Scientific Annals of Computer Science vol. 21, 2011, pp. 39–72

Modular Verification of Interactive Systems with an Application to Biology

Peter DRÁBIK[1], Andrea MAGGIOLO-SCHETTINI[1], Paolo MILAZZO[1]

Abstract

We propose sync-programs, an automata-based formalism for the description of biological systems, and a modular verification technique for such a formalism that allows properties expressed in the universal fragment of CTL to be verified on suitably chosen fragments of models, rather than on whole models. As an application we show the modelling of the *lac* operon regulation process and the modular verification of some properties. Verification of properties is performed by using the NuSMV model checker and we show that by applying our modular verification technique we can verify properties in shorter times than those necessary to verify the same properties in the whole model.

1 Introduction

Many formalisms originally developed by computer scientists to model systems of interacting components have been applied to biology, also with extensions to allow more precise descriptions of the biological behaviours [4, 8, 12, 16, 32, 33]. Model checking permits the verification of properties of a system (expressed as logical formulas) by exploring all the possible behaviours of the system. It has been successfully applied to analysis of biological systems. Examples of well-established formal frameworks that can be used to model, simulate and model check descriptions of biological systems are [24, 22, 12].

[1]Dipartimento di Informatica, Largo B. Pontecorvo 3, 56127 Pisa, Italy.
Email: {drabik, maggiolo, milazzo}@di.unipi.it

Model checking techniques have traditionally suffered from the state explosion problem. Standard approaches to the solution of this problem are based on abstractions and similar model reduction techniques (see e.g. [14]). Moreover, the use of Binary Decision Diagrams (BDDs) to represent the state spaces (symbolic model checking) often allows the tractable size of models to be significantly increased [6].

Another method for trying to avoid the state explosion problem exploits the natural decomposition of the system. The goal is to verify properties of individual components and infer that these hold in the complete system. This is the approach that we follow in this paper, and it can be particularly efficient when the modelled systems consist of a high number of components, whereas properties of interest deal only with a rather small subset of them (as often happens with biological systems).

A class of properties whose satisfaction is preserved from the components to the complete system was identified in Grumberg and Long [23] as ACTL, the universal fragment of CTL temporal logic. A technique proposed by Attie [2] exploits the preservation of these properties in order to verify concurrent programs and synthesise systems from specifications. Attie uses a formalism called synchronisation skeletons [13], an abstraction of sequential processes, suitable for describing distributed systems. The synchronisation skeletons are state-machines where states are connected by edges representing conditional transitions. The transitions can be conditioned by the states of other synchronisation skeletons.

In this paper we propose an extension of Attie's approach and application of the modular verification technique to systems biology. However, synchronisation skeletons are not suitable for modelling biological systems because of their interleaving nature. In fact, in synchronisation skeletons a process may perform an autonomous transition by looking at other processes states.

We define sync-programs, an automata-based formalism of interactive systems which extends Attie's approach by allowing processes to perform transitions simultaneously. Actually, we consider a very general form of synchronisation that allows components of a sync-program to perform a transition either autonomously, or by synchronising with another component, or by synchronising with more than one other components. This permits a wide range of interactions between biological entities to be suitably described.

To be able to apply the proposed modular verification technique, the

systems under consideration are subject to some restrictions. In particular, we assume infinite behaviours over a finite number of states and fairness of systems. The fairness condition consists of requiring that each component of the system contributes to the overall behaviour with infinitely many transitions.

As an example of modelling of biological systems and of modular verification of some properties we apply our formalism to the well-known biological process of *lac* operon gene regulation. We exploit the NuSMV model checker [10] to verify properties on this model by applying our modular verification technique. This requires translating the model into the input language of the verification tool. In the considered example, verifying properties by using our modular verification technique, namely on a suitable portion of the model, takes much less time than verifying the same properties on the whole model.

The rest of this paper is organised as follows. In Section 2 we define the syntax and the semantics of sync-programs. In Section 3 we define and prove the correctness of our modular verification technique. In Section 4 we give the model of the *lac* operon gene regulation process. In Section 5 we describe the translation of the *lac* operon model into the input language of NuSMV and show some results of verification of properties. Finally, in Section 6 we draw our conclusions and discuss related work and further developments.

This paper is an extended and revised version of [20]. In [20] properties to be verified on the *lac* operon model were only discussed. On the contrary, in the present paper we give a translation of the model into the NuSMV input language, we show results of verification of the properties by using the tool and we compare the time necessary to verify such properties by using our modular verification approach with respect to the time of verifying the same properties on the whole model.

2 Sync-programs

In this section we define the syntax and the semantics of the sync-programs, namely Attie's synchronisation skeletons extended with synchronisation.

2.1 Syntax

To model biological systems, we use a component-based approach. Each component represents a biological entity, e.g. a protein or an enzyme.

We assume a finite *index set* I, where an index i represents a unique identifier of a component. By $I(i)$ we denote the set $I - \{i\}$. With a component with index i a set AP_i of *atomic propositions* is associated, which encode the state of the component. The sets of atomic propositions are pairwise disjoint for all the components, i.e. if $i \neq j$ then $AP_i \cap AP_j = \emptyset$.

A component is modelled by using a finite state machine called a sync-automaton.

Definition 1 *A* sync-automaton P_i^I*, where i is a component index from index set I, is a tuple (S_i, S_i^0, R_i):*

- $S_i \subseteq \mathcal{P}(AP_i)$ *is the set of* states*;*
- $S_i^0 \subseteq S_i$ *is the set of* initial states*;*
- $R_i \subseteq S_i \times SC_i \times S_i$*, where $SC_i \subseteq \mathcal{P}(\bigcup_{j \in I}(\mathcal{P}(AP_j) \times \mathcal{P}(AP_j)))$, are labelled* moves *between states.*

Each state of a sync-automaton P_i^I is a truth value assignment to atomic propositions of component C_i. We denote a move from state s_i to state t_i with a label c by $s_i \xrightarrow{c} t_i$. The move from state s_i to t_i with label c intuitively means that automaton P_i^I can move from s_i to t_i if the activities of automata in $I(i)$ satisfy condition c called a synchronisation condition.

Definition 2 *A* synchronisation condition *of sync-automaton P_i^I is a label of the form $\{A_j{:}B_j \mid j \in L\}$, where $L \subseteq I(i)$ and A_j, B_j are sets of atomic propositions drawn from AP_j or their negations.*
$\{A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}\}$ where $\{j_1, \ldots, j_n\} \subseteq I(i)$ and A_j, B_j are sets of atomic propositions drawn from AP_j or their negations.

We denote a synchronisation condition $\{A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}\}$ as a formula $\wedge_{j \in L} A_j{:}B_j$, where $L = \{j_1, \ldots, j_n\}$ and A_j, B_j are conjunctions of atomic propositions from AP_j. The set $L \subseteq I(i)$ contains indices of the sync-automata with which P_i^I wants to synchronise. For every j in L, the sets of propositions A_j and B_j are to be satisfied in the starting and ending state, respectively, of the concurrently performed move of P_j^I. In other words, $A_j{:}B_j$ in a label of a move of P_i^I says that every move in P_j^I that can be performed in parallel with this move of P_i^I is obliged to lead from a state satisfying A_j to a state satisfying B_j. It is worth mentioning that A_j and B_j need not be full descriptions of states, they can be partial and

thus be satisfied by more than one state. In order for the synchronisation of several sync-automata to actually take place, we will require that the move performed by each sync-automaton refers in its synchronisation condition to every other participating sync-automaton.

Note that it is possible for L to be empty. Intuitively, this means that the considered move of sync-automaton P_i^I does not have any requirements on other sync-automata. We write a synchronisation condition of this form, i.e. $\wedge_{j\in\emptyset} A_j{:}B_j$, as $NOSYNC$. Move $s_i \xrightarrow{NOSYNC} t_i$ represents an autonomous move of P_i^I i.e. the sync-automaton moves without performing synchronisation.

In the special case that set A_j is empty, we shall write $true_j{:}B_j$. In this case the sync-automaton is willing to synchronise with any move of P_j^I satisfying B_j in the ending state. Symmetrically, if B_j is empty, A_j needs to be "matched" in the starting state, and we write $A_j{:}true_j$. If both A_j and B_j are empty, the sync-automaton is ready to participate in synchronisation with any move of P_j^I and we write this condition as $true_j{:}true_j$. For the sake of simplicity we will always write $true{:}B_j$ and $A_j{:}true$ for $true_j{:}B_j$ and $A_j{:}true_j$, respectively. We cannot do the same simplification on $true_j : true_j$, namely we cannot remove the two subscripts j, because this would make it impossible to understand which is the sync-automaton the synchronisation condition refers to.

Moreover, note that multiple moves between the same pair of states are possible. Loops are covered by the definition as well, and we use an abbreviation $A_j \circlearrowleft$ for a condition of the form $A_j{:}A_j$.

