Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica
INF01

Ph.D. Thesis

Modular Verification of Biological Systems

Peter Drábik

Supervisor	Supervisor
Prof. Andrea Maggiolo-Schettini	Dr. Paolo Milazzo

Referee	Referee
Prof. Paul Attie	Prof. Monika Heiner

Chair
Prof. Pierpaolo Degano

November 22, 2011

Abstract

Systems of interest in systems biology (such as metabolic pathways, signalling pathways and gene regulatory networks) often consist of a huge number of components interacting in different ways, thus exhibiting very complex behaviours. In biology, such behaviours are usually explored by means of simulation techniques applied to models defined on the basis of system observation and of hypotheses on its functioning. Model checking has also been recently applied to the analysis of biological systems. This analysis technique typically relies on a state space representation whose size, unfortunately, makes the analysis often intractable for realistic models. A method for trying to avoid the state space explosion problem is to consider a decomposition of the system, and to apply a modular verification technique. In particular, properties to be verified often concern only a small portion of the modelled system rather than the system as a whole. Hence, for each property it would be useful to be able to isolate a minimal fragment of the model that is necessary to verify such a property.

In this thesis we introduce a modular verification technique in which the system of interest is described by means of an automata-based formalism, called sync-programs, that supports modular construction. Our modular verification technique is based on results of Grumberg et al. and on their application to the theory of concurrent systems proposed by Attie and Emerson. In particular, we adapt Attie and Emerson's approach to deal with biological systems by allowing automata to synchronise by performing transitions simultaneously.

Modular verification allows qualitative aspects of systems to be analysed with the guarantee that properties proved to hold in a suitable model fragment also hold in the whole model. The correctness of the verification technique is proved. The class of properties preserved is ACTL$^-$, the universal fragment of temporal logic CTL. The preservation holds only for positive answers and negative answers are not necessarily preserved.

In order to verify properties we use the NuSMV model checker, which is a well-established and efficient instrument. We provide a formal translation of sync-programs to simpler automata, which can be given as input to NuSMV. We prove the correspondence of the verification problems.

We show the application of our verification technique in some biological case studies. We compare the time required to verify the property on the whole model with the time needed to verify the same property by only considering those modules which are involved in the behaviour of the system related to the property.

In order to handle modelling and verification of more realistic biological scenarios, we propose also a dynamic version of our formalism. It allows entities to be created dynamically, in particular by other already running entities, as it often happens in biological systems. Moreover, multiple copies of the same entities can be present at the same time in a system. We show a correspondence of our model with Petri Nets. This has a consequence that tools developed for Petri Nets could be used also for dynamic sync-programs. Modular verification allows properties expressed as DACTL$^-$ formulae (dynamic version of ACTL$^-$) to be veried on a portion of the model.

The results of analysis of the case study of the MAP kinase cascade activated by surface and internalised EGF receptors, which consists of 143 species and 80 reactions, suggest applicability and scalability of the approach.

The results raise the prospect of rendering tractable problems that are currently intractable in the verification of biological systems. In addition, we expect that the techniques developed in the thesis could be applied with profit not only to models of biological systems, but more generally to models of concurrent systems.

Acknowledgments

First of all I thank my supervisors Prof. Andrea Maggiolo-Schettini and Dr. Paolo Milazzo for their wisdom, guidance and help.

I'm grateful to my reviewers Prof. Monika Heiner and Prof. Paul Attie and the thesis committee members Prof. Chiara Bodei and Prof. Gianluigi Ferrari for their precious comments and suggestions and Prof. Pierpaolo Degano for his assistance.

Without Damas Gruska I would never have started my Ph.D. studies, what a magnificent suggestion. I express my thanks to Jean Krivine for hosting me during my stay in Paris and to Fabio Martinelli for the support in Pisa.

I'm indebted to my personal Morpheus Franco Alberto and to my computer science friends and colleagues Rebecca, Lam, Claudio, Alessio, Rui, Giovanni, Giulio, Igor, Rossano, Dung, Lillo, Matteo, Gabriele, Valentina and Anna.

I acknowledge a long-term friendship with the physics department, thanks Luca, Lopa, Sebnem, Naveen, Kim, Sergiy, Antonio, Daniele, Anna and Andrea.

