [9], a variant of LTL with a three-valued semantics, was designed to overcome this gap between monitoring finite and infinite traces. In an automaton approach, a finite trace is a *good prefix* if any infinite continuation of the trace will always be accepted, a *bad prefix* if there is no continuation of the trace to build an accepting trace, and an *ugly prefix* otherwise. A good, bad or ugly prefix of an LTL formula is evaluated to *true*, *false* or inconclusive (denoted by ?) respectively. The semantics of a LTL_3 formula ϕ with respect to a finite trace u in an abstract reduction system $\mathcal{A}$ is defined as follows:

$$[u \models \phi] = \begin{cases} \mathit{true} & \text{if } \forall\, w \text{ continuation for } u,\ uw \models \phi \\ \mathit{false} & \text{if } \forall\, w \text{ continuation for } u,\ uw \not\models \phi \\ ? & \text{otherwise} \end{cases}$$

We adopt the following approach. We start with an inconclusive guard and we check that a property ϕ is satisfied by the current execution by applying a strategy encoding the formula. If no decision can be made, either to state that the property is satisfied or not, we pass on the inconclusive guard meaning that all along the current execution there is no state that makes the property not satisfied.

4.1 Syntax

We consider the following set of LTL formulae defined over the set of structural formulae $\mathcal{F}(\mathcal{M})$:

$$\phi ::= \mathit{true} \mid \mathit{false} \mid \varphi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi$$

We define *guarded biochemical programs* with the guard being an LTL formula ϕ over molecules, prefixed by a satisfaction operator (either $\models$, $\not\models$ or $\models^?$), or a conjunction of such formulae:

$$\begin{array}{rrcl} \textbf{(Satisfaction operators)} & \alpha & ::= & \models \mid \not\models \mid \models^? \\ \textbf{(Guards)} & \mathcal{H} & ::= & \alpha\ \phi \mid \mathcal{H} \wedge \mathcal{H} \\ \textbf{(Guarded biochemical programs)} & \mathcal{GS} & ::= & [\mathcal{K}]_{\mathcal{H}} \mid (\mathcal{M} \Rrightarrow \mathcal{K})@[\mathcal{K}]_{\mathcal{H}} \end{array}$$

The atomic guard $\models \phi$ ($\not\models \phi$) states that the formula ϕ is satisfied (not satisfied respectively) by the current execution, whereas $\models^? \phi$ corresponds to a consistent inconclusive satisfaction answer all along the current execution. We consider guards $\alpha\ \phi$ with ϕ in positive conjunctive normal form and we decompose guards such that the LTL formula ϕ is not a conjunction. The structural congruence relation on atomic guards, denoted also $\equiv$, is defined inductively as follows:

$$\alpha\ \phi_1 \equiv \alpha\ \phi_2 \text{ if } \phi_1 \text{ and } \phi_2 \text{ are syntactically equivalent}$$
$$\alpha\ (\phi_1 \wedge \ldots \wedge \phi_n) \equiv (\alpha\ \phi_1) \wedge \ldots \wedge (\alpha\ \phi_n)$$
$$(\not\models \phi) \wedge (\alpha\ \phi_1) \wedge \ldots \wedge (\alpha\ \phi_n) \equiv \not\models (\phi \wedge \phi_1 \wedge \ldots \wedge \phi_n)$$

We introduce an intermediate form of guarded programs, $(\mathcal{M} \Rrightarrow \mathcal{K})@[\mathcal{K}]_{\mathcal{H}}$, consisting of the application of an abstraction on a guarded program of the form $[K]_H$. We need it to define the new extended reduction relation for guarded program.

Let $\mathcal{Z}$ and $\mathcal{W}$ denote respectively sets of variables for configurations and for guarded programs. We define new *abstractions over guarded programs* of the form $W \Rrightarrow GS$ and $[Z]_H \Rrightarrow GS$ where $W \in \mathcal{W}$, $Z \in \mathcal{Z}$ and GS a guarded program.

Definition 4.1 The *structural congruence relation* on guarded biochemical programs is the least equivalence relation on guarded biochemical programs generated by the equation:

$$[K]_H \equiv [K']_{H'} \text{ if and only if } K \equiv K' \text{ and } H \equiv H'$$

and by the following equations oriented from left to right:

$$(M \Rrightarrow K)@[K']_H \equiv [(M \Rrightarrow K)@K']_H$$
$$(W \Rrightarrow [K]_H)@[K']_{H'} \equiv [K]_H$$
$$([Z]_H \Rrightarrow [K]_H)@[K']_H \equiv [\{Z \mapsto K'\}K]_H$$

where $\{Z \mapsto K'\}K$ denotes the replacement of all occurrences of the variable Z in K by K'.

In the following we explain each of the three application reductions included in the definition of the structural congruence relation above. First, the application of an abstraction $M \Rrightarrow K$ on a guarded program $[K']_H$ is equivalent to the application of the abstraction to the top-level configuration of the guarded program since the abstraction $M \Rrightarrow K$ as no effect on the guard H. Second, the application of an abstraction with a guarded program variable as left-hand side on any guarded program returns its right-hand side: the matching substitution associates W to $[K']_{H'}$ and the application result is the right-hand side $[K]_H$ (the matching substitution does not modify it). Third, the application of an abstraction $[Z]_H \Rrightarrow [K]_H$ on a guarded program $[K']_{H'}$ produces a matching substitution associating variable Z to K' and returns the right-hand side $[K]_H$ of the abstraction where every occurrence of the variable Z is replaced by K.