On fig. 1 we show an example of a sync-automaton, denoted P_{lac}^I where I is the index set including indices Lac, $Beta$, Glu and $Allo$. It is the sync-automaton that we will use to describe lactose in the lac operon model that we will give in Section 4. The sync-automaton has two states. For each state we display only the atomic propositions true in that state. There is a $NOSYNC$ move between the two states representing an autonomous state change performed by lactose. Moreover, there are two looping moves in the state Lac_in, each representing synchronisation with two other sync-automata. In particular, $Beta\_high \circlearrowleft$ and $Beta\_low \circlearrowleft$ are conditions for synchronisation with synch-automaton P_β^I modelling β-galactosidase, $true : Glu\_high$ and $Glu\_low \circlearrowleft$ are conditions for synchronisation with synch-automaton P_{glu}^I modelling glucose, and $true : Allo\_low$ is a condition for synchronisation with synch-automaton P_{allo}^I modelling allolactose. The inclusion of the looping move in the state Lac_out with five other sync-

automata will be explained later in Section 4.

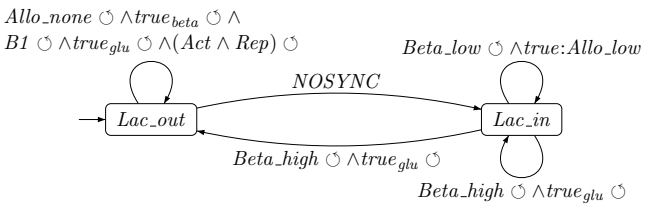

Figure 1: P^I_{lac} – Lactose

Having sync-automata related by an index set, that is their synchronisation conditions refer to sync-automata within the index set, a sync-program is obtained by running the sync-automata in parallel.

Definition 3 *Let $I = \{1, \ldots, n\}$ be an index set. The* sync-program *is a tuple $P^I = (S^I_0, P^I_1 || \ldots || P^I_n)$, where each P^I_i is a sync-automaton. The set $S^I_0 = S^0_1 \times \ldots \times S^0_n$ is the set of initial states of the sync-program.*

A *sync-subprogram* represents the behaviour of its sync-automata in isolation. We obtain a sync-subprogram by projecting a sync-program onto an index set $J \subseteq I$. We denote this by the projection operator $\restriction J$.

Definition 4 *Let $J \subseteq I$ be an index set and $J = \{j_1, \ldots, j_k\}$. Let $P^I = (S^I_0, P^I_1 || \ldots || P^I_n)$ with $P^I_i = (S_i, S^0_i, R_i)$ for each $i \in I$. Then $P^I \restriction J = (S^J_0, P^J_{j_1} || \ldots || P^J_{j_k})$ with $P^J_j = (S_j, S^0_j, R'_j)$ for each $j \in J$ where*

- *S_j and S^0_j are as in P^I_j;*
- *$R'_j = \{s_j \xrightarrow{\wedge_{j' \in L \cap J} A_{j'}:B_{j'}} t_j \mid s_i \xrightarrow{\wedge_{j' \in L} A_{j'}:B_{j'}} t_j \in R_j\}$.*

Initial states are $S^J_0 = S^0_{j_1} \times \ldots \times S^0_{j_n}$.

The projection contains sync-automata from J, each sync-automaton has the same states as its counterpart in P^I but synchronisation conditions on their moves concern only sync-automata from J. We remark that a sync-subprogram $P^I \restriction J$ is still a sync-program with index set J, hence it can be also denoted by P^J.

As an example, on fig. 2 there is sync-automaton P^I_{lac} after projection onto $\{lac, glu\}$.

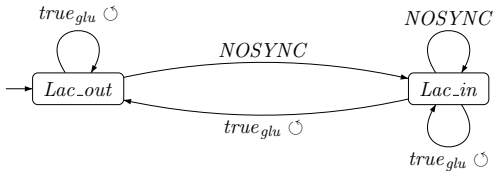

Figure 2: P^I_{lac} – Lactose

2.2 Semantics

Let I be an index set, where $I = \{1, \ldots, n\}$. An *I-state* is a tuple $s = (s_1, \ldots, s_n)$ where each s_i is a state of the sync-automaton P^I_i. An I-state represents a configuration of a program. I-state $s = \{s_1, \ldots, s_n\}$ can be projected onto a single component index $i \in I$ as follows: $s\lceil i = s_i$. Similarly, s projected onto $J \subseteq I$ with nodes $\{j_1, \ldots, j_k\}$, is $s\lceil J = (s_{j_1}, \ldots, s_{j_k})$. In program P^I we define the type of a move, represented by function $type : \bigcup_{i \in I} R_i \rightarrow I$, the index of the sync-automaton the move belongs to. Function *types* provides the same functionality on a set of moves.

Now we can proceed to defining the semantics of sync-programs as a labelled transition system on I-states, called an I-structure. First, we give the definition of a synchronisation, which is a complete set of moves from different automata having all their synchronisation requirements satisfied.

Definition 5 *Let I be an index set. We call a set of moves MOV a* synchronisation *iff*

1. *all moves in MOV are from distinct sync-automata in I*
2. *if $m \in MOV$ is of type i and has the form $s_i \xrightarrow{\wedge_{j \in L} A_j : B_j} t_i$, then for all $j \in L$ there is a move $m' \in MOV$ of type j and has the form $s_j \xrightarrow{sc_j} t_j$ and for all $p \in A_j$: $s_j(p) = tt$ and for all $p \in B_j$: $t_j(p) = tt$*
3. *MOV is* complete*, i.e. if $m \in MOV$ is of type i and has the form $s_i \xrightarrow{\wedge_{j \in L} A_j : B_j} t_i$, then $L = types(MOV) - \{i\}$*

A pair of I-states such that the second state can be reached from the first one by carrying out the synchronisation in question is called a support of the synchronisation.

Definition 6 *Let I be an index set. Consider a synchronisation MOV consisting of moves $s_i \xrightarrow{sc_i} t_i$ for all $i \in types(MOV)$. We call a couple of states (s,t) the* support of synchronisation *MOV iff s and t are I-states and*

1. *$s\lceil i = s_i$ for all $i \in types(MOV)$*
2. *$t\lceil i = t_i$ for all $i \in types(MOV)$*
3. *for all $i \in |I| - types(MOV)$: $s\lceil i = t\lceil i$.*

The semantics thus is a labelled transition systems over I-states where the transitions between two states s and t are obtained from the synchronisations with support (s,t).

Definition 7 *Let $I = \{1,\ldots,n\}$ be an index set. The* semantics *of $P^I = (S_I^0, P_1^I||...||P_n^I)$ is given by the I-*structure *$M_I = (S_I, S_I^0, R_I)$, where S_I is a set of I-states, $S_I^0 \subseteq S_I$ is the set of initial states and $R_I \subseteq S_I \times P(|I|) \times S_I$ is a transition relation giving the transitions of P^I. A transition (s,l,t) is in R_I iff there is a nonempty set MOV of moves such that $l = types(MOV)$ and MOV is a synchronisation with support (s,t).*

A transition of the form (s,l,t) corresponds to the situation where sync-automata with indices in l perform moves and the rest stays idle.

The synchronisation corresponding to a transition label l may contain only one index, let us assume it is i. In this case there is a move in the sync-automaton P_i^I that does not require synchronisation with other sync-automata, i.e. set L in the synchronisation condition of such a move is empty. Note that the second condition of the definition of the synchronisation is satisfied vacuously. In this situation sync-automaton P_i^I performs an autonomous $NOSYNC$ move from $s\lceil i$ to $t\lceil i$.

The case in which l contains more indices corresponds to the synchronisation of the sync-automata. Sync-automata with index in l can perform a move if all their synchronisation requirements against other sync-automata are satisfied. In particular, for sync-automaton P_j^I set A_j must be satisfied in the starting state and B_j in the ending state of the transition, respectively. Moreover, inclusion of L in l guarantees that all the required sync-automata will really participate in the synchronisation. Moreover no other automata are in the synchronisation, thus relating the synchronisation with its support.

The completeness requirement in the definition of a synchronisation intuitively means that in order for the synchronisation of several sync-automata to take place, it is necessary that the move performed by each sync-automaton refers in its synchronisation condition to every other participating sync-automaton. This also implies that it is not possible that the synchronisation is composed of several disjoint sets, each of which could be a synchronisation alone.

As an example of a transition, suppose that $I = \{1, 2, 3\}$ and sync-automaton P_1^I contains $a \xrightarrow{A:B} b$, P_2^I contains a move $A \xrightarrow{a:b} B$ and sync-automaton P_3^I has a move $X \xrightarrow{\neg a:b} Y$. Then there is a synchronisation $\{a \xrightarrow{A:B} b, A \xrightarrow{a:b} B\}$, its support is $([a, A, X], [b, B, X])$. Thus, $\mathcal{M}_I$ contains a transition $([a, A, X], \{1, 2\}, [b, B, X])$ representing that sync-automata with indices 1 and 2 synchronise and P_3^I remains idle.

A transition (s, l, t) in an I-structure can be projected onto $J \subseteq I$ such that $l \cap J \neq \emptyset$ as follows: $(s, l, t)\lceil J = (s\lceil J, l \cap J, t\lceil J)$.