I owe a lot to girls and boys from my ultimate frisbee team Ultimate Pirates Viareggio, to Federica from Rifugio Casentini and to the people of Italy.

Finally, I would like to thank my girlfriend Ada, my parents and my sister Katarína for their love and support throughout my studies and beyond.

Contents

Chapter 1

Introduction

1.1 Motivation

A big challenge of current biology is understanding the principles and functioning of complex biological systems. Despite the great effort of molecular biologists investigating the functioning of cellular components and networks, we still do not know how to answer the question "how a cell works". At least not to a level to be able to easily modify or repair a cell.

In the last decades, scientists have gathered an enormous amount of molecular level information. The data are collected and analysed by means of bioinformatics, a growing discipline which includes genomics (finding the collection of all genes, for many genomes), transcriptomics (the collection of all actively transcribed genes), proteomics (the collection of all proteins), and metabolomics (the collection of all metabolites). For instance the Human Genome Project has obtained a huge quantity of data. However, this is just a beginning of understanding the human genome.

To uncover the principles of functioning of a biological system, just collecting data does not suffice. Actually, it is necessary to understand the functioning of parts and the way these interact in complex systems.

The aim of *systems biology* is to build, on top of the data, the science that deals with principles of operation of biological systems. The comprehension of these principles is done by modelling and analysis exploiting mathematical means.

A typical scenario of modelling a biological system is as follows. To build a model that explains the behaviour of a real biological system, first a formalism needs to be chosen. Then a model of the system is created, simulation is performed and the behaviour is observed. The model is validated by comparing the results with the real experiments.

The advantage of simulation is not only validation of laboratory experiments, but also prediction of behaviour under new conditions and automation of the whole process.

Simulation can give either the average system behaviour or a number of possible system behaviours. This may be insufficient when one is interested in analysing all the behaviours of a system.

A technique, called model checking, developed in computer science to study the behaviour of systems of interacting agents, may be of help. This technique permits the verification of properties (expressed as logical formulae) by exploring all the possible behaviours of a system.

This analysis technique typically relies on a state space representation whose size, unfortunately, makes the analysis often intractable for realistic models. This is true in particular for systems of interest in systems biology (such as metabolic pathways, signalling pathways, and gene regulatory networks), which often consist of a huge number of components interacting in different ways, thus exhibiting very complex behaviours.

The aim of this thesis is to develop a modular verification framework for models of biological systems that may allow the state space to be considered for the verification of properties to be significantly reduced.

1.2 State of the Art

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 [11, 20, 25, 32, 79, 80]. Examples of well-established formal frameworks that can be used to model, simulate and model check descriptions of biological systems are [25, 48, 55].

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. [29]). Moreover, the use of Binary Decision Diagrams (BDDs) [30] to represent the state spaces (symbolic model checking) often allows the tractable size of models to be significantly increased [18].

A method for trying to avoid the state space explosion problem is to consider a decomposition of the system, and to apply a modular verification technique allowing global properties to be inferred from properties of the system components. This is the approach that we follow in this thesis, and it can be particularly efficient when the modelled systems consist of a high number of components, whereas properties of interest deal only with rather small subset of them. This is often the case for properties of biological systems. Hence, for each property it would be useful to be able to isolate a minimal fragment of the model that is necessary for verifying such a property. If such a fragment can be obtained by working only on the syntax of the model, the application of a standard verification technique on the semantics of the fragment avoids the state explosion. In Figure 1.1 the approach is schematised, where the projection $\lceil J$ the syntactic operation used to obtain a fragment. The objective is to relate the properties that hold in the semantics of the fragment to the properties that hold in the semantics of the whole model. In particular, for a simple projection there are properties that are preserved from the parts to the whole.

$$\begin{tikzcd}
\boxed{P^I} \arrow[r, "Sem"] \arrow[dd, dashed, "\upharpoonright J"] & \boxed{Sem(P^I)} \arrow[dr, dashed, "\lceil J"] & \\
 & & \boxed{Sem(P^I)\lceil J} \\
\boxed{P^I\upharpoonright J} \arrow[r, "Sem"] & \boxed{Sem(P^I\upharpoonright J)} \arrow[ur, dotted, "?"] &
\end{tikzcd}$$