Definition 4.2 A guarded biochemical program is in *normal form* if it has the form $[K]_H$ where K is in normal form and H is a conjunction of atomic guards whose underlying LTL formulae are in positive conjunctive normal form.

Example 4.3 An example of safety formula in the library service model is the property that a student should always be able to borrow a book from any departmental library $\mathsf{G}(\Diamond S(id, dpt, 2) \wedge \Diamond B(b, nil, 0)))$ with formula $S(id, dpt, 2)$ standing for a student connected to a library in requesting mode and formula $B(b, nil, 0)$ standing for an available book. We can formulate other formulae such as: $\neg\mathsf{G}(\Diamond B(b, SID, i) \wedge \Diamond B(b, SID, j))$ stating that a book cannot be simultane-

ously borrowed by two students or the liveness property $\mathsf{F}(\Diamond B(id, nil, 0)$ for a book to be available at some point.

4.2 Semantics of Guarded Biochemical Programs

A structural formula φ in $\mathcal{F}(\mathcal{M})$ is mapped to a strategy $\tau(\varphi)$ and it is interpreted over a molecule M from a program configuration as the result of the strategy application $\tau(\varphi)@M$. We define the mapping τ from structural formulae to strategies inductively as follows:

$$\tau(\mathit{true}) \triangleq \mathsf{id} \qquad \tau(\mathit{false}) \triangleq \mathtt{fail}$$

$$\tau(\Diamond M) \triangleq M \Rightarrow M \qquad \tau(\neg\varphi) \triangleq \mathtt{not}(\tau(\varphi))$$

$$\tau(\varphi_1 \wedge \varphi_2) \triangleq \mathtt{seq}(\tau(\varphi_1), \tau(\varphi_2)) \qquad \tau(\varphi_1 \vee \varphi_2) \triangleq \mathtt{first}(\tau(\varphi_1), \tau(\varphi_2))$$

$$\tau(\varphi_1 \rightarrow \varphi_2) \triangleq Y \Rrightarrow \mathtt{seq}(\tau(\varphi_1), \mathtt{first}(\mathsf{stk} \Rrightarrow Y, \tau(\varphi_2)))@Y$$

Lemma 4.4 *Let φ be a structural formula in $\mathcal{F}(\mathcal{M})$ and M a molecule. Then the normal form of $\tau(\varphi)@M$ is either M or* stk.

Proposition 4.5 *If M is a molecule and φ a structural formula in the calculus, then (i) $M \models \varphi$ if and only if $\tau(\varphi)@M \equiv M$, and (ii) $M \not\models \varphi$ if and only if $\tau(\varphi)@M \equiv \mathsf{stk}$.*

Intuitively, if the application of the strategy encoding φ on a molecule M fails, then the formula is not satisfied by M. The above proposition shows the soundness and completeness of encoding the semantics of structural formulae via the evaluation mechanism of strategies. The proofs of these last two results can be found in [1].

For s_0 the initial state of the ARS $\mathcal{A}$ and ϕ a temporal property that we want to verify along the execution of the system, we start with the *initial guarded biochemical program* $s_0 \models^? \phi$. In the following we define a new reduction relation on guarded programs denoted $\Longrightarrow$ which encompasses the reduction relation $\longrightarrow_{\equiv}$ on biochemical programs and verifies at each step the satisfaction of the guard.

The process of verifying the satisfaction of a guard along an execution trace *consumes* it: an LTL formula and a state s generate another LTL formula. To define the consumption of an LTL formula ϕ by the reduction relation in the calculus, we use *derivatives* [18] for LTL. The evaluation of derivations provide a way of determining if a prefix is good or bad for a formula. For s a state and ϕ an LTL formula, the derivative of ϕ in state s, denoted by $\phi\{s\}$, has the property that for any finite trace u, $su \models \phi$ iff $u \models \phi\{s\}$ and $su \not\models \phi$ iff $u \not\models \phi\{s\}$.

Theorem 4.6 (Derivative [18]) *For any LTL formula ϕ and for any finite trace $u = s_0 s_1 \ldots s_n$, u is a good (bad) prefix for ϕ if and only if $\phi\{s_0\}\{s_1\}\ldots\{s_n\}$ evaluates to true (false).*

The recursive definition of the operator $_\{_\}$ is:

$$\mathit{false}\{s\} = \mathit{false} \qquad \mathit{true}\{s\} = \mathit{true}$$

$$(\neg\phi)\{s\} = \neg(\phi\{s\}) \qquad (\phi_1 \vee \phi_2)\{s\} = \phi_1\{s\} \vee \phi_2\{s\}$$

$$(\phi_1 \wedge \phi_2)\{s\} = \phi_1\{s\} \wedge \phi_2\{s\} \qquad (\phi_1 \rightarrow \phi_2)\{s\} = \phi_1\{s\} \rightarrow \phi_2\{s\}$$

The definition of derivatives for the rest of temporal formulae is given together with the definition of the reduction relation $\Longmapsto$ on guarded biochemical programs. The latter is defined by structural induction on the LTL formulae by associating a strategy $\Theta(\phi)$ to each formula ϕ.