A concept that will allow us to reason about properties of programs, is that of path.

Definition 8 *Let I be an index set. A* path *in an I-structure $\mathcal{M}_I$ is a sequence of I-states and transition labels $\pi = (s^1, l^1, s^2, l^2, \ldots)$ such that for all m, $(s^m, l^m, s^{m+1}) \in \mathcal{R}_I$. A* fullpath *is a maximal path.*

A fullpath is infinite unless for some $s^{m'}$ there is no $s^{m'+1}$ and $l^{m'}$ such that $(s^{m'}, l^{m'}, s^{m'+1}) \in \mathcal{R}_I$. Let π^m denote the suffix of π starting in m-th I-state.

For a $J \subseteq I$ let us define a J-block of π to be a maximal subsequence of π that starts and ends in a state and does not contain a transition label containing any i such that $i \in J$. Thus we can consider π to be a sequence of J-blocks with two successive J-blocks linked by a transition label l such that $l \cap J \neq \emptyset$ (note that a J-block can consist of a single state). It also follows that $s\lceil J = t\lceil J$ for any pair of states s, t in the same J-block. Thus, if Bl is a J-block, we define $Bl\lceil J$ to be $s\lceil J$ for some state s in Bl. We now give the formal definition of path projection. Let Bl^n denote the n-th J-block of π.

Let π be $(Bl^1, l^1, Bl^2, l^2, \ldots)$ where Bl^m is a J-block for all m. Then the path projection is given by: $\pi\lceil J = (Bl^1\lceil J, l^1 \cap J, Bl^2\lceil J, l^2 \cap J, \ldots)$.

3 Modular Verification

In order to analyse the behaviour of a biological system we would like to verify properties of computation of sync-program P^I representing the system. Say that a property f_J only regards part of the system, in particular the part involving only sync-automata from $J \subseteq I$. We would like to check satisfaction of f_J on the semantics of P^I. In order to avoid space explosion, we want to check it on a smaller and more abstract semantics, in particular on the semantics of $P^J = P^I \restriction J$. The subprogram P^J abstracts from the behaviour of sync-automata non-present in J and poses less restrictions in terms of synchronisation requirements. Thus its semantics represents an overapproximation of the behaviour of part of P^I concerning J.

To be able to perform the verification on the smaller semantics we need to prove that every computation concerning sync-automata from J of the program P^I is present as a computation of P^J.

It is reasonable to define a computation as a fullpath in the semantics of the sync-program. We need to show that every fullpath in the semantics of P^I projected onto J is a fullpath in the semantics of P^J. Firstly, we prove that every path in $\mathcal{M}_I$ projected onto J is a path in $\mathcal{M}_J$.

Lemma 1 (Transition projection) *Let I be an index set and $\mathcal{M}_I = (\mathcal{S}_\mathcal{I}, \mathcal{S}_0^I, \mathcal{R}_I)$ the semantics of sync-program P^I. For all I-states s, t in $\mathcal{S}_I$ and all $l \in \mathcal{P}(I)$, transition (s, l, t) is in $\mathcal{R}_I$ iff for all $J \subseteq I$ such that $l \cap J \neq \emptyset$, $(s, l, t)\lceil J$ is in $\mathcal{R}_J$, where $\mathcal{M}_J = (\mathcal{S}_\mathcal{J}, \mathcal{S}_0^J, \mathcal{R}_J)$ is the semantics of sync-program $P^J = P^I \restriction J$.*

Proof: Direction right to left. Suppose that for any $J \subseteq I$ such that $l \cap J \neq \emptyset$, $(s, l, t)\lceil J \in \mathcal{R}_J$. By taking $J = I$ we get $(s, l, t) \in \mathcal{R}_I$.

Direction left to right. Suppose that $(s, l, t) \in \mathcal{R}_I$, we will show $(s, l, t)\lceil J \in \mathcal{R}_J$ for any $J \subseteq I$ such that $l \cap J \neq \emptyset$.

From the definition of the semantics of sync-program P^I we have that there is a synchronisation MOV with support (s, t) such that $types(MOV) = l$. We need to show that in $\mathcal{M}_J$ there is a synchronisation MOV' with support $(s\lceil J, t\lceil J)$ such that $types(MOV') = l \cap J$. Let us consider the set $MOV' = \{s_i \xrightarrow{\wedge_{j \in L \cap J} A_j : B_j} t_i \mid s_i \xrightarrow{\wedge_{j \in L} A_j : B_j} t_i \in MOV \text{ and } i \in J\}$. Since $l \cap J \neq \emptyset$, MOV' is not empty. Then

- all moves are from distinct automata, because it was so in MOV.

- if $m \in MOV'$ is of type i and has the form $s_i \xrightarrow{\wedge_{j\in L} A_j:B_j} t_i$, then for all $j \in L \cap J$ there is a move $m' \in MOV'$ of type j and has the form $s_j \xrightarrow{sc_j} t_j$ and for all $p \in A_j$: $s_j(p) = tt$ and for all $p \in B_j$: $t_j(p) = tt$. In particular it is the move $s_j \xrightarrow{sc'_j} t_j$ where $sc'_j = \wedge_{j'\in L'\cap J} A_j{:}B_j$ if $sc_j = \wedge_{j'\in L'} A_j{:}B_j$.
- the completeness of MOV' comes from the completeness of MOV.

Hence, by definition of a synchronisation MOV' is a synchronisation in P^J. Moreover, since for all $i \in I - l : s\lceil i = t\lceil i$, it holds for subset $J \cap (I - l) = J - J \cap l$. That means that $(s\lceil J, t\lceil J)$ is the support of synchronisation MOV' in P^J. Also, $types(MOV') = l \cap J$. Thus, by definition of semantics of P^J the tuple $(s\lceil J, l \cap J, t\lceil J) = (s, l, t)\lceil J$ is in $\mathcal{R}_J$. □

Lemma 2 (Path projection) *Let I be an index set and $\mathcal{M}_I$ semantics of sync-program P^I. For every $J \subseteq I$ if π is a path in $\mathcal{M}_I$ then $\pi\lceil J$ is a path in $\mathcal{M}_J$, where $\mathcal{M}_J$ is the semantics of sync-program $P^J = P^I \restriction J$.*

Proof: Let $\pi = (Bl^1, l^1, Bl^2, l^2, \ldots)$ be a path in $(M)_I$ and Bl^m J-blocks for all m. By s^m and t^m denote first and last state of Bl^m, respectively. By definition of I-structure we have that transition (t^m, l^m, s^{m+1}) is in $\mathcal{M}_I$ for all m. By transition projection lemma transition $(t^m, l^m, s^{m+1})\lceil J = (t^m\lceil J, l^m \cap J, s^{m+1}\lceil J)$ is in $\mathcal{M}_J$ for all m. Now since $s^m\lceil J = t^m\lceil J$ for all m, we get that $(s^m\lceil J, l^m \cap J, s^{m+1}\lceil J)$ in $\mathcal{M}_J$ for all m. Hence sequence $(s^1\lceil J, l^1 \cap J, s^2\lceil J, l^2 \cap J, \ldots)$ satisfies the definition of a path in $\mathcal{M}_J$. □

However, with computation defined as a fullpath, violation of the desired computation preservation might occur. In particular, violation arises when an independent partition P of sync-program P^I exists that can be executed forever, while not allowing execution of other enabled sync-automata outside P. Consider a fullpath π in $\mathcal{M}_I$ and a state t from which only sync-automata in P are executed. When projecting π onto $J = (I - P)$ composed only of idle sync-automata, a finite path $\pi\lceil J$ is obtained. However, as in t some automata from J are enabled but not executed, in $\mathcal{M}_J$ they can be executed and thus a path ending in $t\lceil J$ is not a fullpath. Hence, π is a computation of P^I but $\pi\lceil J$ is not computation of P^J.

We need to refine the definition of computation, so that for all J a computation of P^I projected onto J will be a computation in P^J. We restrict ourselves to a special class of "fair" computations, in particular those in which every sync-automaton is executed infinitely many times. We define fairness as a property of paths in $\mathcal{M}_I$.

Definition 9 *A path* $\pi = (s^1, l^1, s^2, l^2, \ldots)$ *in* $\mathcal{M}_I$ *is* fair *iff for all* $i \in I$ *we have that* $\{m \mid i \in l^m\}$ *is infinite.*

Note that every fair path is infinite and each component involved in a fair path must have an infinite behaviour. Finite behaviours of components can be simulated by adding looping moves to the final states.

For the systems we aim to describe, the fairness assumption is reasonable since we regard a behaviour of biological system correct when all components are able to perform their function. Moreover, there is a class of systems where all fullpaths are fair, we provide a non-trivial example of such a system in Section 4.

We remark, that transient behaviour of the system can be studied by considering only a portion of its infinite behaviour.

Now we prove, that this definition of fairness guarantees preservation of computation under projection.