Figure 1.1: Modular verification – principle

A class of properties whose satisfaction is preserved from the components to the complete system was identified in Grumberg and Long [52] as ACTL, the universal fragment of CTL temporal logic. In particular, these properties can express behavioural patterns that hold universally, expressed by using the universal path quantifier of the logic. The rationale is that the universal properties declare that a behavioural property holds for all behaviours of the subsystem. Since by composing the subsystem, some behaviours can be occluded but none are added, the property continues to hold also after the composition. The composition plays the dual role to the projection, because the subsystems of interest are obtained by the projection of the whole system. This result is thus useful for verification of properties of parts.

A technique proposed by Attie and Emerson [7, 5, 6] exploits the preservation of these properties in order to verify concurrent programs and synthesise systems from specifications. Attie and Emerson use a formalism called synchronisation skeletons [27], an abstraction of sequential processes, suitable for describing distributed systems. The synchronisation skeletons are state-machines where states are connected by arcs representing conditional transitions. The transitions can be conditioned by the states of other synchronisation skeletons.

In order to be able to exploit the above-mentioned modular verification technique of property preservation in models of biological systems, it needs to be adapted. The interleaving nature of synchronisation skeletons makes them not suitable for modelling for systems biology. Indeed, in biological systems it is often the case that two or more entities interact and perform an action simultaneously. In synchronisation skeletons an agent checks the state of other agents and in condition on these it performs an action. Say that we want to model a synchronised transition of two agents. The agents agree on the synchronisation and one of them performs the move first. But then it is impossible to guarantee that the other agent finishes the synchronisation by performing the promised move and arrive at a correct state, since there is no mechanism to enforce the performing of a move. Therefore it is necessary to have a language primitive that allows for performing several transitions simultaneously.

1.3 Contributions

For the purpose of modular verification in this thesis we define sync-programs, an automata-based formalism of interactive systems which extends Attie and Emerson's approach by allowing processes to perform moves simultaneously. Actually, we consider a quite 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 component. This permits a wide range of interactions between biological entities to be suitably described.

The modular verification allows qualitative aspects of systems to be analysed with the guarantee that properties proved to hold in a suitable model fragment also hold in the whole model. The correctness of the verification technique is proved. The class of properties preserved is ACTL^-, the universal fragment of temporal logic CTL without the nextstate modality. The preservation holds only for positive answers, and negative answers are not necessarily preserved.

In order to verify properties of whole models or of model fragments it is possible to translate them into the input language of an existing model checking tool. Specifically, we use the NuSMV model checker [23], which is a well-established and efficient instrument. We provide a formal translation of sync-programs to simpler automata, which can be given as input to NuSMV. We prove the correspondence of the verification problems.

We show the application of our verification technique in some biological case studies. We compare the time required to verify the property on the whole model with the time needed to verify the same property by only considering those modules which are involved in the behaviour of the system related to the property.

In sync-programs we use a general multi-way type of synchronisation. In this synchronisation a move of an automaton explicitly indicates all the moves required to synchronise with it. Although being correct, this sometimes leads to quite complex model descriptions. Generalising, we modify the definition of synchronisation in order to obtain more succinct models. However, we argue that in order for the modular verification technique to work, a specific type of synchronisation is required for which we identify a necessary condition. Interestingly, this condition implies that a synchronisation that makes the modular verification correct coincides with the notion used in the original definition of sync-programs.

Another aspect that we deal with in the definition of our formalism is fairness, which is a constraint on a behaviour of the system that is often considered in concurrent systems but novel in the applications to systems biology. The notion of fairness is not only necessary to apply the modular verification but from our investigation it turns out to be useful to describe the behaviour of a biological system more accurately.

In order to handle modelling and verification of more realistic biological scenarios, we propose a dynamic version of our formalism. It allows entities to be created dynamically, in particular by other already running entities, as it often happens in biological systems. Moreover, multiple copies of the same entities can be present at the same time in a system. We show a translation of our formalism to Petri Nets. This has a consequence that tools developed for Petri Nets could be used also for dynamic sync-programs. Modular verification allows properties expressed as DACTL$^-$ formulae (dynamic version of ACTL$^-$) to be verified on a portion of the model.

