Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 227 (2009) 37–58

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Translating Stochastic CLS into Maude

Thomas Anung Basuki[a,b,1] Antonio Cerone[a,2] Paolo Milazzo[b,3]

[a] *International Institute for Software Technology, United Nations University, Macau SAR, China*

[b] *Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy*

Abstract

This paper describes preliminary results on the application of statistical model-checking to systems described with Stochastic CLS. Stochastic CLS is a formalism based on term rewriting that allows biomolecular systems to be described by taking into account their structure and by allowing very general events to be modelled. Statistical model-checking is an analysis technique that permits properties of a system to be studied on the results of a number of stochastic simulations. We choose Real-Time Maude as a tool that supports the modelling and analysis of systems with real-time properties. We adapt Gillespie's algorithm for simulating chemical systems into our approach. The resulting method is applied to analyse some simple examples and a model of the lactose operon regulation in E.coli.

Keywords: Calculus of Looping Sequences, Maude, model-checking, biological system.

1 Introduction

In the last few years many formalisms have been either adapted or defined to model biomolecular systems [20,19,8,11,6]. The use of formal means in the description of biomolecular systems allows models to be constructed compositionally and unambiguously, and allows the application of analysis techniques that are common in Computer Science, but almost unknown to biologists.

Biologists usually model biomolecular systems by means of differential equations. These can be studied analytically and numerically in order to understand, for example, the average behaviour of a system and its sensitivity to perturbations in the system parameters and in the initial conditions. Moreover, stochastic simulation has become recently a widely followed approach to study the behaviour of biomolecular systems. Such a technique can be applied repeatedly to obtain a number of possible behaviours of a system. The results of stochastic simulations

[1] Email: anung@iist.unu.edu

[2] Email: antonio@iist.unu.edu

[3] Email: milazzo@di.unipi.it

doi:10.1016/j.entcs.2008.12.103

are more accurate than those obtained by the numerical solution of the differential equations, in particular when the number of components of the system is very small.

Model-checking is one of the analysis techniques commonly used in Computer Science that could be applied to biomolecular systems. This technique permits the verification of properties of a system (expressed as logical formulas) by exploring all possible behaviours the system may have. To take into account all the possible behaviours of a system (each associated with a probability) a stochastic (or probabilistic) tool has to be used, such as PRISM [14], Murphi [18] or PMaude [3].

In this paper we face the problem of applying model-checking to biomolecular systems described with Stochastic CLS [5] and we show some preliminary results. Stochastic CLS is the stochastic extension of the Calculus of Looping Sequences (CLS), that is a formalism based on term rewriting that allows biomolecular systems to be described by taking into account their structure and by allowing very general events to be modelled. Among the tools we mentioned above, the most suitable for the application to Stochastic CLS is PMaude. Such a tool is the probabilistic extension of Maude [10], a rewrite-based modelling language. The other tools we mentioned, namely PRISM and Murphi, are less suitable because they do not offer a natural way to describe the structures considered in Stochastic CLS.

Unfortunately, the VESTA tool [1], the only model-checker for PMaude models is not accessible due to a problem in the download procedure. Since PMaude does not have model-checking capabilities, we will use Real-Time Maude [17] an extension of Maude with time rather than probabilities. We will follow the approach of Thorvaldsen and Olveczky [16] to add probabilistic behaviour to a Real-Time Maude specification: roughly speaking, we will exploit a pseudo-random number generator to develop a stochastic simulation algorithm used by the engine of Real-Time Maude to construct and analyse a possible behaviour of the system. Different behaviours will be obtained by initializing the number generator with different seeds. A consequence of this approach is that we will loose the completeness of the state space exploration (we can no longer ensure that all possible behaviours are considered). However, this approach may allow us to analyse very complex and infinite state systems, as essentially we restrict the verification to a finite (arbitrary) number of possible behaviours. This approach is also known as *statistical model-checking* [21].

As regards related work, Andrei, Ciobanu and Lucanu [4] define an operational semantics of P systems which is used to give a translation of such systems into Maude specifications. The aim of such a translation is to obtain executable specifications of P systems (that can also be verified). Their work was very challenging because P systems are based on a notion of maximal parallelism that is not easy to implement in Maude. Our work has a different purpose. We are interested in studying biomolecular systems and the major challenge, in our case, is to handle stochasticity. Other examples of application of model-checking to biomolecular systems are cited in the references [13,9]. An example of well-established formal framework that can be used to model, simulate and model-check descriptions of biological systems is the PEPA process algebra and related tools [2]. Calder, Gilmore and Hilston use the PEPA process algebra and its related tools to model and analyse the influence of Raf Kinase Inhibitor Protein (RKIP) on the Extracellular signal Regulated Kinase (ERK) signalling pathway [7].

The rest of the paper is structured as follows. Section 2 contains a brief description of CLS and Stochastic CLS. Section 3 gives an overview of Maude and Real-Time Maude and describe the translation of Stochastic CLS into the latter. Section 4 shows the analysis of some examples and of a model of the lactose operon regulation in E.coli. Section 5 concludes the paper with an overview of related work.

2 Calculi of Looping Sequences

In this section we recall the Calculus of Looping Sequences (CLS). We assume a possibly infinite alphabet $\mathcal{E}$ of symbols ranged over by $a, b, c, \ldots$

Definition 2.1 *Terms* T and *Sequences* S are given by the following grammar:

$$T \ ::= \ S \quad \big| \quad \left(T\right)^L \rfloor\, T \quad \big| \quad T \mid T \qquad\qquad S \ ::= \ \epsilon \quad \big| \quad a \quad \big| \quad S \cdot S$$

where a is a generic element of $\mathcal{E}$, and ϵ represents the empty sequence. We denote with $\mathcal{T}$ the infinite set of terms, and with $\mathcal{S}$ the infinite set of sequences.

In CLS we have a sequencing operator $\_ \cdot \_$, a parallel composition operator $\_ \mid \_$, a looping operator $\left(\_\right)^L$ and a containment operator $\_ \rfloor \_$. Sequencing can be used to concatenate elements of the alphabet $\mathcal{E}$. The empty sequence ϵ denotes the concatenation of zero symbols. By definition, looping and containment are always applied together, hence we can consider them as a single binary operator $\left(\_\right)^L \rfloor \_$. Brackets can be used to indicate the order of application of the operators, and we assume $\left(\_\right)^L \rfloor \_$ to have precedence over $\_ \mid \_$.

Sequences of CLS can model DNA/RNA strands by describing each gene with a symbol of the alphabet. Similarly, they can be used to model proteins by describing protein interaction sites with alphabet symbols. Membranes are closed surfaces, often interspersed with proteins, which may have a content. Looping and containment allow the representation of membranes with their contents. For example, the term $\left(a \mid b\right)^L \rfloor\, c$ represents a membrane with the elements a and b on its surface and containing the element c. Other macro–molecules can be modeled as single alphabet symbols, or as short sequences. Finally, juxtaposition of entities can be described by the parallel composition of their representations.

In CLS we may have syntactically different terms representing the same structure. We introduce a structural congruence relation to identify such terms.

Definition 2.2 The *structural congruence relations* $\equiv_S$ and $\equiv$ are the least congruence relations on sequences and on terms, respectively, such that $\equiv$ includes $\equiv_S$ and satisfying the following rules:

$$S_1 \cdot (S_2 \cdot S_3) \equiv_S (S_1 \cdot S_2) \cdot S_3 \qquad S \cdot \epsilon \equiv_S \epsilon \cdot S \equiv_S S$$
$$T_1 \mid T_2 \equiv T_2 \mid T_1 \qquad T_1 \mid (T_2 \mid T_3) \equiv (T_1 \mid T_2) \mid T_3 \qquad T \mid \epsilon \equiv T$$

Rules of the structural congruence state the associativity of $\cdot$ and $\mid$, the commutativity of the latter and the neutral role of ϵ.

Rewrite rules will be defined essentially as pairs of terms, in which the first term describes the portion of the system in which the event modeled by the rule may occur, and the second term describes how that portion of the system changes when the event occurs. In the terms of a rewrite rule we allow the use of variables. As a consequence, a rule will be applicable to all terms which can be obtained by properly instantiating its variables. Usually, CLS is defined by considering variables of three kinds, namely term, sequence and element variables, that can be instantiated with terms, sequences and individual alphabet symbols, respectively. Here, for the ease of the translation into Maude, we consider only one type of variables, namely term variabels. We assume a set of (term) variables $\mathcal{V}$ ranged over by $X, Y, Z, \ldots$. A pattern is a term which may include variables.