Definition 4.7 The reduction relation on guarded biochemical programs extends the interaction relation on biochemical programs with an evaluation of all atomic guards:

$$[K]_{\alpha_1\phi_1\wedge\ldots\wedge\alpha_n\phi_n} \Longmapsto [K']_{\alpha'_1\phi_1\wedge\ldots\wedge\alpha'_n\phi_n} \text{ if } [K]_{\alpha_i\phi_i} \Longmapsto [K']_{\alpha'_i\phi_i} \text{ for all } i$$

$$[K]_{\models^?\phi} \Longmapsto (\Theta(\phi)@[K']_{\models^?\phi})@[K]_{\models^?\phi} \text{ if } [K] \longrightarrow_{\equiv} [K']$$

$$[K]_{\models\phi} \Longmapsto [K']_{\models\phi} \text{ if } [K] \longrightarrow_{\equiv} [K']$$

$$[K]_{\not\models\phi} \Longmapsto [K']_{\not\models\phi} \text{ if } [K] \longrightarrow_{\equiv} [K']$$

where $\Theta(\phi)$ is the strategy associated to the formula ϕ.

In order to address the case where the current execution reaches an irreducible program with an inconclusive guard, we need to extend the congruence relation on guarded biochemical programs as follows:

$$[K]_{\models^?\phi} \equiv \Theta_{\downarrow}(\phi)@[K]_{\models^?\phi} \text{ if } [K] \text{ is irreducible}$$

To achieve the complete definition of the reduction relation on guarded biochemical programs, we have to define strategy mappings Θ and $\Theta_{\downarrow}$ for each type of LTL formulae ϕ used as a guard, relying in the intuition provided by derivatives. This is the purpose of the following five subsections.

4.2.1 Strategy encoding for structural formulae

The derivative of a structural formula φ in a state s is *true* if $s \models \varphi$ and *false* otherwise. The strategy encoding the temporal formula φ evaluates the application of the strategy $\tau(\varphi)$ on the current state s and returns the guard $\models \varphi$ if $s \models \varphi$, otherwise the guard $\not\models \varphi$.

$$\Theta(\varphi) \triangleq [Z]_{\models^?\varphi} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi), W_1 \Rrightarrow [Z]_{\models\varphi}, W_2 \Rrightarrow [Z]_{\not\models\varphi})$$

$$\Theta_{\downarrow}(\varphi) \triangleq \texttt{ifThenElse}(\tau(\varphi), [Z]_{\models^?\varphi} \Rrightarrow [Z]_{\models\varphi}, [Z]_{\models^?\varphi} \Rrightarrow [Z]_{\not\models\varphi})$$

4.2.2 Strategy encoding for $\mathsf{G}\varphi$ formulae

The derivative of the globally operator for a non-final state s and a final state $s_{\downarrow}$ is defined as follows:

$$(\mathsf{G}\varphi)\{s\} = \begin{cases} \mathit{false} & \text{if } \neg\varphi\{s\} \\ \mathsf{G}\varphi & \text{otherwise} \end{cases} \qquad (\mathsf{G}\varphi)\{s_{\downarrow}\} = \begin{cases} \mathit{false} & \text{if } \neg\varphi\{s_{\downarrow}\} \\ \mathit{true} & \text{otherwise} \end{cases}$$

The strategy encoding $\mathsf{G}\varphi$ tests the satisfaction of φ in the current state: if the answer is positive then the guard is inconclusive for the current execution and the verification continues with the same guard, otherwise the guard is negative meaning that the temporal formula $\mathsf{G}\varphi$ is not satisfied by the current execution. If

the current state is final, then the inconclusive guard turns into a positive one if this state satisfies φ as well, since the formula φ is satisfied in every state of the execution. The strategies Θ and $\Theta_{\downarrow}$ encoding $\mathsf{G}\varphi$ for non-final and final states are:

$$\Theta(\mathsf{G}\varphi) \triangleq [Z]_{\models^?\mathsf{G}\varphi} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi), W_1 \Rrightarrow [Z]_{\models^?\mathsf{G}\varphi}, W_2 \Rrightarrow [Z]_{\not\models\mathsf{G}\varphi})$$
$$\Theta_{\downarrow}(\mathsf{G}\varphi) \triangleq \texttt{ifThenElse}(\tau(\varphi), [Z]_{\models^?\mathsf{G}\varphi} \Rrightarrow [Z]_{\models\mathsf{G}\varphi}, [Z]_{\models^?\mathsf{G}\varphi} \Rrightarrow [Z]_{\not\models\mathsf{G}\varphi})$$

4.2.3 Strategy encoding for $\mathsf{F}\varphi$ formulae

The derivative of the eventually operator for a non-final state s and a final state $s_{\downarrow}$ is defined as follows:

$$(\mathsf{F}\varphi)\{s\} = \begin{cases} \mathit{true} & \text{if } \varphi\{s\} \\ \mathsf{F}\varphi & \text{otherwise} \end{cases} \qquad (\mathsf{F}\varphi)\{s_{\downarrow}\} = \varphi\{s_{\downarrow}\} = \begin{cases} \mathit{true} & \text{if } \varphi\{s_{\downarrow}\} \\ \mathit{false} & \text{otherwise} \end{cases}$$