We show how to verify a particular class of biological systems, metabolic pathways, which are series of biochemical reactions occurring within a cell. For this class of systems we show how to automatically decompose the monolithic model given as a set of reactions subject to some assumptions. We show encoding of the model as a sync-program and an ad-hoc translation to NuSMV. This case study, which consists of 143 species and 80 reactions, also provides a proof of concept for the scalability of the approach. In addition, it suggests that a tool for the analysis of pathways could be developed that allows biologists to study indispensability of components, effects of the removal (or mutation) of some components and causality relationships among components (or species). The tool would use model checking internally without exhibiting it to the user. Furthermore, the modular verification would consent to obtain results of analyses in the order of seconds or minutes even for complex pathways.

The results raise the prospect of rendering tractable problems that are currently intractable in the verification of biological systems. In addition, we expect that the techniques developed in the thesis could be applied with profit not only to models of biological systems, but more generally to models of concurrent systems.

1.4 Structure of the Thesis

The thesis is structured as follows.

- In Chapter 2 we recall some background notions in Computer Science, Mathematics and Biology that will be assumed in the rest of the thesis.

- In Chapter 3 we present the automata-based formalism of sync-program that is suitable for modelling biological systems with modularity in mind. The formalism is an extension of synchronisation skeletons with the possibility of performing moves simultaneously. We present the syntax and the semantics of sync-programs. As an example of modelling of biological systems we apply our formalism to the well-known biological process of *lac* operon gene regulation.

- In Chapter 4 we develop the modular verification technique for sync-programs. We specify the projection operators needed to obtain fragments of the program and show that any computation of a sync-program is preserved under the projection. Then we prove for any ACTL$^-$ formula, that if it is satisfied in a semantics of a projected sync-program, i.e. in a fragment of a model, then the formula is satisfied also in the whole sync-program. Finally, we apply the modular verification to the model of the *lac* operon and for some interesting properties of the system we indicate the smallest fragment sufficient for the verification.

- In Chapter 5 with the aim of doing the verification practically, we formally specify the translation of sync-programs to simpler automata that can be implemented in the tool NuSMV. We show the correctness of our translation with respect to the verification of ACTL$^-$ properties. We verify the properties of the *lac* operon model from the preceding chapter in NuSMV. We compare the time necessary to verify such properties by using our modular verification approach with respect to the the time of verifying the same properties on the whole model.

- In Chapter 6, we investigate the class of systems for which it is possible to provide a modular verification technique in the line of Chapter 4. Generalising, we modify the definition of synchronisation in order to obtain more succinct models. We identify a necessary condition for the technique and show that it coincides with our original definition of synchronisation. We briefly compare sync-automata to other formalisms and sketch the possibility of porting the verification technique to other formalisms.

- In Chapter 7, we extend the approach by allowing sync-automata (the components of a sync-program) to be created dynamically by other already running sync-automata. Moreover, we allow several instances of the same sync-automata to be executed concurrently, without any bound on the number of concurrent instances of the same sync-automaton. This extension is hence a new formalism that we call *dynamic sync-programs* and it is biologically motivated. We give the syntax and the semantics of the new formalism along with a Petri net representation which allows us to draw some results about the decidability of some problems for dynamic sync-programs. We develop a modular verification technique for dynamic sync-programs which will allow to prove formula preservation of a dynamic variant of the ACTL$^-$ logic that we define. We apply the approach to a biological case study, namely the EGF signalling pathway.

- In Chapter 8, show how to verify metabolic pathways. We show how to automatically identify components of a pathway and implement them as sync-automata. We also give an ad-hoc translation to NuSMV that shows to be more efficient for this class of systems than the one given in Chapter 5. We exemplify the approach on a case study of MAP kinase cascade activated by surface and internalised EGF receptors and verify some biologically relevant properties in a modular way.

- In Chapter 9, in order to be able to describe quantitative aspects of biological systems, we sketch a possible stochastic extension of dynamic sync-programs. Without studying their modular verification, we motivate and suggest possible future developments. We show an application of the extended formalism on systems of interest in epidemiology where we study some issues by using the probabilistic model checking as a tool.