Definition 2.3 *Patterns P of CLS are given by the following grammar:*

$$P \ ::= \ S \ \big| \ (P)^L \rfloor P \ \big| \ P \mid P \ \big| \ X$$

where a is a generic element of $\mathcal{E}$ and X is a generic element of $\mathcal{V}$. We denote with $\mathcal{P}$ the infinite set of patterns.

We assume the structural congruence relation to be trivially extended to patterns. An *instantiation* is a partial function $\sigma : \mathcal{V} \to \mathcal{T}$. Given $P \in \mathcal{P}$, with $P\sigma$ we denote the term obtained by replacing each occurrence of each variable $\rho \in \mathcal{V}$ appearing in P with the corresponding term $\sigma(\rho)$. With Σ we denote the set of all the possible instantiations and, given $P \in \mathcal{P}$, with $Var(P)$ we denote the set of variables appearing in P. Now we define rewrite rules.

A *rewrite rule* is a pair of patterns (P_1, P_2), denoted with $P_1 \mapsto P_2$, where $P_1, P_2 \in \mathcal{P}$, $P_1 \not\equiv \epsilon$ and such that $Var(P_2) \subseteq Var(P_1)$. A rule $P_1 \mapsto P_2$ states that a term $P_1\sigma$, obtained by instantiating variables in P_1 by instantiation function σ, can be transformed into the term $P_2\sigma$. We give the semantics of CLS as a transition system, in which states correspond to terms, and transitions correspond to rule applications.

Definition 2.4 Given a finite set of rewrite rules $\mathcal{R}$, the *semantics* of CLS is the least transition relation $\to$ on terms closed under structural congruence $\equiv$ and satisfying the following inference rules:

$$1.\ \frac{P_1 \mapsto P_2 \in \mathcal{R} \quad P_1\sigma \not\equiv \epsilon \quad \sigma \in \Sigma}{P_1\sigma \to P_2\sigma} \qquad 2.\ \frac{T_1 \to T_2}{T \mid T_1 \to T \mid T_2}$$

$$3.\ \frac{T_1 \to T_2}{(T)^L \rfloor T_1 \to (T)^L \rfloor T_2} \qquad 4.\ \frac{T_1 \to T_2}{(T_1)^L \rfloor T \to (T_2)^L \rfloor T}$$

A *model* in CLS is given by a term describing the initial state of the system and by a set of rewrite rules describing all the events that may occur.

Stochastic CLS is the extension of CLS with stochastic rates associated with the application of rewrite rules. More precisely, rewrite rules in Stochastic CLS are enriched with a rate constant that is multiplied, in the semantics, by the number of different occurrences of instantiations of the left hand side of the rule in the term to which the rule is applied. This corresponds to what usually done in chemical kinetics, where the rate of occurrence of a chemical reaction is computed by multiplying the kinetic constant of the reaction by the number of possible combinations of reactants. Rather than giving the (quite complex) definition of the semantics of Stochastic CLS, we will directly show, in the following, how we compute the rate of application of a Stochastic CLS rule in a Maude specification.

3 Translation of Stochastic CLS into Real-Time Maude

Maude [10] is a specification language equipped with efficient analysis tools, which supports three modelling paradigms: algebraic style (via equations), rewrite logic (via rewrite rules) and object-oriented paradigm (via classes and messages). System components are modelled in Maude as modules. A *functional module* is a Maude module that contains only the *signature* and *equations* of a system. The signature specifies type structure (in terms of sorts, subsorts and kinds) and operators. Equations are applied from left to right to simplify expressions. Functional modules are enclosed by keywords `fmod` and `endfm`. A *system module* is a functional module enriched with rewrite rules. System modules are enclosed by keywords `mod` and `endm`. An *object-oriented module* is basically a system module, equipped with a richer syntax to define classes (and objects), messages and configurations of objects and messages. Object-oriented modules are enclosed by keywords `omod` and `endom`.

Real-Time Maude [17] extends Maude by introducing two more kinds of module, *timed modules* and *object-oriented timed modules*. Timed modules are enclosed by keywords `tmod` and `endtm`, and object-oriented timed modules are enclosed by keywords `tomod` and `endtom`. Any timed module or object-oriented timed module automatically imports a predefined TIME module, which defines abstract time domains. We can choose between two kinds of time domain: discrete time, which uses natural numbers, and dense time, which uses positive rational numbers.

There are two kinds of rewrite rules in Real-Time Maude: instantaneous rules and tick rules. Instantaneous (conditional) rewrite rules are written as

```
crl [l] :  t => t' if cond .
```

where an analogous unconditional rule can be obtained by replacing `crl` with `rl` and by omitting `if cond`. A conditional rewrite rule can only fire if the condition *cond* is satisfied. Label *l* is the name of the rewrite rule, which is useful for debugging purpose. Pattern *t* matches the system state and, after the rule is executed, changes to *t'*. These rewrite rules take 0 time to occur. Tick rules are written as

```
crl [l] : {t} => {t'} in time τ if cond .
```

where τ denotes the duration of the rewrite.

Tick rules can be either deterministic or nondeterministic. Time-deterministic tick rules have the following forms:

```
crl [l] : {t} => {t'} in time c if cond .
```
where c is a constant.

Time-nondeterministic tick rules have one of the following forms:
```
crl [l] : {t} => {t'} in time x if cond /\ x r u /\ cond'
[nonexec] .
crl [l] : {t} => {t'} in time x if cond [nonexec] .
```
where x is a time variable which does not occur in t and which is not initialised in the condition, and r is either $<$ or $<=$. Attribute `nonexec` ensures that the rule is not directly executed in Maude. In fact, variable x is not assigned to any value, making nondeterministic tick rules nonexecutable in Maude. A time-sampling strategy, which assigns a value to variable x, must be chosen by the user at run-time to enable the execution of these rules.

In order to define the translation of Stochastic CLS into Real-Time Maude we first give a formal translation of CLS into rewriting logic (the formalism underlying Maude) in Section 3.1. After this, since Real-Time Maude is not stochastic, we show in Section 3.2 how Gillespie's stochastic simulation algorithm can be implemented in Real-Time Maude so to add stochasticity to CLS models. This will require a function to compute the number of occurrences of left-hand sides of CLS rewrite rules in the term representing the current state of a stochastic simulation. Such a function is given in Section 3.3. Finally, we show how Stochastic CLS terms are rules can be actually represented in Real-Time Maude in Section 3.4.

3.1 Translation of CLS into Rewriting Logic

Rewriting logic is parametrized with respect to the version of the underlying equational logic. Since the abstract syntax of CLS consists of two different sorts (terms and sequences), for a natural translation of CLS into rewriting logic we should consider a many-sorted logic. However, since we defined CLS sequences in such a way that they cannot contain variables, we can consider them as taken from a given countably infinite set $\mathcal{S}$. This permits us to rewrite the syntax of CLS terms and patterns in the following equivalent (single sorted) way:

$$\begin{array}{l} T \ ::= \epsilon \quad \big| \quad S \quad \big| \quad \left(T\right)^L \rfloor\, T \quad \big| \quad T \,|\, T \\ P \ ::= \epsilon \quad \big| \quad S \quad \big| \quad \left(P\right)^L \rfloor\, P \quad \big| \quad P \,|\, P \quad \big| \quad X \end{array}$$

where $S \in \mathcal{S}$ and $X \in \mathcal{V}$.

Now, we recall from Marti-Oliet and Meseguer [15] the definition of rewriting logic based on an unsorted equational logic and with unconditional rewrite rules. Given a *signature* (Σ, E), where $\Sigma = \{\Sigma_n \mid n \in I\!N\}$ is a ranked alphabet and E is a set of equations on Σ-terms, *sentences* of rewriting logic have the form $[t]_E \longrightarrow [t']_E$, where t and t' are Σ-terms possibly involving some variables from the countably infinite set $X = \{x_1, \ldots, x_n, \ldots\}$, and $[t]_E$ and $[t']_E$ are E-equivalence classes of t and t'. In what follows we shall denote the set of all such equivalence classes with $T_{\Sigma,E}(X)$ and we always omit the subscript E from the notation of E-equivalence classes.

A *rewrite theory* $\mathbf{R}$ is a 4-tuple $\mathbf{R} = (\Sigma, E, L, R)$ where (Σ, E) is a signature, L is a set of labels and $R \subseteq L \times T_{\Sigma,E}(X)^2$ is a set of *rewrite rules*. We denote a rewrite rule (r, t, t') with $r : [t] \longrightarrow [t']$.