The strategy encoding $\mathsf{F}\varphi$ tests if the current state satisfies φ. If it does, then one state satisfying φ in the current execution has been found, hence the inconclusive guard is turned to a positive one. Otherwise, the guard is passed to the next state as inconclusive. If the current state is final and it satisfies φ, then the execution satisfies the temporal formula $\mathsf{F}\varphi$; otherwise, the execution does not satisfy $\mathsf{F}\varphi$. The strategies Θ and $\Theta_{\downarrow}$ encoding $\mathsf{F}\varphi$ for non-final and final states are the following respectively:

$$\Theta(\mathsf{F}\varphi) \triangleq [Z]_{\models^?\mathsf{F}\varphi} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi), W_1 \Rrightarrow [Z]_{\models\mathsf{F}\varphi}, W_2 \Rrightarrow [Z]_{\models^?\mathsf{F}\varphi})$$
$$\Theta_{\downarrow}(\mathsf{F}\varphi) \triangleq \texttt{ifThenElse}(\tau(\varphi), [Z]_{\models^?\mathsf{F}\varphi} \Rrightarrow [Z]_{\models\mathsf{F}\varphi}, [Z]_{\models^?\mathsf{F}\varphi} \Rrightarrow [Z]_{\not\models\mathsf{F}\varphi})$$

4.2.4 Strategy encoding for $\varphi_1\mathsf{U}\varphi_2$ formulae

The derivatives of $\varphi_1\mathsf{U}\varphi_2$ for non-final and final states respectively are defined as follows:

$$(\varphi_1\mathsf{U}\varphi_2)\{s\} = \begin{cases} \mathit{true} & \text{if } \varphi_2\{s\} \\ (\varphi_1\mathsf{U}\varphi_2) & \text{if } \neg(\varphi_2\{s\}) \text{ and } \varphi_1\{s\} \\ \mathit{false} & \text{if } \neg(\varphi_2\{s\}) \text{ and } \neg(\varphi_1\{s\}) \end{cases}$$
$$(\varphi_1\mathsf{U}\varphi_2)\{s_{\downarrow}\} = \begin{cases} \mathit{true} & \text{if } \phi_2\{s_{\downarrow}\} \text{ or } (\neg(\varphi_2\{s_{\downarrow}\}) \text{ and } \varphi_1\{s_{\downarrow}\}) \\ \mathit{false} & \text{if } \neg(\varphi_2\{s_{\downarrow}\}) \text{ and } \neg(\varphi_1\{s_{\downarrow}\}) \end{cases}$$

The strategy encoding the temporal formula $\varphi_1\mathsf{U}\varphi_2$ tests first if the current state satisfies φ_2: if it does, then the inconclusive guard is transformed into a positive guard; otherwise, it tests if the current state satisfies φ_1: if it does, then we continue with an inconclusive guard (hence all previous and current states in the execution satisfy φ_1), else we found a state where neither φ_1 or φ_2 are satisfied, hence the formula $\varphi_1\mathsf{U}\varphi_2$ is not satisfied by the execution. The strategies Θ and $\Theta_{\downarrow}$ encoding $\varphi_1\mathsf{U}\varphi_2$ for non-final and final states are the following respectively:

$$\Theta(\varphi_1 \mathsf{U} \varphi_2) \triangleq [Z]_{\models^? \varphi_1 \mathsf{U} \varphi_2} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_2), W_1 \Rrightarrow [Z]_{\models \varphi_1 \mathsf{U} \varphi_2},$$
$$W_2 \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_1), W_3 \Rrightarrow [Z]_{\models^? \varphi_1 \mathsf{U} \varphi_2}, W_4 \Rrightarrow [Z]_{\not\models \varphi_1 \mathsf{U} \varphi_2}))$$
$$\Theta_{\downarrow}(\varphi_1 \mathsf{U} \varphi_2) \triangleq \texttt{ifThenElse}(\tau(\varphi_2), [Z]_{\models^? \varphi_1 \mathsf{U} \varphi_2} \Rrightarrow [Z]_{\models \varphi_1 \mathsf{U} \varphi_2},$$
$$[Z]_{\models^? \varphi_1 \mathsf{U} \varphi_2} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_1), W_3 \Rrightarrow [Z]_{\models \varphi_1 \mathsf{U} \varphi_2}, W_4 \Rrightarrow [Z]_{\not\models \varphi_1 \mathsf{U} \varphi_2}))$$

4.2.5 Strategy encoding for $\varphi_1 \mathsf{R} \varphi_2$ formulae

The derivatives of $\varphi_1 \mathsf{R} \varphi_2$ for non-final and final states respectively are defined as follows:

$$(\varphi_1 \mathsf{R} \varphi_2)\{s\} = \begin{cases} \mathit{true} & \text{if } \varphi_2\{s\} \text{ and } \varphi_1\{s\} \\ (\varphi_1 \mathsf{R} \varphi_2) & \text{if } \varphi_2\{s\} \text{ and } \neg(\varphi_1\{s\}) \\ \mathit{false} & \text{if } \neg(\varphi_2\{s\}) \end{cases}$$

$$(\varphi_1 \mathsf{R} \varphi_2)\{s_{\downarrow}\} = \begin{cases} \mathit{true} & \text{if } \phi_2\{s_{\downarrow}\} \\ \mathit{false} & \text{if } \neg(\varphi_2\{s_{\downarrow}\}) \end{cases}$$