Lemma 3 (Fullpath projection) *Let* $J \subseteq I$ *be an index set. If* π *is a fair fullpath in* $\mathcal{M}_I$*, then* $\pi\lceil J$ *is a fair fullpath in* $\mathcal{M}_J$*.*

Proof: By path projection lemma $\pi\lceil J$ it is a path in $\mathcal{M}_J$. Since π is a fair path in $\mathcal{M}_I$ by definition of path projection we get that $\pi\lceil J$ is a fair path in $\mathcal{M}_J$. From the definition of fairness follows that every fair path is infinite, i.e. it is a fullpath. □

Now we exploit the fullpath projection lemma to prove that behavioural properties expressed in a suitable logic that hold in a semantics of sync-subprogram also hold in the original sync-program. The logic we will consider is a fragment of the Computation Tree Logic CTL* which is an extension of classical logic that allows reasoning about an infinite tree of state transitions. It uses operators about branches (non-deterministic choices) and time (state transitions). Two path quantifiers A and E are thus introduced to handle non-determinism: Af meaning that f is true on all branches, and Ef that it is true on at least one branch. The time operators are F, G, X, U and U_w; Xf meaning f is true at the next transition, Gf that f is always true, Ff that f is eventually true, fUg meaning f is always true until g becomes true, and fU_wg meaning f is always true until g might become true. In this logic, Ff is equivalent to $trueUf$, fWg to $(fUg) \vee Gf$. We have the following duality properties: $\neg(E(f)) = A(\neg f)$, $\neg(F(f)) = G(\neg f)$, $\neg(fUg) = (\neg gU_w \neg f)$.

The Computation Tree Logic CTL is the fragment of CTL* where each temporal operator must be preceded by a path operator, and each path operator has to be immediately followed by a temporal operator.

Following Attie, we assume the logic ACTL$^-$ for specification of properties. ACTL is the "universal fragment" of CTL which results from CTL by restricting negation to propositions and eliminating the existential path quantifier and ACTL$^-$ is ACTL without the AX modality.

Definition 10 *The* syntax of ACTL$^-$ *is defined inductively as follows:*

- *The constants true and false are formulae. p and $\neg p$ are formulae for any atomic proposition p.*
- *If f, g are formulae, then so are $f \wedge g$ and $f \vee g$.*
- *If f, g are formulae, then so are $AX_j f$, $A[fUg]$ and $A[f\ U_w g]$.*

We define the logic ACTL^-_J to be ACTL$^-$ where the atomic propositions are drawn from $AP_J = \{AP_i \mid i \in J\}$. Abbreviations in ACTL$^-$: $AFf \equiv A[true\ Uf]$ and $AGf \equiv A[f\ U_w false]$.

Properties expressible by ACTL$^-$ formulae represent a significant class of properties investigated in systems biology literature as identified in [29], such as properties concerning exclusion (*It is not possible for a state S to occur*), necessary consequence (*If a state S_1 occurs, then it is necessarily followed by a state S_2*), and necessary persistence (*A state S must persist indefinitely*). Oscillatory behaviour [7] is describable by an ACTL$^-$ formula as well.

Occurrence, possible consequence, sequence and possible persistence are of inherently existential nature, and are not expressible in ACTL$^-$.

Definition of the semantics of ACTL$^-$ formulae on the I-structure follows. Note that only fair fullpaths are considered.

Definition 11 Semantics of ACTL$^-$. *We define $\mathcal{M}_I, s \vDash f$ (resp. $\mathcal{M}_I, \pi \vDash f$) meaning that f is true in structure $\mathcal{M}_I$ at state s (resp fair fullpath π). We define $\vDash$ inductively:*

- *$\mathcal{M}_I, s \vDash true$. $\mathcal{M}_I, s \nvDash false$. $\mathcal{M}_I, s \vDash p$ iff $s(p) = tt$.*
  $\mathcal{M}_I, s \vDash \neg p$ iff $s(p) = ff$.
- *$\mathcal{M}_I, s \vDash f \wedge g$ iff $\mathcal{M}_I, s \vDash f$ and $\mathcal{M}_I, s \vDash g$.*
  $\mathcal{M}_I, s \vDash f \vee g$ iff $\mathcal{M}_I, s \vDash f$ or $\mathcal{M}_I, s \vDash g$.

- $\mathcal{M}_I, s \vDash Af$ *iff for every fair fullpath* $\pi = (s, l^1, \ldots)$ *in* $\mathcal{M}_I$: $\mathcal{M}_I, \pi \vDash f$.
- $\mathcal{M}_I, \pi \vDash f$ *iff* $\mathcal{M}_I, s \vDash f$, *where* s *is the first state of* π
- $\mathcal{M}_I, \pi \vDash f \wedge g$ *iff* $\mathcal{M}_I, \pi \vDash f$ *and* $\mathcal{M}_I, \pi \vDash g$. $\mathcal{M}_I, \pi \vDash f \vee g$ *iff* $\mathcal{M}_I, \pi \vDash f$ *or* $\mathcal{M}_I, \pi \vDash g$.
- $\mathcal{M}_I, \pi \vDash fUg$ *iff there exists* $m \in \mathbb{N}$ *such that* $\mathcal{M}_I, \pi^m \vDash g$ *and for all* $m' < m : \mathcal{M}_I, \pi^{m'} \vDash f$.
- $\mathcal{M}_I, \pi \vDash fU_w g$ *iff for all* $m \in \mathbb{N}$, *if* $\mathcal{M}_I, \pi^{m'} \nvDash g$ *for all* $m' < m$ *then* $\mathcal{M}_I, \pi^m \vDash f$.

Now we give the main theorem of the paper which states that all ACTL$^-$ properties that hold in $\mathcal{M}_J$ also hold in $\mathcal{M}_I$.

Theorem 1 (Property preservation) *Let* $J \subseteq I$ *be an index set,* s *an* I*-state and* f *an* $ACTL_J^-$ *property. If* $\mathcal{M}_J, s\lceil J \vDash f$ *then* $\mathcal{M}_I, s \vDash f$.

Proof: By induction on the structure of f (for all s).

$f = p$. By definition of state projection and the fact that AP_is are pairwise disjoint, for all atomic propositions p from AP_J we get that $\mathcal{M}_J, s\lceil J \vDash p$ iff $\mathcal{M}_I, s \vDash p$. Analogously for $f = \neg p$.

$f = g \wedge h$. From the assumption $\mathcal{M}_J, s\lceil J \vDash g \wedge h$ by CTL semantics, $\mathcal{M}_J, s\lceil J \vDash g$ and $\mathcal{M}_J, s\lceil J \vDash h$. By induction hypothesis $\mathcal{M}_I, s \vDash g$ and $\mathcal{M}_I, s \vDash h$. Hence, $\mathcal{M}_I, s \vDash g \wedge h$. Case $f = g \vee h$ is proved analogously.

$f = A[gU_w h]$. Let π be an arbitrary fair fullpath starting in s. We establish $\mathcal{M}_I, \pi \vDash [gU_w h]$. By fullpath projection lemma $\pi\lceil J$ is a fair fullpath in $\mathcal{M}_J$, hence by the assumption $\mathcal{M}_J, \pi\lceil J \vDash [gU_w h]$. There are two cases:

1. $\mathcal{M}_J, \pi\lceil J \vDash Gg$. Let t be any state along π. By CTL semantics $\mathcal{M}_J, t\lceil J \vDash g$. by induction hypothesis we have $\mathcal{M}_I, t \vDash g$. Since t was an arbitrary state of π, we get $\mathcal{M}_I, \pi \vDash Gg$ and thus $\mathcal{M}_I, \pi \vDash gU_w h$.
2. $\mathcal{M}_J, \pi\lceil J \vDash [gUh]$. Let $s_J^{m''}$ be the first state along $\pi\lceil J$ that satisfies h. Then there is at least one state $s^{m''}$ along π such that $s^{m''}\lceil J = s_J^{m''}$. Let $s^{m'}$ be first such state. By induction hypothesis $\mathcal{M}_I, s^{m'} \vDash h$. From the definition of path projection any s^m with $m < m'$ projects to $s^m\lceil J$ that is before $s_J^{m'}$ in $\pi\lceil J$. By the assumption $\mathcal{M}_J, s^m\lceil J \vDash g$, hence by induction hypothesis $\mathcal{M}_I, s^m \vDash g$. By CTL semantics we get $\mathcal{M}_I, \pi \vDash gUh$.

In both cases we showed $\mathcal{M}_I, \pi \vDash gU_wh$. Since π was arbitrary fair fullpath starting in s, we conclude $\mathcal{M}_I, s \vDash A[gU_wh]$.

$f = A[gUh]$. Let π be an arbitrary fair fullpath starting in s. By fullpath projection lemma $\pi\lceil J$ is a fair fullpath in $\mathcal{M}_J$ and by the assumption $\mathcal{M}_J, \pi\lceil J \vDash [gUh]$. By the above case we get $s \vDash A[gUh]$. □

4 Application

In this section we give a model of the *lac* operon regulation process in terms of a sync-program. Our model is inspired by the CCS model of the same process given in [31].