A rewrite theory $\mathbf{R}$ *entails* a sequent $[t] \longrightarrow [t']$, namely $R \vdash [t] \longrightarrow [t']$, if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction*:

$$(r1)\quad \frac{}{[t] \longrightarrow [t]} \qquad (r2)\quad \frac{[w_1] \longrightarrow [w_1'] \quad \dots \quad [w_n] \longrightarrow [w_n']}{[t(w_1, \dots, w_n/x_1, \dots, x_n)] \longrightarrow [t'(w_1', \dots, w_n'/x_1, \dots, x_n)]}$$

$$(r3)\quad \frac{[t_1] \longrightarrow [t_1'] \quad \dots \quad [t_n] \longrightarrow [t_n']}{[f(t_1, \dots, t_n)] \longrightarrow [f(t_1', \dots, t_n')]} \qquad (r4)\quad \frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_2]}$$

where $(r2)$ applies for each rewrite rule $r : [t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)]$ in R.

The syntax and the structural congruence of CLS can be translated easily into a signature (Σ, E). We have $\Sigma = \Sigma_0 \cup \Sigma_2$ where $\Sigma_0 = \mathcal{S} \cup \{\epsilon\}$, $\Sigma_2 = \{(\_)^L \rfloor \_, \_ \mid \_\}$ and E containing equations stating the commutativity and associativity of $\_ \mid \_$ and the neutral role of ϵ with respect to $\_ \mid \_$. The variables that can appear in Σ-terms coincide with CLS variables. A set of CLS rewrite rules $\mathcal{R}$ can be translated into a rewrite theory (Σ, E, L, R) as follows. Let us assume that $\mathcal{R}$ contains n rules, (Σ, E) is the signature obtained from the syntax and the structural congruence of CLE, L is the set of labels $\{R1, \dots, Rn\}$ and each rule $P_1 \mapsto P_2$ in $\mathcal{R}$ (let us assume it is the i-th rule of $\mathcal{R}$) is translated into the rewrite rule $Ri : [P_1] \longrightarrow [P_2]$ in R.

Now, by assuming $\rightarrow^*$ to be the symmetric and transitive clousure of $\rightarrow$, we can prove the following results.

Lemma 3.1 *Given $T_1, T_2 \in \mathcal{T}$, it holds $T_1 \rightarrow T_2 \Longrightarrow [T_1] \longrightarrow [T_2]$.*

Proof. We prove the implication by induction on the derivation of the transition relation of CLS. The base case is when the transition $T_1 \rightarrow T_2$ is derived by applying rule 1 of the semantics of CLS. In this case a CLS rewrite rule $P_1 \mapsto P_2$ has been applied with $P_1\sigma \equiv T_1$ and $P_2\sigma \equiv T_2$. Such a rewrite rule has a corresponding rewriting logic rewrite rule in R that can be applied by means of deduction rule $(r2)$ in order to obtain $[T_1] \longrightarrow [T_2]$. The induction cases are when the transition $T_1 \rightarrow T_2$ is obtained by applying as the last deduction rule either rule 2, 3 or 4 of the CLS semantics. In all these cases an equivalent rewriting logic transition can be derived by applying deduction rule $(r3)$. For instance, if $T_1 \equiv T_1' \mid T_1'' \rightarrow T_2' \mid T_1'' \equiv T_2$, namely rule 2 has been applied as the last one, we have that $(r3)$ can be used with premises $[T_1'] \longrightarrow [T_2']$ (to which the induction hypothesis applies) and $[T_1''] \longrightarrow [T_2'']$ in order to obtain $[T_1] \longrightarrow [T_2]$. □

Proposition 3.2 *Given $T_1, T_2 \in \mathcal{T}$, it holds $T_1 \rightarrow^* T_2 \Longleftrightarrow [T_1] \longrightarrow [T_2]$.*

Proof. The implication from left to right follows from Lemma 3.1 and from deduction rules $(r1)$ and $(r4)$ of rewriting logic that are reflexivity and transitivity rules. The implication from right to left can be proved by induction on the derivation of the transition relation of rewriting logic. The only non-trivial aspect of this proof is that deduction rules ($r2$) and ($r3$) allow the simultaneous application of two rewrite rules in two different positions of the current term. For instance, we might have $[T_1' \mid T_1''] \longrightarrow [T_2' \mid T_2'']$ by applying ($r3$) with premises $[T_1'] \longrightarrow [T_2']$ and $[T_1''] \longrightarrow [T_2'']$ with both $T_1' \not\equiv T_2'$ and $T_1'' \not\equiv T_2''$ (namely, a rule has been applied to T_1' and another one – possibly the same – to T_2'). This problem can be solved by observing that the two rules are applied to different (independent) subterms, hence the simultaneous application can be simulated by a sequence of two rule applications in the semantics of CLS. Hence we have that $[T_1' \mid T_1''] \longrightarrow [T_2' \mid T_2'']$ corresponds to $T_1' \mid T_1'' \to T_2' \mid T_1'' \to T_2' \mid T_2''$, that is $T_1' \mid T_1'' \to^* T_2' \mid T_2''$. □

We remark that the transition relation of rewriting logic contains transitions between non-ground terms (namely patterns in $\mathcal{P}$ in which some variable occurs) that have no corresponding transition in CLS semantics. However, these transitions never arise when an initial ground term is considered.

3.2 Stochastic Simulation Algorithm

The most followed approach to the stochastic simulation of chemical reactions is the one proposed by Gillespie [12]. Gillespie stated the problem as follows: A volume V contains a mixture of N chemical species $S_1, \ldots, S_N$ which can interreact through M chemical reaction channels $(R_1, \ldots, R_M)$. Given the initial numbers of molecules of each species, what will these molecular population levels be at any later time?

The state of the system is represented by a vector $\mathbf{X}(t) = (X_1(t), \cdots, X_N(t))$, where $X_i(t)$ represents the number of S_i molecules in V at time t. Gillespie assumed that for every reaction channel R_μ, there is a constant c_μ such that $c_\mu dt$ is the average probability a particular combination of reactant molecules in R_μ will react accordingly in the next infinitesimal time interval dt. To calculate the probability that a reaction R_μ will occur in V in the next infinitesimal time interval $(t, t+dt)$, we must multiply $c_\mu dt$ by the total number of distinct combinations of R_μ reactant molecules in V at time t.

Given that $\mathbf{X}(t) = \mathbf{x}$, the total number of distinct combinations of reactant molecules in R_μ is denoted with $h_\mu(\mathbf{x})$. Gillespie defined the *propensity function* $a_\mu(\mathbf{x})$ for reaction R_μ as the product of $h_\mu(\mathbf{x})$ and c_μ, such that $a_\mu(\mathbf{x})\ dt$ is the probability that one R_μ reaction will occur in the next infinitesimal time interval $[t, t+dt)$. Propensity functions are used by Gillespie to define the following stochastic algorithm for the simulation of chemical reactions. Let $a_0(\mathbf{x}) = \sum_{i=1}^{M} a_i(\mathbf{x})$, and let τ and μ be random numbers computed as follows:

$$\tau = \frac{1}{a_0(\mathbf{x})} ln(\frac{1}{r_1}) \tag{1}$$

$$\mu = the\ integer\ for\ which \sum_{v=1}^{\mu-1} a_v(\mathbf{x}) < \ r_2 a_0(\mathbf{x}) \leq \sum_{v=1}^{\mu} a_v(\mathbf{x}) \tag{2}$$

where $r_1, r_2 \in [0,1]$ are two real values generated by a random number generator. Gillespie's algorithm is as follows:

Step 0 Input M values representing reaction constants $c_1, \ldots, c_M$, and N values representing initial molecular population numbers $X_1, \ldots, X_N$. Input *total_time* and initialise time variable t to 0.

Step 1 Calculate $a_i(\mathbf{x})$ for $i = 1$ to M. Calculate $a_0(\mathbf{x})$.

Step 2 Generate r_1 and r_2 and calculate τ and μ.

Step 3 Increase t by τ. If $t >$ *total_time* then stop simulation, otherwise execute R_μ, update $X_1, \ldots, X_N$ accordingly and return to **Step 1**.

Time progression in Real-Time Maude is controlled by tick rules. Deterministic tick rules only allow us to set time progression to a constant value. Nondeterministic tick rules give more flexibility by allowing us to set time progression to the value of a variable. This variable is assigned to a value by using a time-sampling strategy, which is chosen by the user at run-time. However, neither deterministic tick rules nor nondeterministic ones are suitable for implementing Gillespie's algorithm in which time progression must be calculated as in Eq. (1). To solve this problem, we add another time variable to Gillespie's algorithm that is always incremented by a fixed amount of time Δt. This new time variable is interpreted by Real-Time Maude (and its analysis tools) as the simulation time. Note that the simulation algorithm is still exact as the new time variable has no influence on the simulation.

For our purpose, we modify Gillespie's algorithm as follows:

Step 0 Input M values representing reaction constants $c_1, \ldots, c_M$, and N values representing initial molecular population numbers $X_1, \ldots, X_N$. Input *total_time* and initialise time variable t to 0.