The strategy encoding the temporal formula $\varphi_1 \mathsf{R} \varphi_2$ tests first if the current state satisfies φ_2: if it does, then the inconclusive guard is transformed into a positive guard; otherwise, it tests if the current state satisfies φ_1: if it does, then we continue with an inconclusive guard (hence all previous and current states in the execution satisfy φ_1), else we found a state where neither φ_1 or φ_2 are satisfied, hence the formula $\varphi_1 \mathsf{R} \varphi_2$ is not satisfied by the execution. The strategies Θ and $\Theta_{\downarrow}$ encoding $\varphi_1 \mathsf{R} \varphi_2$ for non-final and final states are the following respectively:

$$\Theta(\varphi_1 \mathsf{R} \varphi_2) \triangleq [Z]_{\models^? \varphi_1 \mathsf{R} \varphi_2} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_2),$$
$$W_1 \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_1), W_2 \Rrightarrow [Z]_{\models \varphi_1 \mathsf{R} \varphi_2}, W_3 \Rrightarrow [Z]_{\models^? \varphi_1 \mathsf{R} \varphi_2}), W_4 \Rrightarrow [Z]_{\not\models \varphi_1 \mathsf{R} \varphi_2})$$
$$\Theta_{\downarrow}(\varphi_1 \mathsf{R} \varphi_2) \triangleq \texttt{ifThenElse}(\tau(\varphi_2), [Z]_{\models^? \varphi_1 \mathsf{R} \varphi_2} \Rrightarrow [Z]_{\models \varphi_1 \mathsf{R} \varphi_2}, [Z]_{\models^? \varphi_1 \mathsf{R} \varphi_2} \Rrightarrow [Z]_{\not\models \varphi_1 \mathsf{R} \varphi_2})$$

The properties of the strategy definitions above are summarised by the next proposition:

Proposition 4.8 *Let $s_{\models^? \varphi}$ be the current guarded biochemical program. If the next biochemical program computed by the evaluation rule* (**Interaction**) *is s', then its guard is computed, according to Def. 4.7, as the normal of the application $(\Theta(\phi)@s'_{\models^? \phi})@s_{\models^? \phi}$. Otherwise, if the biochemical program s is irreducible, then its guard is computed by reducing to normal form the application $(\Theta_{\downarrow}(\phi)@s_{\models^? \phi})$. Figure 4 lists these computations when the current guard ϕ ranges over φ, $\mathsf{G}\varphi$, $\mathsf{F}\varphi$, $\varphi_1 \mathsf{U} \varphi_2$ and $\varphi_1 \mathsf{R} \varphi_2$.*

We are now ready to prove the correctness of our encoding of the runtime verification of LTL formulae in the calculus using strategies, based on the result of Theorem 4.6.

Theorem 4.9 (Correctness) *For any LTL formula ϕ guarding a biochemical program s^0 and for any finite trace $u = s^0 s^1 \ldots s^n$, u is a* good *prefix for ϕ if and only if either:*

	$\Theta(\varphi)$	$\Theta(\mathsf{G}\varphi)$	$\Theta(\mathsf{F}\varphi)$	$\Theta_{\downarrow}(\varphi)$	$\Theta_{\downarrow}(\mathsf{G}\varphi)$	$\Theta_{\downarrow}(\mathsf{F}\varphi)$
$s \models \varphi$	$s'_{\models\varphi}$	$s'_{\models^?\mathsf{G}\varphi}$	$s'_{\models\mathsf{F}\varphi}$	$s_{\models\varphi}$	$s_{\models\mathsf{G}\varphi}$	$s_{\models\mathsf{F}\varphi}$
$s \not\models \varphi$	$s'_{\not\models\varphi}$	$s'_{\not\models\mathsf{G}\varphi}$	$s'_{\models^?\mathsf{F}\varphi}$	$s_{\not\models\varphi}$	$s_{\not\models\mathsf{G}\varphi}$	$s'_{\not\models\mathsf{F}\varphi}$

	$\Theta(\varphi_1\mathsf{U}\varphi_2)$	$\Theta_{\downarrow}(\varphi_1\mathsf{U}\varphi_2)$
$s \models \varphi_2$	$s'_{\models\varphi_1\mathsf{U}\varphi_2}$	$s_{\models\varphi_1\mathsf{U}\varphi_2}$
$s \models (\varphi_1 \wedge \neg\varphi_2)$	$s'_{\models^?\varphi_1\mathsf{U}\varphi_2}$	$s_{\models\varphi_1\mathsf{U}\varphi_2}$
$s \not\models (\varphi_1 \vee \varphi_2)$	$s'_{\not\models\varphi_1\mathsf{U}\varphi_2}$	$s_{\not\models\varphi_1\mathsf{U}\varphi_2}$