4.1 *Lac* Operon Regulation

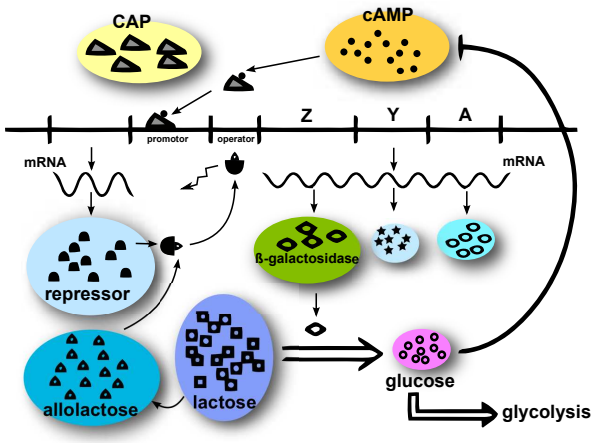

Figure 3: A diagram of *lac* operon regulation [18]

Bacteria have a simple general mechanism for coordinating the regulation of genes encoding products that participate in a set of related processes: these genes are clustered on the chromosome and are transcribed together. The gene cluster plus additional sequences that function together in regulation are called an operon [25].

The *lac* operon (see fig. 3) contains three genes related to lactose metabolism. The *lac* Z, Y and A genes encode β-galactosidase, galactoside permease and thiogalactoside transacetylase, respectively. β-galactosidase

converts lactose to galactose and glucose or to allolactose. Galactoside permease transports lactose into the cell and thiogalactoside transacetylase appears to modify toxic galactosides to facilitate their removal from the cell.

In the absence of lactose, the *lac* operon genes are repressed, namely they are transcribed at a basal level. This negative regulation is done by a molecule called Lac repressor, which binds to the operon, blocking the activity of RNA polymerase. The binding sites are called operators, the main operator is named O_1. The *lac* operon has two secondary binding sites for the Lac repressor: O_2 and O_3. To repress the operon, the Lac repressor binds to both the main operator and one of the two secondary sites.

When cells are provided with lactose, the *lac* operon is induced. An inducer molecule binds to a specific site on the Lac repressor, causing dissociation of the repressor from the operators. The inducer in the *lac* operon system is allolactose, an isomer of lactose. When unrepressed, transcription of *lac* genes is increased, but not at its highest level.

In addition, availability of glucose, the preferred energy source of bacteria, affects the expression of the *lac* genes. Expressing the genes for proteins that metabolise sugars such as lactose is wasteful when glucose is abundant. The *lac* operon deals with it through a positive regulation. The effect of glucose is mediated by cAMP, as a coactivator, and an activator protein known as cAMP receptor protein (CRP). When glucose is absent, CRP-cAMP binds to a site near the *lac* promoter and stimulates RNA transcription. In the presence of glucose, the synthesis of cAMP is inhibited and cAMP declines. Binding to DNA declines, thereby decreasing the expression of the *lac* operon.

CRP-cAMP is therefore a positive regulatory element responsive to glucose levels, whereas the Lac repressor is a negative regulatory element responsive to lactose. Consequently, strong induction of the *lac* operon requires both lactose (to inactivate the Lac repressor) and a lowered concentration of glucose (to trigger an increase in cAMP and increase binding of cAMP to CRP).

4.2 The Model

We will fix the index set $I = \{lac, \beta, allo, op, rep, pos, glu\}$ representing lactose, β-galactosidase, allolactose, *lac* operon, repressor, CRP-cAMP regulation and glucose, respectively. For the sake of simplicity we do not model

the activities of galactoside permease and thiogalactoside transacetylase.

We provide a sync-automaton for each biological component. In particular lactose is modelled by P^I_{lac} with $AP_{lac} = \{Lac\_none, Lac\_low\}$, β-galactosidase by P^I_β with $AP_{glu} = \{Beta\_low, Beta\_high\}$ and allolactose by P^I_{allo} with $AP_{allo} = \{Allo\_none, Allo\_low\}$. The *lac* operon is represented by P^I_{op} with $AP_{op} = \{Act, Rep\}$, the *lac* repressor by P^I_{rep} with $AP_{rep} = \{B1, B2, B3, Ballo\}$, the positive regulation by P^I_{pos} with $AP_{pos} = \{cAMP\_high, CRP{-}cAMP\}$ and finally glucose is represented by P^I_{glu} with $AP_{glu} = \{Glu\_high, Glu\_low\}$.

Sync-automaton P^I_{lac} (fig. 1 in Section 2) has two states, mappings of the set of atomic propositions AP_{lac} to $\{tt, ff\}$. For each state we display only the atomic propositions true in that state. Initially, there is no lactose in the cell. The scenario of the lactose never entering the cell is represented by the looping move in the state *lac_out*. This move synchronises with looping moves in other sync-automata representing the state corresponding to such a behaviour. The requirement of synchronisation with the other sync-automata is due to fairness, since no sync-automaton is allowed to remain blocked. External lactose entering the cell is modelled as a *NOSYNC* move because it is caused by mechanisms that are not considered in our model. Once inside the cell, lactose is transformed in the presence of β-galactosidase to glucose. This is modelled as a synchronisation with P^I_{glu} and P^I_β. On the other hand, when the enzyme β-galactosidase is absent, lactose is transformed to allolactose and it is non-deterministically decided whether all the lactose is consumed.

In sync-automaton P^I_β (fig. 4) β-galactosidase has two states representing its concentration level which are affected by activation and repression of *lac* operon P^I_{op}. When reacting with lactose, this enzyme, at low level, can produce allolactose or, at high level, can produce glucose and galactose. Since galactose does not participate in regulation we do not include it in our model. The change in the level of β-galactosidase is caused by the full expression of the operon, and that is done via two channels, positive and negative regulation. If lactose never enters the cell, β-galactosidase level remains low.

Allolactose P^I_{allo} (fig. 5) can be present at low concentration in the cell or be absent. Its level is increased as a result of the reaction of lactose with the β-galactosidase enzyme. When present, it can bind to *lac* repressor P^I_{repr} and its concentration will reduce. Allolactose remains absent, if lactose never enters the cell.

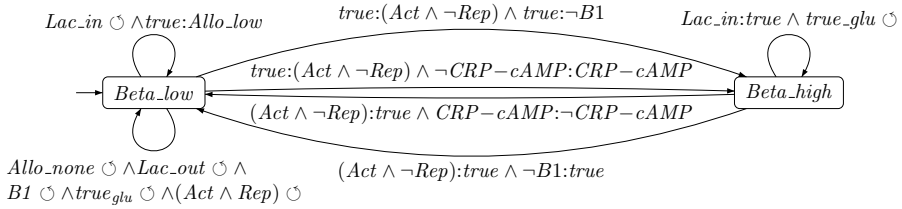

Figure 4: P^I_β – β-galactosidase

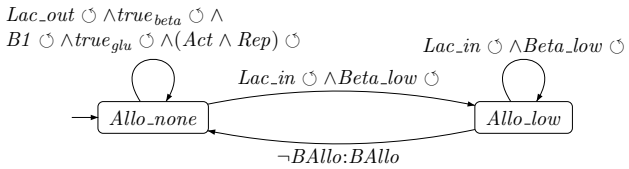

Figure 5: P^I_{allo} – Allolactose

The *lac* operon P^I_{op} (fig. 6) has four states, all possible truth value assignments to AP_{op}. Atomic propositions Act, Rep represent that the *lac* operon activated and repressed, respectively. Repression and unrepression (horizontal moves on fig. 6) are determined by negative regulation P^I_{repr} while activation and inactivation (vertical moves on fig. 6) by positive regulation P^I_{pos}. Note that full transcription of the operon genes occurs only when both unrepressed and activated (state $Act, \neg Rep$). This state also determines the concentration of β-galactosidase. Note that the absence of lactose makes the operon persist in the active but repressed state.

The *lac* repressor P^I_{repr} (fig. 7) has five states. After binding of *lac* repressor protein to O_1 site, it might bind either to the O_2 or O_3 sites. The level of *beta*-galactosidase may be decreased if it was high before the binding. These bindings repress the operon. When the inducer allolactose binds to the repressor, it releases the operator sites and unrepresses the operon, possibly changing the level of β-galactosidase. If allolactose does not bind, a looping move occurs.