Step 1 Calculate a_i for $i = 1$ to M. Calculate $\sum_{v=1}^{M} a_v$. Set τ and *delta* to 0. Select a value to initialise *seed*.

Step 2 Generate r_1 and calculate τ according to equation (1). Increase *seed* by 2.

Step 3 While $t + \tau \geq tstep$ do increase *tstep* by Δt. If $tstep > total\_time$ then stop simulation. Otherwise generate r_2 and select μ according to equation (2), increase *seed* by 2 and increase t by τ.

Step 4 Execute R_μ. Update $X_1, \ldots, X_N$ and $a_1, \ldots, a_N$ according to the execution of R_μ.

Step 5 Calculate $\sum_{v=1}^{M} a_v$. Return to **Step 2**.

In this algorithm we use variable *tstep* to record the current time of the simulation. Variable *total_time* is used to limit the duration of simulation. Time is always increased by Δt. Between two time progressions, several reactions may occur. Variable t is used to record the real simulation time according to Gillespie's algorithm. Variable τ is used to record the time lapse until next reaction occurs. In step 3, we compare t (after increased by τ) with *tstep* to check whether the next reaction will occur within this interval, or within the next interval. If the next reaction will not occur within the current interval, *tstep* should be continuously increased by Δt until the correct time interval is found. We use the built-in random number generator from Real Time Maude, which can be used to generate random numbers in [0,1]. Variable *seed* is a parameter of the random number generator. In order to generate distinct random numbers, this variable must be updated after being used to generate a random number.

3.3 Computing the Combinations of Reactants

We have seen that the combinations of reactants play a crucial role in Gillespie's algorithm. We have also seen that the analogous of the combinations of reactants in Stochastic CLS are the different occurrences of the instantiations of a left hand sides of rewrite rules. To count these occurrences we define a function $occ(T, T')$ that gives the number of occurrences of a term T in another term T'. This function will be used to compute the propensity function in our simulation algorithm.

To use the function in our simulation algorithm, we need to define it carefully in order not to make the computation take too much time. Therefore we try to minimise the recursive definition of the function. We start with an idea of grouping parallel composition of similar terms. Let *Grouped Terms GT* and *Base Terms BT* be given by the following grammar:

$$GT \;::=\; \{BT\}N \;\;\Big|\;\; GT \mid GT \qquad\qquad BT \;::=\; \big(GT\big)^L \rfloor\, GT \;\;\Big|\;\; S$$

where S is as defined in Section 2 and N is a natural number greater than 0.

It is easy to see that every CLS term can be represented as a grouped term. For example $\big(a \mid b \cdot b \mid b \cdot b\big)^L \rfloor \big(\big(c \mid c \mid c \mid c\big)^L \rfloor (d \cdot d \mid d \cdot d \mid e)\big)$ can be represented as $\{\big(\{a\}1 \mid \{b \cdot b\}2\big)^L \rfloor (\{\big(\{c\}4\big)^L \rfloor (\{d \cdot d\}2 \mid \{e\}1)\}1\}1$. Actually, more than one grouped representations of a CLS term can be given. In the following we shall assume always a minimal representation in which structurally congruent terms that are composed in parallel are always grouped together. For instance, as the only valid grouped representation of $\big(a\big)^L \rfloor \big(\big(b\big)^L \rfloor (c \mid d) \mid \big(b\big)^L \rfloor (d \mid c)\big)$ we will always consider $\{\big(\{a\}1\big)^L \rfloor (\{\big(\{b\}1\big)^L \rfloor (\{c\}1 \mid \{d\}1)\}2)\}1$ and not $\{\big(\{a\}1\big)^L \rfloor (\{\big(\{b\}1\big)^L \rfloor (\{c\}1 \mid \{d\}1)\}1 | \{\big(\{b\}1\big)^L \rfloor (\{d\}1 \mid \{c\}1)\}1)\}1$ since the two inner loopings are structurally congruent and can be grouped together.

Definition 3.3 Grouped terms subset ($\subseteq$) is defined as follows:

(i) $\{BT\}\, N \;\subseteq\; \{BT\}\, M \mid GT$ if $N \leq M$

(ii) $\{BT\}\, N \mid GT \;\subseteq\; \{BT\}\, M \mid GT_1$ if $N \leq M$ and $GT \subseteq GT_1$

(iii) $GT \;\not\subseteq\; GT_1$ if both (i) and (ii) are not satisfied

Now, by relying on the grouped representation of CLS terms we define a function *occ* that, given two (grouped) terms GT_1 and GT_2 computes the number of occurrences of GT_1 in GT_2 (up to structural congruence).

Definition 3.4 Relation $occ(GT_1, GT_2)$ is recursively defined as follows:

$$occ(GT, \epsilon) = 0 \quad \text{if } GT \not\equiv \epsilon \qquad occ(\epsilon, GT) = 1 \tag{1-2}$$

$$occ(GT_1, GT_2) = occ(GT_1, GT_3) \quad \text{if } GT_2 \equiv GT_3 \tag{3}$$

$$occ(\{BT_1\}M|GT_1, \{BT_2\}N|GT_2) = \binom{N}{M} \times occ'(GT_1, GT_2) + occ(\{BT_1\}M|GT_1, GT_2) \quad \text{if } BT_1 \equiv BT_2 \tag{4}$$

$$\begin{aligned} &occ(\{BT\}M|GT_1, \{(GT_2)^L \rfloor GT_3\}N|GT_4) = \\ &N \times (occ(\{BT\}M|GT_1, GT_2) + occ(\{BT\}M|GT_1, GT_3)) + occ(\{BT\}M|GT_1, GT_4) \\ &\text{if } \{BT\}M \not\subseteq \{(GT_2)^L \rfloor GT_3\}N|GT_4 \end{aligned} \tag{5}$$

$$occ(\{BT\}M|GT_1, GT_2) = 0 \quad \text{if } \{BT\}M \not\subseteq GT_2 \text{ and } \forall GT_3, GT_4, N.\{(GT_3)^L \rfloor GT_4\}N \not\subseteq GT_2 \tag{6}$$

where relation $occ'(GT_1, GT_2)$ is recursively defined as follows:

$$occ'(GT, \epsilon) = 0 \quad \text{if } GT \not\equiv \epsilon \qquad occ'(\epsilon, GT) = 1 \tag{1'-2'}$$

$$occ'(GT_1, GT_2) = occ'(GT_1, GT_3) \quad \text{if } GT_2 \equiv GT_3 \tag{3'}$$

$$occ'(\{BT_1\}M|GT_1, \{BT_2\}N|GT_2) = \binom{N}{M} \times occ'(GT_1, GT_2) \quad \text{if } BT_1 \equiv BT_2 \tag{4'}$$

$$occ'(\{BT\}M|GT_1, GT_2) = 0 \quad \text{if } \{BT\}M \not\subseteq GT_2 \tag{5'}$$

Let us consider a notion of *set of layers* of a term containing the term itself (called first layer) and all its subterms that are operands of some looping and containment operator. For instance, $a \mid (b)^L \rfloor (c)^L \rfloor (d \mid d)$ has itself, b, $(c)^L \rfloor (d \mid d)$, c and $d \mid d$ as layers. Every computation of *occ* consists of two parts. The first part computes the number of occurrences of a term in the same layer. The second part computes the number of occurrences of a term in the inner layers. The function *occ* calls *occ'* to recursively compute the first part, while the second part is computed by recursively calling *occ* itself. We give an example of computation of the *occ* function.

Example 3.5 We compute the number of occurrences of term $a \mid a \mid b$ in term $a \mid a \mid (c)^L \rfloor (a \mid a \mid a \mid b) \mid b \mid b$ as follows:

$$\begin{aligned} & occ(\{a\}2 \mid \{b\}, \{a\}2 \mid \{(\{c\}1)^L \rfloor (\{a\}3 \mid \{b\}1)\}1 \mid \{b\}2) \\ = & \binom{2}{2} \times occ'(\{b\}1, \{(\{c\}1)^L \rfloor (\{a\}3 \mid \{b\}1)\}1 \mid \{b\}2) + occ(\{a\}2 \mid \{b\}1, \{(\{c\}1)^L \rfloor (\{a\}3 \mid \{b\}1)\}1 \mid \{b\}2) \\ = & \binom{2}{2}\binom{2}{1} + occ(\{a\}2 \mid \{b\}1, \{(\{c\}1)^L \rfloor (\{a\}3 \mid \{b\}1)\}1 \mid \{b\}2) \\ = & \binom{2}{2}\binom{2}{1} + occ(\{a\}2 \mid \{b\}1, \{c\}1) + occ(\{a\}2 \mid \{b\}1, \{a\}3 \mid \{b\}1) + +occ(\{a\}2 \mid \{b\}1, \{b\}2) \\ = & \binom{2}{2}\binom{2}{1} + 0 + \binom{3}{2}\binom{1}{1} + 0 = 2 + 3 = 5 \end{aligned}$$

We said that *occ* is a function, but this does not follow immediately from its definition. Let us prove the following result.