	$\Theta(\varphi_1\mathsf{R}\varphi_2)$	$\Theta_{\downarrow}(\varphi_1\mathsf{R}\varphi_2)$
$s \models (\varphi_1 \wedge \varphi_2)$	$s'_{\models\varphi_1\mathsf{R}\varphi_2}$	$s_{\models\varphi_1\mathsf{R}\varphi_2}$
$s \models (\neg\varphi_1 \wedge \varphi_2)$	$s'_{\models^?\varphi_1\mathsf{R}\varphi_2}$	$s_{\models\varphi_1\mathsf{R}\varphi_2}$
$s \not\models \varphi_2$	$s'_{\not\models\varphi_1\mathsf{R}\varphi_2}$	$s_{\not\models\varphi_1\mathsf{R}\varphi_2}$

Fig. 4. Normal forms of $(\Theta(\phi)@s'_{\models^?\phi})@s_{\models^?\phi}$ and $(\Theta_{\downarrow}(\phi)@s_{\models^?\phi})$ for $s_{\models^?\phi}$ the current guarded biochemical program and s' the next unguarded biochemical program

- *$\exists i$, $0 \leq i \leq n$, such that for all $j < i$, $s^{j-1}_{\models^?\phi} \Longmapsto s^{j}_{\models^?\phi}$ and $s^{i-1}_{\models^?\phi} \Longmapsto s^{i}_{\models\phi}$, or*
- *$s^{0}_{\models^?\phi} \Longmapsto s^{1}_{\models^?\phi} \Longmapsto \ldots \Longmapsto s^{n}_{\models^?\phi}$ and $s^{n}_{\models^?\phi} \equiv s^{n}_{\models\phi}$.*

We obtain a similar result for a bad *prefix if we replace the guard $\models \phi$ by $\not\models \phi$.*

4.3 Example: Repairing a property of the loaning service specification

Consider a biochemical program P describing the library loaning service from Examples 2.1 consisting of: the library of the Maths department with 6 Statistics textbooks and 6 Maths students, the library of the CS department with 3 Statistics textbooks and 6 CS students, and the strategies given in Example 2.4. We guard the program P with the safety formula $\phi = \mathsf{G}(\Diamond S(id, dpt, 2) \wedge \Diamond B(b, nil, 0)))$ which requires that a student should always be able to borrow a book from any departmental library. However, this guard may become false if P evolves to a state where both libraries lent all their textbooks and a Maths (resp. CS) student demands a book. In order to solve this problem and offer the students an equal chance to study, exceptionally a CS (resp. Maths) student is forced to return a book if borrowed from the other department, otherwise a student from the same department. Such behaviour is modelled by the strategy $R = \texttt{try}(\mathbf{FRet1}, \mathbf{FRet2})$ with **FRet1** and **FRet2** depicted in Fig. 5. Therefore we provide the guard ϕ with the repairing

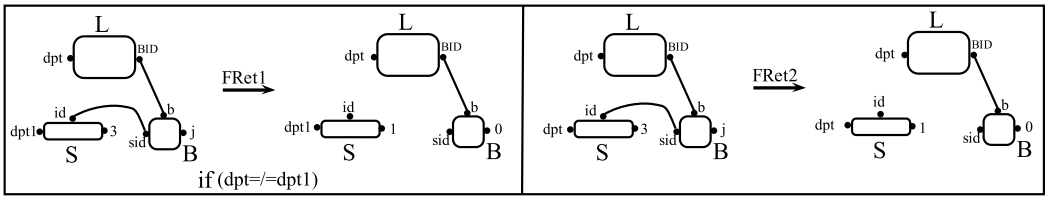

Fig. 5. Rewrite rules describing a student returning a book to the library of another department and of the same department

strategy above and have in the calculus:

$$s_{\not\models\phi(R)} \equiv (R@s)_{\models^?\phi(R)}$$

such that, every time the guard ϕ is not satisfied, the guard is transformed into an inconclusive guard and the repairing rule is injected into the program.

4.4 On LTL Formulae Guards with Two Modal Operators

We have seen by now how to encode the LTL formulae built with at most one path operator using strategies such that their verification during the program execution amounts to strategy evaluation. However there are plenty of interesting LTL formulae with two modal operators. For instance, in the library example the formula $\mathsf{G}(\diamond S(id, dpt, 2) \rightarrow \mathsf{F}(\diamond S(id, dpt, 3)))$ says that always a student request of a book is eventually fulfilled.

In the following we sketch a methodology of encoding guards consisting of LTL formulae built using at most two modal operators such as $\mathsf{G}(\varphi_1 \rightarrow \mathsf{X}\varphi_2)$, $\mathsf{G}(\varphi_1 \wedge \mathsf{X}\varphi_2)$, $\mathsf{G}(\varphi_1 \vee \mathsf{X}\varphi_2)$, $\mathsf{F}(\varphi_1 \rightarrow \mathsf{X}\varphi_2)$, $\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2)$, $\mathsf{F}(\varphi_1 \vee \mathsf{X}\varphi_2)$, $\mathsf{G}(\varphi_1 \rightarrow \mathsf{F}\varphi_2)$, $\mathsf{G}(\varphi_1 \mathsf{U} \varphi_2)$, or $\mathsf{F}(\varphi_1 \mathsf{R} \varphi_2)$. The keys of this encoding are the use of derivatives and LTL formulae consumption [18].