The positive regulation P^I_{pos} (fig. 8) works as follows. When glucose level is low, cAMP concentration will be increased. Coactivator CRP creates a complex CRP-cAMP that binds to *lac* operon, stimulating the transcription. When the glucose concentration is increased, cAMP level will

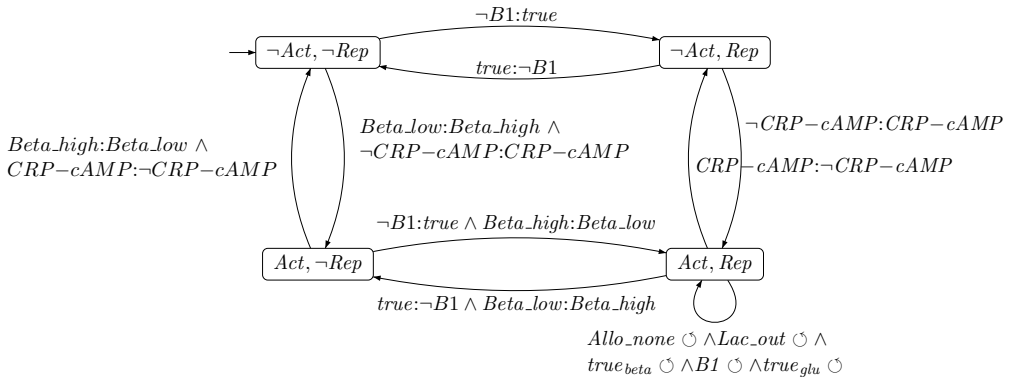

Figure 6: P^I_{op} – Lac operon

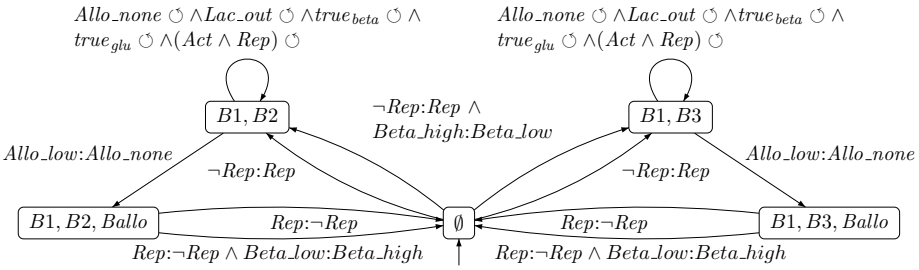

Figure 7: P^I_{repr} – Lac repressor protein (Negative regulation)

decrease and CRP-cAMP releases the operon site, deactivating the transcription. Depending on the state of the operon, the level of β-galactosidase can be affected.

In P^I_{glu} (fig. 9) glucose concentration can be high or low. The decrease of its concentration depends on factors that are not modelled. The increase of the concentration can occur via reaction of lactose and β-galactosidase. It is nondeterministically decided when the concentration level of glucose is considered high enough to pass to the state high. In addition, this component can be queried for the concentration level by P^I_{pos}. The lack of inner lactose makes the glucose level remain unchanged.

The sync-program describing the whole system is obtained by running all sync-automata in parallel $P^I = (S^I_0, P^I_{lac}||P^I_{\beta}||P^I_{allo}||P^I_{op}||P^I_{repr}||P^I_{pos}||P^I_{glu})$. The set of initial states is a combination of the sets of initial states of re-

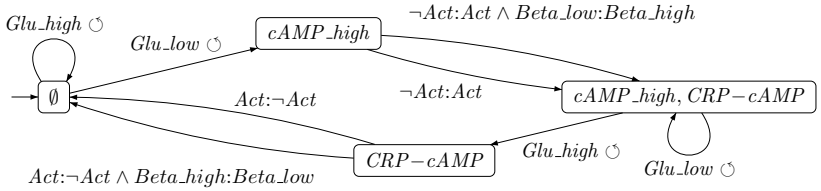

Figure 8: P^I_{pos} – CRP-cAMP (Positive regulation)

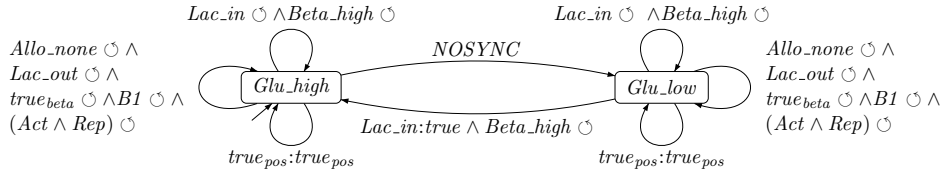

Figure 9: P^I_{glu} – Glucose

spective sync-automata.

5 Verification of Properties with NuSMV

In this section we shortly describe the NuSMV model checker, we present the translation of our model of the *lac* operon regulation process into the NuSMV input language and we show some results of property verification.

5.1 The Tool NuSMV

The NuSMV model checker [10] is a well-established and efficient symbolic model checker for Finite State Machines (FSMs). It is particularly suitable to be used in the specification and verification of sync-programs since its input language provides for modular descriptions and since it accepts CTL (that includes ACTL$^-$) as the language for property specification. Hence, NuSMV can be used both to verify CTL properties on sync-programs in the traditional way and to verify ACTL$^-$ properties by following the modular approach.

The input language can be used to describe systems in three different styles: as a synchronous system, as an asynchronous system, and by means

of a direct specification of the FSM. We shortly describe only the latter style, since it is the one we will use in the translation of our example model.

A direct specification of a FSM in NuSMV may consist of several *modules* that may have parameters. One module is called `main`, and it is the root of the model. As an example consider the following simple NuSMV model consisting of two modules: `main` and `proc`.

```
MODULE main
  VAR
    p1 : proc(p2.mutex);
    p2 : proc(p1.mutex);

MODULE proc(other)
  VAR
    mutex : boolean;
  DEFINE
    free := !other & !mutex;
  INIT
    mutex : FALSE;
  TRANS
    free & next(mutex) | mutex & next(!mutex);
```

The example describes two processes willing to access some resource in mutual exclusion. In module `main` we have two variables (defined in the `VAR` section of the module) `p1` and `p2` corresponding to the two processes. In module `proc` we have one boolean variable `mutex` that is true if the process is accessing the resource. The formal parameter `other` is a reference to the `mutex` variable of the other process (as it is stated in module `main`). In section `DEFINE` it is possible to define some macros (or shortcuts): in this case we define the macro `free` corresponding to `!other & !mutex` that is true if and only if neither of the two processes is accessing the resource (note that !, & and | represent negation, conjunction and disjunction, respectively). In section `INIT` the initial values of variables can be set (in this case `mutex` is set to false). Finally, in section `TRANS` a propositional formula can be given to specify the transition relation of the FSM. The formula can refer to the values of the variables before and after (by using the keyword `next`) the execution of the transition. In the example we have two possible transitions: the first from a state satisfying `free` to a state in which `mutex` is set to true, and the second from a state in which `mutex` holds to a state in which it is set to false.

This example of NuSMV model does not work as expected. In fact, the semantics of the language is such that transitions concerning different modules are performed in parallel. This causes the two processes `p1` and `p2` to be able to set their own `mutex` variables to true in the same step. This problem has to be solved by adding an extra module which chooses at each step which module is to perform the transition, thus simulating asynchronous behaviour. Moreover, the fairness is implemented on the level of the selector. When once a process is selected it must perform a transition, it is enough to schedule the processes infinitely often.

5.2 Ad Hoc Translation of the Example

Now we introduce the translation of our model into the NuSMV input language. The objective is to give one module for each sync-automaton plus a selector module for simulating asynchronous behaviour and the `main` module.

We give one module for each of the sync-automata. For example to P^I_{glu} corresponds the module

```
MODULE gluMod(...)
```

Each module is provided by a list of formal arguments, these will be discussed later in this section.

In the module, there are essentially three kinds of variables. The first are variables corresponding to the atomic propositions of the sync-automaton.

```
VAR
    glu_high : boolean;
```

Moreover, there are variables that express the fact that an atomic proposition is true in a state reachable in one move from the current state.

```
    toGlu_low : boolean;
```

We call these variables *to-variables*. Lastly, there are module-specific variables `true_glu` and `toTrue_glu` that are used in our encoding of the automaton.

We define macros for specifying the states of the module. These states consist of a conjunction of variables or their negations, where all the variables are listed. The states are of two types. The first are states corresponding to the states of the sync-automaton, having all to-variables false.

```
DEFINE
    state1 := glu_high & !toGlu_high & !glu_low & !toGlu_low
              & !dummy & !toTrue_glu;
```

Then there are so-called *intermediate states*. For every distinct move in the sync-automaton, there is one distinct corresponding intermediate state in the module. For example, for the move from state $\{Glu\_low, \neg Glu\_high\}$ to $\{\neg Glu\_low, Glu\_high\}$ there is a corresponding intermediate state. The variables corresponding to the atomic propositions match the corresponding automaton state – source of the move. The assignment to the to-variables corresponds to the target of the move in the sync-automaton.

```
    state7 := !glu_high &  toGlu_high &  glu_low & !toGlu_low
              & !dummy &  toTrue_glu;
```

Note that for every move between two states in the sync-automaton we need a distinct intermediate state in the module. This is assured by an extra variable `dummy` that is true exclusively in this intermediate state. If necessary, there can be more dummy variables in the module. For an example of extra variables, in order to distinguish between the two intermediate states corresponding to the two looping moves from $\{Glu\_low, \neg Glu\_high\}$, we have

```
    state6 := !glu_high & !toGlu_high &  glu_low &  toGlu_low
              &  dummy &  toTrue_glu;
```

whereas `state5` is identical with only one difference, that `dummy` is negated.