Proposition 3.6 *Relation occ is a total function.*

Proof. We first prove that *occ* is total, namely it is defined for all possible pairs of arguments (GT_1, GT_2). Rules (1) and (2) deal with the cases in which one of the two arguments is ϵ, rule (4) with the case in which the first base term of the first argument occurs also in the first layer of the second argument (possibly repeated a different number of times), and rules (5) and (6) with the case in which the first base term of the first argument does not occur in the first layer of the second argument. The difference between (5) and (6) is that in the former the second arguement contains a looping while in the latter it does not. Rule (3) ensures that if the two arguments are in the situation in which rule (4) should apply, then the second argument can be rearranged in such a way that rule (4) becomes applicable. A similar, but simpler, totality proof can be given for occ'.

Now we prove that occ is a function. Given GT_1 and GT_2 we might have more than one ways of computing $occ(GT_1, GT_2)$ mainly for two reasons: (i) if $GT_1 \not\subseteq GT_2$ but $GT_1 = \{BT\}N|GT_1'$ and $GT_2 \equiv \{BT\}M|GT_2'$ with $N < M$ than both rule (4) and one between (5) and (6) can be applied, and (ii) structural congruence applied by means of rule (4) can change the order of the loopings to which rule (5) is applied. We have to prove that in both cases (i) and (ii) all the different ways of computing $occ(GT_1, GT_2)$ lead to the same result. As regards (i), we only have to observe that the binomial coefficient in (4) is equal to zero (by definition) when $N < M$ and that $occ(\{BT_1\}M|GT_1, GT_2)$ essentially searches for occurrences of the first argument in inner layers of GT_2 as done by rule (5). As regards (ii), the result is the same thanks to the commutativity of $+$. Similar considerations can be done on occ'. □

Now, the only problem we have is that CLS rewrite rules may contain variables. (At the moment the occ function is defined only on ground terms.) For the sake of simplicity we restrict here to rewrite rules in which the left hand side is either ground or it has one of the following forms: either $T_1 \mid \big(T_2 \mid X\big)^L \rfloor T_3$, or $T_1 \mid \big(T_2\big)^L \rfloor (T_3 \mid X)$, or $T_1 \mid \big(T_2 \mid X\big)^L \rfloor (T_3 \mid Y)$. Under this restriction the occ function can be extended to handle variables as follows:

$$
\begin{aligned}
&occ(\{(GT_1 \mid X)^L \rfloor GT_2\}1 \mid GT_3, \{(GT_1 \mid GT_4)^L \rfloor GT_2\}N \mid GT_5) = \\
&\qquad N \times occ(GT_1, GT_1 \mid GT_4) \times occ'(GT_3, GT_5) + occ(\{(GT_1 \mid X)^L \rfloor GT_2\}1 \mid GT_3, GT_5) \\
&occ(\{(GT_1)^L \rfloor (GT_2 \mid X)\}1 \mid GT_3, \{(GT_1)^L \rfloor (GT_2 \mid GT_4)\}N \mid GT_5) = \\
&\qquad N \times occ(GT_2, GT_2 \mid GT_4) \times occ'(GT_3, GT_5) + occ(\{(GT_1)^L \rfloor (GT_2 \mid X)\}1 \mid GT_3, GT_5) \\
&occ(\{(GT_1 \mid X)^L \rfloor (GT_2 \mid Y)\}1 \mid GT_3, \{(GT_1 \mid GT_4)^L \rfloor (GT_2 \mid GT_5)\}N \mid GT_6) = \\
&\qquad N \times occ(GT_1, GT_1 \mid GT_4) \times occ(GT_2, GT_2 \mid GT_5) \times occ'(GT_3, GT_6) + \\
&\qquad\qquad occ(\{(GT_1 \mid X)^L \rfloor (GT_2 \mid Y)\}1 \mid GT_3, GT_6)
\end{aligned}
$$

We leave the definition of occ without restrictions on variables as future work.

3.4 Translation of Stochastic CLS terms and rewrite rules

We start by defining the syntax for CLS terms in the following module.

```
(omod CLS is
  pr NAT .
  sorts Elem Seq Term Loop .
  subsorts Elem < Seq < Term .

  op empty : -> Seq [ctor] .
  op _._ : Seq Seq -> Seq [assoc gather (E e) id: empty ctor] .
  op `{_`}_ : Term Nat -> Term .
  op `[_`]LContains`[_`] : Term Term -> Term [prec 41 gather (& &) ctor] .
  op _|_ : Term Term -> Term [assoc comm prec 45 gather (E e) id: empty ctor] .
  .
  .
  .
endom)
```

In this module we define CLS elements, sequences and terms, as well as all related operators. The set of elements is defined as a subset of the set of sequences, and the set of sequences is defined as a subset of the set of terms. We combine the looping and containment operators into one operator, `LContains`. Keywords `assoc` and `comm` are used to define associativity and commutativity of an operator. Keyword `id:` is used to define the identity of an operator. Constructors are denoted by the `ctor` attribute. Keyword `prec` is used to define the precedence of an operator. Keyword `gather` is used to remove ambiguity in parsing, by indicating the precedence of arguments in an operator definition. We give a sequence of as many `E`, `e`, or `&` values as the number of arguments in the operator. An `e` value indicates that the precedence of the argument must be lower than the precedence of the operator. An `E` value indicates that the precedence of the argument must be lower than or equal to the precedence of the operator. To allow any precedence for an argument, we must use `&` in the `gather` attribute.

In module `SCLS` we define data structures needed by the simulation algorithm. The term that models the state of the system is incorporated within class `CLSTerm`. We also define another class, `Admin`, that records all variables in the algorithm. We define the SCLS module as follows:

```
(tomod SCLS is
  inc CLS .
  sorts Propensity Propensities .
  subsort Propensity < Propensities .

  class CLSTerm | term : Term .
  class Admin | seed : Nat, step : Nat, tau : Float, mu : Nat,
                a : Propensities, acum : Propensities, tstep : Float, t : Float .

  .
  .
  .
endom)
```

Attributes `seed`, `step`, `tau`, `mu`, `tstep` and `t` represent variables used by the algorithm. We define `Propensity` as a sort to represent the index and value of a propensity function. The list of all propensity functions are represented as a sort `Propensities`. The attribute `a` represents the list of a_i for $i = 1$ to M, while attribute `acum` represents the list of $\sum_{v=1}^{i} a_v$ for $i = 1$ to M.

We define rewrite rules in another module that imports the SCLS module. In this module we define every chemical species involved in the system as a Maude operator with `Elem` type. We explain this using the *Lotka reactions* [12].

We consider the simple irreversible isomerisation reaction and the Lotka reactions as case studies. The simple irreversible isomerisation reaction is defined as

$$S_1 \xrightarrow{k} S_2 \qquad (3)$$

where $k = 0.5$. The Lotka reactions are defined as follows:

$$S_1 \xrightarrow{k_1} S_1|S_1 \qquad S_1|S_2 \xrightarrow{k_2} S_2|S_2 \qquad S_2 \xrightarrow{k_3} \epsilon \qquad (4)$$

where $k_1 = 10, k_2 = 0.01$ and $k_3 = 10$.

For each case study we define a module that imports the previously defined CLS and SCLS modules. Then we implement step 0 and step 1 of the algorithm by defining equations and rewrite rules that initialise the system as follows.

```
ops S1 S2 : -> Elem .
ops lotka Adm : -> Oid .
op INIT : Term -> GlobalSystem .
op ReactionNum : -> Nat .

eq ReactionNum = 3 .
eq INIT(T) =
    { < lotka : CLSTerm | term : T >
      < Adm : Admin | seed : 0, t : 0.0, step : 1, tstep : 0.0,
          a : ([1 (10 * occ({ S1 } 1,T))] [2 (occ({ S1 } 1 | { S2 } 1,T) / 100)]
               [3 (10 * occ({ S2 } 1,T))]),
                      acumm : ([1 0] [2 0] [3 0]), tau : 0.0, mu : 0 >
      } .
rl [ initialise1 ] :
   < Adm : Admin | step : 1, a : P, acumm : P' >
=>
   < Adm : Admin | seed : random(1), step : 2, a : P, acumm : sum(P',P,1,ReactionNum) > .
.
.
.
rl [ initialise100 ] :
   < Adm : Admin | step : 1, a : P, acum : P' >
=>
   < Adm : Admin | seed : random(100), step : 2, a : P, acum : sum(P',P,1,ReactionNum) > .
```

The operators lotka and Adm represent objects instantiated from classes CLSTerm and Admin. The operator ReactionNum represents the number of reaction channels M. The occ function is used to define the propensity of each reaction. The rules initialise1 to initialise100 perform two things: calculating the cumulative propensity acum using function sum (which is defined in module SCLS), and initialising the random number generator with 100 distinct numbers. It models nondeterminism in the system by allowing the simulation to run in 100 different behaviours.