During the execution of a guarded program $s_{\models^?\phi}$, the guard ϕ is consumed or reduced to a residual LTL formula ϕ', in order to reduce the satisfaction problem of an LTL formula with two modal operators or with the *next* operator X to the satisfaction problem of a simpler LTL formula. Here we want to reduce to a formula with at most one modal operator and without the next operator, whose satisfaction problem has been handled in Sect. 4.2. However, by consuming the guard ϕ, the information about the initial temporal formula to be tested may be lost. We avoid this by annotating the guards along the program execution with the initial guard: for ϕ' the residual guard and ϕ the initial guard, we denote by ϕ'/ϕ the *annotated guard*.

In the following we define the execution of $\mathsf{X}\varphi$-guarded programs. The derivatives of $\mathsf{X}\varphi$ in a non-final state s and in a final state $s_\downarrow$ respectively are defined as follows:

$$(\mathsf{X}\varphi)\{s\} = \varphi/\mathsf{X}\varphi \qquad (\varphi/\mathsf{X}\varphi)\{s\} = \begin{cases} \mathit{true} & \text{if } \varphi\{s\} \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

$$(\mathsf{X}\varphi)\{s_\downarrow\} = \mathit{false} \qquad (\varphi/\mathsf{X}\varphi)\{s_\downarrow\} = \begin{cases} \mathit{true} & \text{if } \varphi\{s_\downarrow\} \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

It is worth noticing that the derivative of $\mathsf{X}\varphi$ reduces to the residual formula φ

while carrying along its original formula $\mathsf{X}\varphi$. Then the strategies encoding the next operator are:

$$\Theta(\mathsf{X}\varphi) \triangleq [Z]_{\models^?\mathsf{X}\varphi} \Rrightarrow (W \Rrightarrow [Z]_{\models^?\varphi/\mathsf{X}\varphi})$$
$$\Theta(\varphi/\mathsf{X}\varphi) \triangleq [Z]_{\models^?\varphi/\mathsf{X}\varphi} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi), W_1 \Rrightarrow [Z]_{\models\varphi/\mathsf{X}\varphi}, W_2 \Rrightarrow [Z]_{\not\models\varphi/\mathsf{X}\varphi})$$
$$\Theta_{\downarrow}(\mathsf{X}\varphi) \triangleq [Z]_{\models^?\mathsf{X}\varphi} \Rrightarrow [Z]_{\not\models\mathsf{X}\varphi}$$
$$\Theta_{\downarrow}(\varphi/\mathsf{X}\varphi) \triangleq \texttt{ifThenElse}(\tau(\varphi), [Z]_{\models^?\varphi/\mathsf{X}\varphi} \Rrightarrow [Z]_{\models\varphi/\mathsf{X}\varphi}, [Z]_{\models^?\varphi/\mathsf{X}\varphi} \Rrightarrow [Z]_{\not\models\varphi/\mathsf{X}\varphi})$$

with $\models (\varphi/\phi) \equiv \models \phi$ and $\not\models (\varphi/\phi) \equiv \not\models \phi$.

Theorem 4.10 *For any LTL formula $\mathsf{X}\varphi$ guarding a biochemical program s^0 and for any finite trace $u = s^0 s^1 \ldots s^n$, u is a* good *prefix for $\mathsf{X}\varphi$ if and only if $n \geq 1$ and $s^0_{\models^?\mathsf{X}\varphi} \Longrightarrow s^1_{\models^?\varphi/\mathsf{X}\varphi} \Longrightarrow s^2_{\models\mathsf{X}\varphi}$. The finite trace u is a* bad *prefix for $\mathsf{X}\varphi$ if and only if $n = 0$ or $n \geq 1$ and $s^0_{\models^?\mathsf{X}\varphi} \Longrightarrow s^1_{\models^?\varphi/\mathsf{X}\varphi} \Longrightarrow s^2_{\not\models\mathsf{X}\varphi}$*

Having illustrated the consumption of guards on encoding the formula $\mathsf{X}\varphi$, we define in the following the strategy encoding a two-temporal modal operator LTL formula $\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2)$ for a non final-state s. Its derivative is defined as:

$$\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2)\{s\} = \begin{cases} \varphi_2/\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2) & \text{if } \varphi_1\{s\} \\ \mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2) & \textit{otherwise} \end{cases}$$

Then the encoding strategies of $\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2)$ are the following:

$$\Theta(\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2)) \triangleq [Z]_{\models^?\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2)} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_1),$$
$$W_1 \Rrightarrow [Z]_{(\models^?\varphi_2/\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2)) \wedge (\models^?\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2))}, W_2 \Rrightarrow [Z]_{\models^?\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2)})$$
$$\Theta(\varphi_2/\mathsf{F}(\varphi_1 \wedge \mathsf{X}\varphi_2)) \triangleq [Z]_{\models^?\varphi_2/\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2)} \Rrightarrow \texttt{ifThenElse}(\tau(\varphi_2),$$
$$W_1 \Rrightarrow [Z]_{\models\varphi_2/\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2)}, W_2 \Rrightarrow [Z]_{\not\models\varphi_2/\mathsf{F}(\varphi_1\wedge\mathsf{X}\varphi_2)})$$

The other LTL formulae mentioned at the beginning of section can be carefully encoded using strategies, by following the top-down decomposition approach, according to the temporal operator on top, until reaching a residual LTL formula consisting of a structural formula, and by keeping track of the decomposition history.