Some of the states are chosen to be initial:

```
INIT
    state1
```

The transitions of the program are made so that only one synchronisation of the modules is carried out at a time. A synchronisation is performed by a series of separate transitions of participating modules, one module at a time, in a specific order. For this purpose we assume an order of the modules. It can be arbitrary but fixed and we choose the order `alloMod`, `betaMod`, `posMod`, `gluMod`, `lacMod`, `opMod`, `reprMod`.

First, all the participating modules conduct the transition to the respective intermediate states. After finishing the first round, the second transition leads to the respective target states. Throughout the whole operation all non-participating modules cannot perform any move. Moreover,

the modules that successfully arrived in the target state will not leave that state until the synchronisation is finished, thus making it a sort of transaction.

In order to guarantee the above scenario, a move of an automaton has to be simulated by two transitions of the corresponding module. The first transition, from the source to the intermediate state has to wait until all the suitable transitions of modules preceding it in the order have been performed, and no other transition has been made in the system.

For example: the loop move in state $\{\neg Glu\_low, Glu\_high\}$ is translated to two transitions in the module `gluMod`. First, transition from `state1` to `state2` has to wait for the transition of module `betaMod` that is earlier in the order, precisely following the move's synchronisation condition. This is expressed by the condition in macro

```
p12 := beta_high & toBeta_high & lac_in;
```

The other part of the condition says that the remaining module, participating in this move but later in the order has to be ready for the transition. Moreover, all modules except for those participating on the synchronisation that precede the current module in the order, must not be in a to-state, as expressed by the following macro

```
n12 := _toTrue_allo &  toTrue_beta & _toTrue_pos &
    _toTrue_glu & _toTrue_lac & _toTrue_op & _toTrue_repr;
```

where `toTrue_mod` is a variable that is true precisely in all to-states of module `mod`. Note that we use an underscored variable in order to denote the negation of a variable, instead of using the language construct of negation `!`. The reason for keeping distinct the variable and its negation will be clear at the end of this subsection.

The transition itself is expressed as

```
state1 & p12 & n12 & next(state3)
```

As for the second move, analogously, the module has to check that all the preceding modules have arrived to the target state and the following modules are the only one waiting for the execution i.e. in a to-state.

```
state3 & p31 & n31 & next(state1)
```

Individual expressions of all the possible transitions connected into a disjunction form a characterisation of the transition relation in the section `TRANS` of the code.

In the case of the translation of a *NOSYNC* move, both transitions are only conditioned by `TRUE`. If a synchronisation condition refers to $true_i$ for some index i the translation operates with conditions using `true_i` and `toTrue_i`.

In the module `main` we instantiate all of the modules, passing the variables of all other modules as parameters to each module.

```
MODULE main
    VAR
        glu : gluMod(allo.allo_low,!allo.allo_low,...
```

Note that `allo.allo_low` is passed to formal parameter `allo_low` in module `gluMod` and its negation to the formal parameter `_allo_low`. The reason for keeping these parameters distinct is facilitating the creation of translation corresponding to a sync-subprogram obtained by means of a projection. For example, to get the sync-subprogram obtained by projecting the model to $\{op, rep\}$, we only need to initialise modules `op` and `repr` in the same manner as before, but with all references to modules outside of the projection substituted with `TRUE`. In this way the synchronisation conditions of modules outside of the projection become superfluous.

The NuSMV source code obtained by the translation of our *lac* operon model can be found online [17], along with the examples from the following section.

5.3 Experiments

We check some known properties of *lac* operon regulation. The checking is performed by the tool NuSMV on the translations of the indicated subprograms.

To guarantee that the properties are verified only in the states of the translated program that correspond to actual states of the sync-program, we need to guide the evaluation of satisfaction of atomic propositions to states that are not to-states. For this reason, each subformula f of an until formula $A[f\ U\ g]$ has to be changed to $\neg toTrue \rightarrow f$, where $toTrue = \bigvee_{i \in I} toTrue_i$ and each subformula g has to be changed to $\neg toTrue \wedge g$. We have to change subformulae of the weak until formula $A[f\ U_w\ g]$ analogously.

Theoretically, satisfaction of a property on $\mathcal{M}_J$ for $J \subseteq I$ guarantees its satisfaction in $\mathcal{M}_I$. Analogously, from the practical point of view, positive result of model checking of a property on a projection of a model implies

its truth in the entire model.

The property (P1) *"The allolactose bound to the repressor implies that the operon is repressed"* expressed by ACTL$^-$ formula

$$AG(Ballo \rightarrow Rep) \tag{P1}$$

is translated by *fortran* to

$$AG(\neg toTrue \rightarrow (Ballo \rightarrow Rep)) \tag{P1'}$$

which is in the NuSMV syntax written as

```
AG( notToTrue -> (repr.ballo -> op.rep) )
```
(P1')

is verified in the synchronisation skeleton representing the sync-subprogram $P^{op,repr}$ in less then 0.1 seconds.

```
NuSMV > check_ctlspec -p "AG(notToTrue->(repr.ballo->op.rep))"
-- specification AG(notToTrue->(repr.ballo->op.rep))  is true
NuSMV > time
elapse: 0.5 seconds
NuSMV > print_usage
BDD nodes allocated: 174750
```

In comparison, the verification in the whole model would take 0.5 seconds.

The property (P2) *"The increase of allolactose concentration can only be mediated by β-galactosidase in low concentration"* encoded in NuSMV as

```
AG( notToTrue -> (allo.allo_none & beta.beta_high ->
A[!allo.allo_low U beta.beta_low]) )
```
(P2')

is verified in the synchronisation skeleton representing the sync-subprogram $P^{allo,\beta}$ in less than 0.1 seconds. In comparison, the verification in the whole model would take 0.4 seconds.

The oscillation property (P3) *"While lactose is inside the cell, the operon will necessarily oscillate between a repressed and an unrepressed state"* encoded in NuSMV the global satisfaction of the conjunction of the following two formulae

```
(op.rep -> (AF (notToTrue & !op.rep) |
A[(notToTrue -> AF(notToTrue & !op.rep)) U
(notToTrue & !lac.lac_in)] ))                    (P3a')
```

and

```
(!op.rep -> (AF (notToTrue & op.rep) |
A[(notToTrue -> AF(notToTrue & op.rep)) U
(notToTrue & !lac.lac_in)] ))                    (P3b')
```

Hence, the Property (P3) encoded in NuSMV is

```
AG((P3a') & (P3b'))                              (P3')
```

This property is verified in the synchronisation skeleton representing the sync-subprogram $P^{\beta,lac,op,repr}$ in 0.1 seconds as instead of 1.2 seconds in the whole model.

The correctness property (P4) *"When the glucose concentration drops and lactose is inside the cell, the* lac *operon will eventually be fully expressed"* encoded in NuSMV as

```
AG(notToTrue -> (glu.glu_low & lac.lac_in ->
AF(notToTrue & op.act & !op.rep)))               (P4')
```

is verified in the synchronisation skeleton program $\mathcal{P}^{\mathcal{I}}$ in 0.9 seconds.

The negative regulation correctness property (P5) *"When glucose concentration is low, the* lac *operon will eventually be unrepressed"* encoded in NuSMV as

```
AG(notToTrue->
(lac.lac_in & op.rep-> AF(notToTrue & !op.rep)))  (P5')
```

is verified in the synchronisation skeleton representing the sync-subprogram $P^{\beta,lac,op,repr}$ in 0.1 seconds (cf. 0.7s).

The positive regulation correctness property (P6) *"When glucose concentration is low, the* lac *operon will eventually be activated"* encoded in NuSMV as

```
AG(notToTrue->
```

	Whole model		Modular	
Property	**Time**	**BDD size**	**Time**	**BDD size**
(P1')	0.4s	174750 nodes	<0.1s	3271 nodes
(P2')	0.4s	180278 nodes	<0.1s	1991 nodes
(P3')	1.2s	326908 nodes	0.1s	35269 nodes
(P4')	0.9s	283614 nodes	0.9s	283614 nodes
(P5')	0.7s	256112 nodes	0.1s	29193 nodes
(P6')	0.6s	222511 nodes	0.1s	21442 nodes

Table 1: Comparison of whole model verification with modular verification

`(glu.glu_low & !op.act-> AF(notToTrue& op.act)))` (P6')

is verified in the synchronisation skeleton representing the sync-subprogram $P^{\beta,lac,op,repr}$ in 0.1 seconds (cf. 0.6s).

Verification of properties on model fragments obtained by projecting on a subset of the system components takes much less time than verification of the same properties on the whole model. We compare in Table 1 the time necessary to verify the considered properties on the whole model and on the suitable model fragment we have identified. The table shows that the increase of efficiency of modular verification with respect to verification on the whole model can be significant, depending on the number of components to be involved in the modular verification.

Another value that is compared in the table is the size of the data structure used by the model checker (a Binary Decision Diagram – BDD). Again, the use of smaller models in the modular verification approach allows smaller data structures to be used for the representation of the state space. This is another important aspect of modular verification, which may permit verification of properties of systems whose complete behaviour representation would require data structures that could be too big to fit in the memory of a computer.