We notice from the above rewrite rules that in object-oriented modules, only relevant attributes are shown in the left-hand side of the rules. In the right-hand side of the rules, only attributes whose values are changed are needed.

Step 2 of the algorithm can be defined using the following rewrite rule.

```
rl [ calculate-tau ] :
   < Adm : Admin | step : 2, acum : (P [ReactionNum F1]), seed : M >
=>
   < Adm : Admin | step : 3, tau : ((- log(float(rand(M)))) / float(F1)), seed : (M + 2) > .
```

Step 3 of the algorithm can be defined using the following rewrite rules.

```
crl [ select-mu ] :
   < Adm : Admin | step : 3, acum : (P [ReactionNum F1]), tau : F'',
                   tstep : F', seed : M, t : F >
=>
   < Adm : Admin | step : 4, mu : findmu(rand(M) * F1,P [3 F1],3),
                   seed : (M + 2), t : (F'' + F) >
if F + F'' < F' .

crl [ tick ] :
   { < Adm : Admin | step : 3, tau : F'', tstep : F', t : F, acum : (P [3 F1]), seed : M >
     C:Configuration }
   =>
   { < Adm : Admin | step : 4, tstep : (F' + float(R)), seed : (M + 2), t : (F'' + F)
                     mu : findmu(rand(M) * F1,P [ReactionNum F1],ReactionNum) >
     C:Configuration} in time R
if F + F'' >= F' [nonexec] .
```

Function `findmu(R,P,M)` is a function that chooses the next reaction (μ) in accordance with equation (2). The second rule is not directly executable in MAUDE, since variable `R` is not yet assigned to any value. Real-Time MAUDE allows the user to choose a time-sampling strategy (or tick mode) for instantiating variable `R` in each tick rule application. In the following we choose the default mode to instantiate variable `R` with 1/100.

```
(set tick def 1/100 .)
```

The execution of Lotka reactions (step 4 of the algorithm) can be defined as follows:

```
crl [ S1 ] :
    < O : CLSTerm | term : (T | { S1 } i) >
    < Adm : Admin | a : ([1 F1] [2 F2] P), mu : 1, step : 4 >
  =>
    < O : CLSTerm | term : (T | { S1 } (i + 1)) >
    < Adm : Admin | a : ([1 (occ({ S1 } 1,T | { S1 } (i + 1)) * 10)]
                          [2 (occ({ S1 } 1 | { S2 } 1,T | { S1 } (i + 1)) / 100)]
                          P),
                    step : 5 >
if i > 0 .

crl [ S2 ] :
    < O : CLSTerm | term : (T | { S1 } i | { S2 } M) >
    < Adm : Admin | mu : 2, step : 4 >
  =>
    < O : CLSTerm | term : (T | { S1 } sd(i,1) | { S2 } (M + 1)) >
    < Adm : Admin | a : ([1 (occ({ S1 } 1,T | { S1 } sd(i,1) | { S2 } (M + 1)) * 10)]
                 [2 (occ({ S1 } 1 | { S2 } 1,T | { S1 } sd(i,1) | { S2 } (M + 1)) / 100)]
                          [3 (occ({ S2 } 1,T | { S1 } sd(i,1) | { S2 } (M + 1)) * 10)]),
                    step : 5 >
if i > 0 /\ M > 0 .

crl [ S3 ] :
    < O : CLSTerm | term : (T | { S2 } i) >
    < Adm : Admin | a : ([2 F2] [3 F3] P), mu : 3, step : 4 >
  =>
    < O : CLSTerm | term : T | { S2 } sd(i,1)>
    < Adm : Admin | a : ([2 (occ({ S1 } 1 | { S2 } 1,T | { S2 } sd(i,1)) / 100)]
                          [3 (occ({ S2 } 1,T | { S2 } sd(i,1)) *  10)] P),
                    step : 5 >
if i > 0 .
```

where `sd(i,1)` is equal to $i-1$. Again here we use the `occ` function to calculate the propensity of each reaction. To optimise the performance of the simulation, we modify the algorithm such that in every application of a rule (that represents a reaction occurs) we only need to recalculate the propensity of reactions that have been modified by the application of the rule.

Step 5 of the algorithm is defined as follows:

```
rl [ summing-propensities ] :
   < Adm : Admin | a : P', acumm : P, step : 5 >
 =>
   < Adm : Admin | a : P', acumm : sum(P,P',1,3), step : 2 > .
```

where `sum(acum,a,1,M)` is a function to modify `acumm` based on new values in `a`.

4 Applications

In this section we show the application of Real-Time Maude analysis tools to models obtained from the translation of Stochastic CLS models.

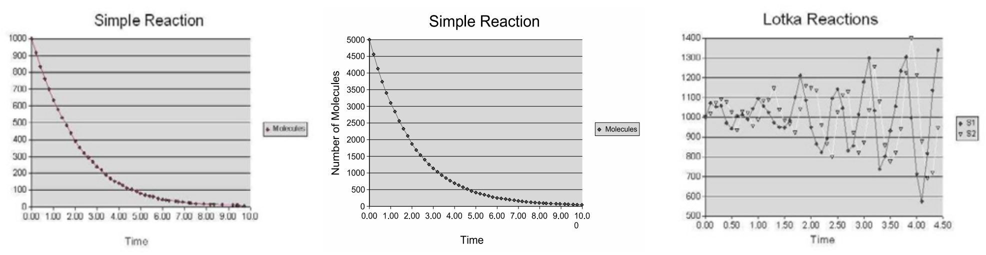

Fig. 1. Simple irreversible isomerisation reaction with 1000, 5000 molecules and the Lotka reactions with 1000 molecules

4.1 Lotka reactions

The first way to analyse reactions using our approach is to run a simulation of their Maude model. Maude has no features to visualise the result of a simulation. Therefore we try to plot charts representing simulation results, and compare our chart to similar charts from previous works (see [12]).

The following command shows a simulation of the Lotka reactions with 100 molecules of S_1 and 100 molecules of S_2 in $1/10$ time units. The simulation is performed using $1/100$ as a tick value.

```
(set tick max def 1/100 .)
(tfrew INIT({S1} 100 |{S2} 100) in time <= 1/10 .)
rewrites: 1000656 in 6094650579ms cpu (97927ms real) (0 rewrites/second)

Timed fair rewrite  INIT({S1}100 |{S2}100)in MODEL-CHECK-LOTKA with mode
    maximal time increase with default 1/100 in time <= 1/10

Result ClockedSystem :
  {< Adm : Admin | a :([1 2480]([2 2976/25][3 480](),acumm :([1 2480]([2
    64976/25][3 76976/25](),t : 9.9892214323839157e-2,
    .
    .
    .
    in time 1/10
```

The above simulation shows that within $1/10$ time units (e.g. after 0.0999 time units) the Lotka reactions will stop because the next tick rule execution will increase time so that it is greater than the time limit ($1/10$ time units). Using attribute `a`, we can calculate the number of molecules of each reactant that are present at the end of the simulation.

Figure 1 shows simulation results for the simple irreversible isomerisation reaction and the Lotka reactions. It shows that our model behaves similarly to Gillespie's one [12].