5 Conclusions and Future Work

The main contributions of this paper are an abstract biochemical calculus to model autonomous systems with runtime verification capabilities for critical properties and a self-repairing method for the calculus when critical properties are violated.

In Sect. 4.4 we only gave a glimpse of how to define strategies encoding LTL formulae built using two temporal operators. Based on these ideas, we can synthesise a methodology or, even better, an automatic procedure which takes as input an LTL formula built using at most two temporal operators and produce the encoding strategy. A challenge will be then to encode even more complex LTL formulae.

For future work we plan to implement the runtime verification technique described in this paper in the PORGY system [2], an environment for visual modelling of complex systems through graphs and graph rewriting rules. PORGY is still under development but already provides tools to visualise traces of rewriting, and a strategy language has been designed in particular to guide the construction of the derivation tree. We envisage applications to wireless sensor networks and biochemical signalling pathways to accompany their formal model development by analysing random executions and critical properties.

References

[1] Andrei, O., "A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems," Ph.D. thesis, Institut National Polytechnique de Lorraine, http://tel.archives-ouvertes.fr/docs/00/35/00/13/PDF/thesis-OanaAndrei.pdf (2008).

[2] Andrei, O., M. Fernández, H. Kirchner, G. Melançon, O. Namet and B. Pinaud, *PORGY: Strategy Driven Interactive Transformation of Graphs*, in: R. Echahed, editor, *TERMGRAPH*, EPTCS **48**, 2011, pp. 54–68.

[3] Andrei, O. and H. Kirchner, *A Rewriting Calculus for Multigraphs with Ports.*, in: *Proceedings of RULE'07*, Electronic Notes in Theoretical Computer Science **219**, 2008, pp. 67–82.

[4] Andrei, O. and H. Kirchner, *A Higher-Order Graph Calculus for Autonomic Computing*, in: M. a. Lipshteyn, editor, *Graph Theory, Computational Intelligence and Thought. Golumbic Festschrift*, Lecture Notes in Computer Science **5420** (2009), pp. 15–26.

[5] Andrei, O. and H. Kirchner, *A Port Graph Calculus for Autonomic Computing and Invariant Verification*, Electronic Notes in Theoretical Computer Science **253** (2009), pp. 17–38.

[6] Banâtre, J.-P., P. Fradet and Y. Radenac, *Higher-Order Chemical Programming Style*, in: J.-P. Banâtre, P. Fradet, J.-L. Giavitto and O. Michel, editors, *UPP*, Lecture Notes in Computer Science **3566** (2004), pp. 84–95.

[7] Banâtre, J.-P., P. Fradet and Y. Radenac, *Programming Self-Organizing Systems with the Higher-Order Chemical Language*, International Journal of Unconventional Computing **3** (2007), pp. 161–177.

[8] Banâtre, J.-P., N. L. Scouarnec, T. Priol and Y. Radenac, *Towards "Chemical" Desktop Grids*, in: *eScience* (2007), pp. 135–142.

[9] Bauer, A., M. Leucker and C. Schallhart, *Monitoring of Real-Time Properties*, in: S. Arun-Kumar and N. Garg, editors, *FSTTCS*, Lecture Notes in Computer Science **4337** (2006), pp. 260–272.

[10] Bauer, A., M. Leucker and C. Schallhart, *Runtime Verification for LTL and TLTL*, Technical Report TUM-I0724, Technische Universität München (2007).

[11] Berry, G. and G. Boudol, *The Chemical Abstract Machine.*, Theoretical Computer Science **96** (1992), pp. 217–248.

[12] Cirstea, H. and C. Kirchner, *The Rewriting Calculus - Part I and II*, Logic Journal of the IGPL **9** (2001), pp. 427—498.

[13] Giavitto, J.-L. and O. Michel, *MGS: a Rule-Based Programming Language for Complex Objects and Collections.*, Electronic Notes in Theoretical Computer Science **59** (2001), pp. 286–304.

[14] Kephart, J. O. and D. M. Chess, *The Vision of Autonomic Computing*, IEEE Computer **36** (2003), pp. 41–50.

[15] Kirchner, C., F. Kirchner and H. Kirchner, *Strategic computations and deductions*, in: C. Benzmueller, C. E. Brown, J. Siekmann and R. Statman, editors, *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday (Paperback)*, Studies in Logic, Mathematical Logic and Foundations **17** (2008), pp. 339–364.

[16] Pazat, J.-L., T. Priol and C. Tedeschi, *Towards a Chemistry-Inspired Middleware to Program the Internet of Services*, ERCIM News **2011** (2011), p. 34.

[17] Pnueli, A., *The Temporal Logic of Programs,*, in: *FOCS* (1977), pp. 46–57.

[18] Sen, K., G. Rosu and G. Agha, *Generating Optimal Linear Temporal Logic Monitors by Coinduction*, in: V. A. Saraswat, editor, *ASIAN*, Lecture Notes in Computer Science **2896** (2003), pp. 260–275.