The worst case in modular verification of a property is the case in which all of the system components are necessary to verify it (as in the case of property (P4')). In this case modular verification coincides with verification on the whole model.

6 Discussion and Conclusions

We have presented a framework for modelling and modular verification of properties of biological systems. In particular we have developed an automata-based formalism of interactive systems that allows system components to perform transitions simultaneously in a rather general way. Moreover we have developed a modular verification technique for such a formalism that allows properties expressed in the universal fragment of CTL to be verified on suitably chosen fragments of models. As an application we have shown the modelling of the *lac* operon regulation process and the modular verification of some properties.

Our formalism, sync-programs, is a form of interacting finite-state automata which bears similarities with other formalisms, such as I/O automata [28], Team automata [26], interacting automata [9] and communities of interacting automata [27]. We have defined our formalism upon finite automata, since they are in particular connected with clear mathematical semantics, systems visualisation and decidability of many crucial behavioural properties [27]. Differently from most of the other definitions of interacting automata, which are based on environmental events or communication channels, we have employed state-based synchronisation conditions. This choice was motivated by the explicit aim of property verification and by the close relation to synchronisation skeletons, the shared-memory model used by Attie[2]. Finally, we have chosen the synchronisation among an arbitrary number of components, rather than a 2-way synchronisation, in order to allow a more natural modelling of biological phenomena. Multi-way synchronisation is used by several other formalisms, such as [26, 1, 34].

In the definition of our modular verification technique we have followed the "property preservation" approach, namely that truth of ACTL$^-$ formulae is preserved from sync-subprograms to the program. This has been originally considered by Grumberg and Long [23] and Dams [15] in different contexts. Other approaches to modular verification infer properties of a system from some properties of its components, e.g. [26] and [30].

Our technique is based on projections which can also be seen as a form of property-driven model reduction. Similar reduction methods have been proposed in [3, 5]. Differently from the other proposals, we operate on the syntax of the model rather than on its semantics.

The property preservation ensures that the truth of ACTL$^-$ is preserved from sync-subprograms to the program. Failure to verify a property in sync-subprograms does not help in establishing its satisfaction in the

whole program. However, it is worth noting, that in some cases model inspection aids finding a larger sync-subprogram that allows for successful verification of the property. Preservation of falsehood of ACTL$^-$ formulas amounts to full CTL preservation and can be obtained only under bisimilarity [15]. For application in systems biology see [31].

The ACTL$^-$ logic includes only universally quantified formulae. Preservation of these properties is guaranteed by the fact that the projection operation yields a reduced model that represents an overapproximation of the system behaviour. In order to preserve the satisfaction of existentially quantified formulae, such as EFg or $E(gUf)$ (as done in [15] in the context of abstract interpretation), one would need a different definition of the projection operation resulting in an underapproximation. Also in this case, however, the technique presents the problem of false negatives as they can be avoided only under bisimilarity between the whole and reduced models [15].

In the present work, we have only considered a fixed number of components. Since in biological systems it is often the case that the components are created and destroyed dynamically, we have developed in [19] an extension of our formalism with run-time creation of sync-automata. Such an extension allows several copies of sync-automata of the same type to be enabled at the same time, and to be created by moves of other sync-automata. The state space of the semantics of a sync-program is hence no longer finite, but it can be shown that such extended sync-programs can be translated into Place/Transition Petri nets, thus inheriting important decidability results. The extension of sync-automata permits a wider class of biological systems to be described. However, from the practical point of view it has still to be assessed whether a new verification tool should be developed for the formalism, or there is some existing model checker for infinite state systems that could be efficiently exploited.

As future work, we plan to extend our property preservation theorem to preserve all ACTL* properties in the way used in [23]. Moreover, we plan to improve our approach with a weaker notion of fairness, in the line of [2]. An ongoing work is giving a formal definition of the translation of sync-programs into a formalism similar to the input language of the NuSMV model checker.

In [11] the authors consider a modular approach to quantitative model checking in a biological context. A quantitative extension of our framework would be desirable in order to describe the systems more precisely. Pre-

liminary ideas about a quantitative extension of sync-programs have been presented in [21].

References

[1] Andre Arnold. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 1999.

[2] Paul C. Attie. Synthesis of large dynamic concurrent programs from dynamic specifications. *CoRR*, abs/0801.1687, 2008.

[3] Roberto Barbuti, Nicoletta De Francesco, Antonella Santone, and Gigliola Vaglini. LORETO: A tool for reducing state explosion in verification of LOTOS programs. *Softw., Pract. Exper.*, 29(12):1123–1147, 1999.

[4] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. A calculus of looping sequences for modelling microbiological systems. *Fundam. Inf.*, 72(1-3):21–35, 2006.

[5] Glenn Bruns. A practical technique for process abstraction. In *Proceedings of the 4th International Conference on Concurrency Theory*, CONCUR '93, pages 37–49, London, UK, 1993. Springer-Verlag.

[6] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[7] Laurence Calzone, Nathalie Chabrier-Rivier, François Fages, and Sylvain Soliman. Machine learning biochemical networks from temporal logic properties. *Transactions on Computational Systems Biology VI*, pages 68–94, 2006.

[8] Luca Cardelli. Brane calculi. *Computational Methods in Systems Biology*, pages 257–278, 2005.

[9] Luca Cardelli. Artificial biochemistry. *Algorithmic Bioprocesses*, pages 429–462, 2009.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International*

Conference on Computer-Aided Verification (CAV 2002), volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[11] Federica Ciocchetta, Maria Luisa Guerriero, and Jane Hillston. Investigating modularity in the analysis of process algebra models of biochemical systems. *CoRR*, abs/1002.4063, 2010.

[12] Federica Ciocchetta and Jane Hillston. Bio-PEPA: A framework for the modelling and analysis of biological systems. *Theor. Comput. Sci.*, 410(33-34):3065–3084, 2009.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[15] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[16] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theor. Comput. Sci.*, 325(1):69–110, 2004.

[17] http://www.di.unipi.it/msvbio/wiki/syncprog.

[18] Glycolytic pathway and lac operon of e. coli. CSML Model database. http://www.csml.org/.

[19] Peter Drábik, Andrea Maggiolo-Schettini, and Paolo Milazzo. Dynamic sync-programs for modular verification of biological systems. In *2nd Int. Workshop on Non-Classical Models of Automata and applications (NCMA'10)*, Jena, Germany, 2010.

[20] Peter Drábik, Andrea Maggiolo-Schettini, and Paolo Milazzo. Modular verification of interactive systems with an application to biology. *Electronic Notes in Theoretical Computer Science*, 268:61–75, 12 2010.

[21] Peter Drábik and Guido Scatena. An application of model checking to epidemiology (extended abstract). In *1st Int. Workshop on Applications of Membrane computing, Concurrency and Agent-based modelling in POPulation biology (AMCA-POP 2010)*, Jena, Germany, 2010.

[22] François Fages, Sylvain Soliman, and Nathalie Chabrier-Rivier. Modelling and querying interaction networks in the biochemical abstract machine biocham. *Journal of Biological Physics and Chemistry*, 4:64–73, 2004.

[23] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[24] John Heath, Marta Kwiatkowska, Gethin Norman, David Parker, and Oksana Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theor. Comput. Sci.*, 391(3):239–257, 2008.

[25] F Jacob and J Monod. Genetic regulatory mechanisms in the synthesis of proteins. *J Mol Biol*, 3:318–356, 06 1961.

[26] Jetty Kleijn. Team Automata for CSCW–A Survey–. *Petri Net Technology for Communication-Based Systems*, pages 295–320, 2003.

[27] I.A. Lomazova. Communities of interacting automata for modelling distributed systems with dynamic structure. *Fundamenta Informaticae*, 60(1-4):225–236, 2004.

[28] Nancy Lynch. Input/output automata: Basic, timed, hybrid, probabilistic, dynamic,... In Roberto Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 191–192. Springer Berlin / Heidelberg, 2003.

[29] Pedro T. Monteiro, Delphine Ropers, Radu Mateescu, Ana T. Freitas, and Hidde de Jong. Temporal logic patterns for querying dynamic models of cellular interaction networks. *Bioinformatics*, 24(16):i227–233, 8 2008.

[30] Michael Pedersen. Compositional definitions of minimal flows in Petri nets. In *Computational Methods in Systems Biology*, pages 288–307. Springer, 2008.

[31] Marcelo Cezar Pinto, Luciana Foss, José Carlos Merino Mombach, and Leila Ribeiro. Modelling, property verification and behavioural equivalence of lactose operon regulation. *Computers in Biology and Medicine*, 37(2):134–148, 2007.

[32] Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.

[33] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: an abstraction for biological compartments. *Theor. Comput. Sci.*, 325(1):141–167, September 2004.

[34] B. Zimmerová. *Modelling and Formal Analysis of Component-Based Systems in View of Component Interaction*. PhD thesis, Masaryk University, Brno, Czech Republic, 2008.