By using the `search` command, we can check all possible behaviours of the system. We have defined 100 rules to initialise the random number generator with 100 distinct random numbers. This allows Maude engine to explore a state space with 100 different behaviours. Although this approach cannot cover all possible behaviours of the system, it yields a significant sample of behaviours. The following example shows the use of a search command initialised with 4 molecules of S_1 and 4 molecules of S_2, and limited to the first 10 states where no more occurrences of S_2 are available in the system.

```
(tsearch [10] INIT({S1} 4 | {S2} 4) =>* {< O:Oid : CLSTerm | term : T:Term > C:Configuration}
such that occ({ S2 } 1,T:Term) = 0 in time <= 1/10 .)
```

```
Timed search [10] in MODEL-CHECK-LOTKA
    INIT({S1}4 |{S2}4)=>* {< O:Oid : CLSTerm | term : T:Term > C:Configuration}
in time <= 1/10 and with mode maximal time increase with default 1/100 :

Solution 1
C:Configuration --> < Adm : Admin | a :([1 50]([2 0][3 0](),acumm :([1 50]([2
    1001/20][3 1201/20](),t : 7.8293318117206676e-2,
    .
    .
    .
Solution 10
C:Configuration --> < Adm : Admin | a :([1 80]([2 0][3 0](),acumm :([1 80]([2
    80][3 80](),t : 5.6307289864345335e-2
```

Another interesting search command is the `find earliest` command, which searches for the earliest time when a given state is reached. The following example shows that the earliest time `S2` vanishes from a system initialised with 4 molecules of S_1 and 4 molecules of S_2 occurs within 3/50 time units (e.g. after 0.0563 time units).

```
(find earliest INIT({S1} 4 | {S2} 4) =>* {< O:Oid : CLSTerm | term : T:Term >
    C:Configuration} such that occ({ S2 } 1,T:Term) == 0 .)

Find earliest {< O:Oid : CLSTerm | term : T:Term > C:Configuration} in MODEL-CHECK-LOTKA
    such that INIT({S1}4 |{S2}4)=>* {< O:Oid : CLSTerm | term : T:Term > C:Configuration}
with mode maximal time increase with default 1/100 :

Result: {< Adm : Admin | a :([1 80]([2 0][3 0](),acumm :([1 80]([2 2002/25][3 2252/25](),
    t : 5.6307289864345335e-2,mu : 3,seed : 1646868826,step : 5,tstep : 6.0e-2,
    tau : 1.3291670449923896e-3 > < lotka : CLSTerm | term :({S1}8)>} in time 3/50
```

To perform model-checking we define another module as follows:

```
(tomod MODEL-CHECK-LOTKA is
   inc TIMED-MODEL-CHECKER .
   pr LOTKA-INIT .

   op vanished : Term -> Prop .
   op IsLessThan : Term Term -> Prop .

   eq { < O : CLSTerm | term : T' > C} |= vanished(T) = (occ(T,T') == 0) .
   eq { < O : CLSTerm | term : T'' > C} |= IsLessThan(T,T') = (occ(T,T'') < occ(T',T'')) .

   .
   .
   .
endtom)
```

In the above module, we define some properties of the system, using our `occ` function:

vanished(T) indicates that term T has vanished from the system,

IsLessThan(T,T') indicates that the number of occurences of term T in the system behaviour is less than the number of occurences of T'.

We give two examples of model-checking for Lotka reactions with 4 molecules of S_1 and 4 molecules of S_2 as initial states. The first example shows that `S2` will eventually vanish from the system in 1 time unit. The second example shows that the amount of `S2` will become eventually less than the amount of `S1` in the system in 1 time unit.

```
(mc INIT({S1} 4 | {S2} 4) |=t <> vanished({ S2 } 1) in time <= 1 .)

Model check INIT({S1}4 |{S2}4) |=t <> vanished({S2}1)in MODEL-CHECK-LOTKA in time
    <= 1 with mode maximal time increase with default 1/100

Result Bool :
  true
```

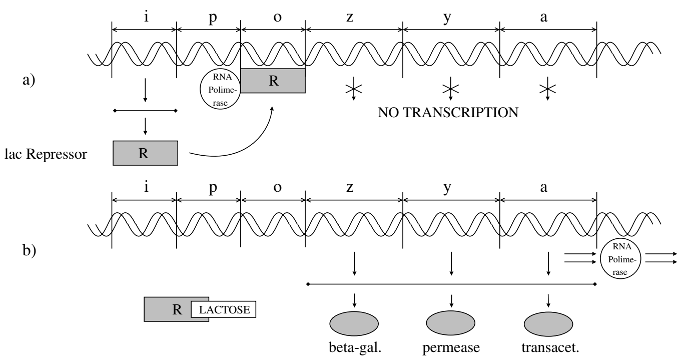

Fig. 2. The regulation process in the Lac Operon.

```
(mc  INIT({S1} 4 | {S2} 4) |=t <> IsLessThan({ S2 } 1,{ S1 } 1) in time <= 1 .)

Model check INIT({S1}4 |{S2}4) |=t <> IsLessThan({S2}1,{S1}1)in MODEL-CHECK-LOTKA in time
    <= 1 with mode maximal time increase with default 1/100

Result Bool :
  true
```

Maude uses *Linear Temporal Logic* (LTL) as the logic to express properties in model checking. The `<> p` formula is an LTL formula that means *eventually* property p will hold in the system.

4.2 The lactose operon

The lactose operon is a sequence of six genes of the E.coli bacterium that are responsible for producing three enzymes for lactose degradation, namely the *lactose permease*, which is incorporated in the membrane of the bacterium and actively transports the sugar into the cell, the *beta galactosidase*, which splits lactose into glucose and galactose, and the *transacetylase*, whose role is marginal. The first three genes of the operon (i,p,o) regulate the production of the enzymes, and the last three (z,y,a), called *structural genes*, are transcribed (when allowed) into the mRNA for beta galactosidase, lactose permease and transacetylase, respectively.

The regulation process is as follows (see Figure 2): gene i encodes the *lac Repressor*, which, in the absence of lactose, binds to gene o (the *operator*). Transcription of structural genes into mRNA is performed by the RNA polymerase enzyme, which usually binds to gene p (the *promoter*) and scans the operon from left to right by transcribing the three structural genes z, y and a into a single mRNA fragment. When the lac Repressor is bound to gene o, it becomes an obstacle for the RNA polymerase, and transcription of the structural genes is not performed. On the other hand, when lactose is present inside the bacterium, it binds to the Repressor and this cannot stop anymore the activity of the RNA polymerase. In this case the transcription is performed and the enzymes for lactose degradation are synthesized.

In Stochastic CLS we can model the membrane of the bacterium as the looping $(m)^L$, where the alphabet symbol m generically denotes the whole membrane surface in normal conditions. Moreover, we model the lactose operon as the sequence $lacI \cdot lacP \cdot lacO \cdot lacZYA$ ($lacI{-}A$ for short), in which each symbol corresponds

$$
\begin{aligned}
lacI\text{–}A &\overset{0.02}{\longmapsto} lacI\text{–}A \mid Irna && \text{(R1)}\\
Irna &\overset{0.1}{\longmapsto} Irna \mid repr && \text{(R2)}\\
polym \mid lacI \cdot lacP \cdot x \cdot lacZYA &\overset{0.1}{\longmapsto} lacI \cdot PP \cdot x \cdot lacZYA && \text{(R3)}\\
lacI \cdot PP \cdot x \cdot lacZYA &\overset{0.01}{\longmapsto} polym \mid lacI \cdot lacP \cdot x \cdot lacZYA && \text{(R4)}\\
lacI \cdot PP \cdot lacO \cdot lacZYA &\overset{20.0}{\longmapsto} polym \mid Rna \mid lacI\text{–}A && \text{(R5)}\\
Rna &\overset{0.1}{\longmapsto} Rna \mid betagal \mid perm \mid transac && \text{(R6)}\\
repr \mid lacI \cdot y \cdot lacO \cdot lacZYA &\overset{1.0}{\longmapsto} lacI \cdot y \cdot RO \cdot lacZYA && \text{(R7)}\\
lacI \cdot y \cdot RO \cdot lacZYA &\overset{0.01}{\longmapsto} repr \mid lacI \cdot y \cdot lacO \cdot lacZYA && \text{(R8)}\\
repr \mid LACT &\overset{0.005}{\longmapsto} RLACT && \text{(R9)}\\
RLACT &\overset{0.1}{\longmapsto} repr \mid LACT && \text{(R10)}\\
\big(X\big)^{L} \rfloor (perm \mid Y) &\overset{0.1}{\mapsto} \big(perm \mid X\big)^{L} \rfloor Y && \text{(R11)}\\
LACT \mid \big(perm \mid X\big)^{L} \rfloor Y &\overset{0.001}{\mapsto} \big(perm \mid X\big)^{L} \rfloor (LACT|Y) && \text{(R12)}\\
betagal \mid LACT &\overset{0.001}{\mapsto} betagal \mid GLU \mid GAL && \text{(R13)}\\
perm \overset{0.001}{\mapsto} \epsilon \qquad Irna &\overset{0.001}{\mapsto} \epsilon \qquad transac \overset{0.001}{\mapsto} \epsilon && \text{(R14-R16)}\\
repr \overset{0.002}{\mapsto} \epsilon \qquad betagal &\overset{0.01}{\mapsto} \epsilon \qquad Rna \overset{0.01}{\mapsto} \epsilon && \text{(R17-R19)}\\
RLACT &\overset{0.002}{\mapsto} LACT && \text{(R20)}
\end{aligned}
$$

Fig. 3. Rewrite rules of the Stochastic CLS model of the lactose operon.

to a gene (apart from the last three genes that are grouped together in the symbol $lacZYA$). We replace $lacO$ with RO in the sequence when the lac Repressor is bound to gene o, and $lacP$ with PP when the RNA polymerase is bound to gene p. When the lac Repressor and the RNA polymerase are unbound, they are modeled by the symbols $repr$ and $polym$, respectively. We model the mRNA of the lac Repressor as the symbol $Irna$, a molecule of lactose as the symbol $LACT$, and beta galactosidase, lactose permease and transacetylase enzymes as symbols $betagal, perm$ and $transac$, respectively. Finally, since the three structural genes are transcribed into a single mRNA fragment, we model such mRNA as a single symbol Rna.

The initial state of the bacterium when no lactose is present in the environment and when 100 molecules of lactose are present are modeled by the following terms (where $n \times T$ stands for a parallel composition $T \mid \ldots \mid T$ of length n):

$$
\begin{aligned}
Ecoli \; &::= \; \big(m\big)^{L} \rfloor (lacI\text{–}A \mid 30 \times polym \mid 1 \times repr) && (5)\\
EcoliLact \; &::= \; Ecoli \mid 100 \times LACT && (6)
\end{aligned}
$$

The dynamics of the system is modeled by the rules in Figure 3, where x can be either $lacO$ or RO and y either $lacP$ or PP. A rule with one of these placeholders can be implemented in Maude either by writing two different rules or by using conditional rules.

Rules (R1) and (R2) describe the transcription and translation of gene i into the lac Repressor. Rules (R3) and (R4) describe binding and unbinding of the RNA polymerase to gene p. Rules (R5) and (R6) describe the transcription and translation of the three structural genes. Transcription of such genes can be performed only when the sequence contains $lacO$ instead of RO, that is when the lac Repressor is not bound to gene o. Rules (R7) and (R8) describe binding and unbinding of the lac Repressor to gene o. Finally, rules (R9) and (R10) describe the binding and unbinding, respectively, of the lactose to the lac Repressor.

Rule (R11) describes the incorporation of the lactose permease in the membrane of the bacterium, rule (R12) the transportation of lactose from the environment to the interior performed by the lactose permease, and rule (R13) the decomposition of the lactose into glucose (denoted GLU) and galactose (denoted GAL) performed by the beta galactosidase. Finally, rules from (R14) to (R20) describe the degradation of all the proteins and pieces of mRNA involved in the process:

We translate the Stochastic CLS model into Maude and analyse it. The simulation runs quite fast and produces similar behaviour as the result in the work by Barbuti, Carvagna, Maggiolo-Schettini, Milazzo and Pardini [6]. Our intention in this paper is to show that we can also perform not only chart-based analysis, but can also perform analysis on some logical properties of the system. We show that by analysing two properties of this case study.

The first property is related with the amount of enzymes (*beta galactosidase* and *lactose permease*) in the absence of lactose in the environment. The amount of such enzymes (in number of molecules) should always below some limit. Here we show that our model satisfies this property, with 20 as the limit. We use the Maude search command to check whether there is a state where the number of *beta galactosidase* or *lactose permease* is greater than 20. The Maude engine shows `No solution` as the answer, which means that there is no such state.

```
(tsearch INIT({ [{ m } 1] LContains [{ laci . lacp . laco . lacz . lacy . laca } 1 |
  { polym } 30 | { repr } 1] } 1) =>* {< O:Oid : CLSTerm | term : T:Term > C:Configuration}
  such that occ({ perm } 1,T:Term) > 20 or occ({ betagal } 1,T:Term) > 20 in time <= 1500 .)

Timed search in MODEL-CHECK-LOTKA
    INIT({[{m}1]LContains[{laci . lacp . laco . lacz . lacy . laca}1 |{
    polym}30 |{repr}1]}1)=>* {< O:Oid : CLSTerm | term : T:Term >
    C:Configuration}
in time <= 1500 and with mode maximal time increase with default 1 :

No solution
```

The second property is related with the amount of of enzymes *beta galactosidase* and *lactose permease* in the presence of lactose. We want to show that with the presence of lactose, the number of such enzymes will eventually be greater than 20. Now we use the Maude model check command to verify this property. The result shows that this property holds in our system.

```
((mc  INIT({ [{ m } 1] LContains [{ laci . lacp . laco . lacz . lacy . laca } 1 |{ polym } 30 |
  { repr } 1] } 1 | { LACT } 100) |=t (<> IsGreaterThanN({ betagal } 1,20)) /\
  (<> IsGreaterThanN({ perm } 1,20)) in time <= 1500 .)

Model check INIT({[{m}1]LContains[{laci . lacp . laco . lacz . lacy . laca}1 |{
    polym}30 |{repr}1]}1 |{LACT}100) |=t <> IsGreaterThanN({betagal}1,20)/\ <>
    IsGreaterThanN({perm}1,20)in MODEL-CHECK-LOTKA in time <= 1500 with mode
    maximal time increase with default 1

Result Bool :
  true
```

5 Conclusions and Future Work

We have proposed an approach to study Stochastic CLS specifications of biological systems by making use of Real-Time Maude. Our approach can be used to analyse a biological system not only by observing the chart representing the simulation result, but also by means of logical formulae. We show the applicability of our approach by verifying two properties of the lac operon model.

For our future work, we are interested to explore more about probabilistic model checking. In this paper we analyse properties that can only have boolean values. It will be interesting to extend the language to support answering queries with numeric values, such as probability of an event occur in a period of time. Currently only queries related with time can be answered, for instance finding the earliest time an event occurs.

As we see in Section 3.3, the definition of *occ* still has restriction on the use of variables. In the future we are interested to define *occ* without any restriction on the variables used.

References

[1] http://osl.cs.uiuc.edu/~{}ksen/vesta2/.

[2] http://www.dcs.ed.ac.uk/pepa/.

[3] Gul Agha, José Meseguer, and Koushik Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electric Notes in Theoretical Computer Science*, 153:213 – 239, 2006.

[4] O. Andrei, G. Ciobanu, and D. Lucanu. A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, 373:163–181, 2007.

[5] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, P. Tiberi, and A. Troina. Stochastic cls for the modeling and simulation of biological systems. *Transactions on Computational Systems Biology*, 2008. in press.

[6] Roberto Barbuti, Giulio Caravagna, Andrea Maggiolo-Schettini, Paolo Milazzo, and Giovanni Pardini. The Calculus of Looping Sequences. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *Formal Methods for Computational Systems Biology, LNCS 5016*, pages 387–423. Springer, 2008.

[7] Muffy Calder, Stephen Gilmore, and Jane Hillston. Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. In Anna Ingolfsdottir and Hanne Riis Nielson, editors, *Proceedings of the BioConcur Workshop on Concurrent Models in Molecular Biology*, August 2004.

[8] Luca Cardelli. Brane Calculi - interactions of biological membranes. In V.Danos and V.Schachter, editors, *Computational Methods in Systems Biology. International Conference CMSB 2004, LNCS 3082*, pages 257 – 280. Springer, 2005.

[9] Nathalie Chabrier-Rivier, Marc Chiaverini, Vincent Danos, François Fages, and Vincent Schachter. Modeling and querying biomolecular interaction networks. *Theoretical Computer Science*, 325(1):25–44, September 2004.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.3)*, July 2007.

[11] V. Danos and C. Laneve. Formal molecular biology. *Theoretical Computer Science*, 325:69–110, 2004.

[12] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340 – 2361, 1977.

[13] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 391:239–257, 2008.

[14] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: a hybrid approach. *Int. Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[15] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In *Electronic Notes in Theoretical Computer Science*, volume 4, pages 190–225. Elsevier, 1996.

[16] Peter C. Ölveczky and Stian Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In *Proc. 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 07), LNCS 4468*, 2007.

[17] Peter Csaba Ölveczky. *Real-Time Maude 2.3 Manual*, August 2007.

[18] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Bounded probabilistic model checking with the murphi verifier. In *Formal Methods in Computer-Aided Design (FMCAD'04), LNCS 3312*, pages 15–17. Springer, 2004.

[19] M.J. Pérez-Jiménez and F.J. Romero-Campero. P systems, a new computational modelling tool for systems biology. In *Transactions on Computational Systems Biology VI*, volume 4220 of *LNBI*, pages 176–197. Springer, 2006.

[20] C. Priami, Aviv Regev, W. Silverman, and Ehud Shapiro. Application of a stochastic name passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.

[21] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On Statistical Model Checking of Stochastic Systems. In Kousha Etessami and Sriram K. Rajamani, editors, *17th International Conference on Computer Aided Verification (CAV'05), LNCS 3576*, pages 266–280. Springer, 2005.