Finally, in Chapter 10 we draw some conclusions and discuss related work and possible further developments.

1.5 Published Material

Part of the material presented in this thesis has appeared in several publications, in particular:

- Sync-programs and their modular verification presented in Chapters 3 and 4 have appeared in [38]. An extended version along with the translation of the case study from Section 3.3 according to the translation function given in Section 5.1 and the experiments from Section 5.5 are published in [39].

- The modular verification of dynamic sync-programs presented in Chapter 7 have appeared in [37].

- The application of the stochastic extension of dynamic sync-programs to Epidemiology and their analysis by using the probabilistic model checking from Chapter 9 were presented in [41].

- An investigation of the class of systems to which it is possible to apply the modular verification in relation to synchronisation, based on Chapter 6, has been published in [40].

All the published material is presented in this thesis in a revised and extended form.

Chapter 2

Background

2.1 Transition Systems and Logic

Transition systems Finite state transition systems (FSTSs) [30] provide a general description of a dynamical system that implicitly or explicitly underlies most description formalisms for concurrent systems. In particular this formalism also called Kripke structure is frequently derived from high-level languages used for modelling the dynamics of biological systems.

Moreover, statements in temporal logics, the specification language introduced in the following sections, are usually interpreted on FSTSs. A finite state transition system is formally defined as a tuple $\Sigma =< S, AP, L, T, S_0 >$, where S is a set of states, AP is a set of atomic propositions, $L : S \rightarrow 2^{AP}$ is a labelling function that associates to a state $s \in S$ the set of atomic propositions satisfied by s, $T \subseteq S \times S$ is a relation defining transitions between states, and $S_0 \subseteq S$ is a set of initial states.

A labelled transition system is an FSTS in which transitions are enriched with labels. In a labelled transition system the label of a transition usually denotes the event that has caused the transition.

Logic Computation Tree Logic CTL* [27] 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 progression (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, $f\ U\ g$ meaning f is always true until g becomes true, and $f\ U_w\ g$ meaning that either f remains true until g becomes true or f holds forever. In this logic, Ff is equivalent to *true* $U\ f$, $f\ U_w\ g$ to $(f\ U\ g) \vee Gf$. We have the following duality properties: $\neg(E(f)) = A(\neg f)$, $\neg(F(f)) = G(\neg f)$, $\neg(f\ U\ g) = (\neg g\ U_w\ \neg f)$.

The semantics of CTL* is evaluated on finite state transition systems. A path in FSTS $\Sigma =< S, AP, L, T, S_0 >$, starting from state s_0 is an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that $(s_i, s_{i+1}) \in T$ for all $i \geq 0$. We denote by π^k the path $s_k, s_{k+1}, \ldots$. The following is the inductive definition of the truth value of a CTL* formula in a state s or on a path π, in a FSTS Σ.

$$\begin{array}{lll}
\Sigma, s \models \alpha & \text{iff} & \alpha \in L(s) \\
\Sigma, s \models \neg f & \text{iff} & s \not\models f \\
\Sigma, s \models f \vee g & \text{iff} & \Sigma, s \models f \text{ or } \Sigma, s \models g \\
\Sigma, s \models Ef & \text{iff} & \text{there exists a path } \pi \text{ starting from } s \text{ s.t. } \Sigma, \pi \models f \\
\Sigma, s \models Af & \text{iff} & \text{for all paths } \pi \text{ starting from } s,\ \Sigma, \pi \models f \\
 & & \\
\Sigma, \pi \models f & \text{iff} & \Sigma, s \models f \text{ where } s \text{ is the first state of } \pi \\
\Sigma, \pi \models \neg f & \text{iff} & \Sigma, \pi \not\models f \\
\Sigma, \pi \models f \vee g & \text{iff} & \Sigma, \pi \models f \text{ or } \Sigma, \pi \models g \\
\Sigma, \pi \models Xf & \text{iff} & \Sigma, \pi^1 \models f \\
\Sigma, \pi \models f\ U\ g & \text{iff} & \text{there exists } k \geq 0 \text{ s.t. } \Sigma, \pi^k \models g \text{ and} \\
 & & \Sigma, \pi^j \models f \text{ for all } 0 \leq j < k \\
\Sigma, \pi \models f\ U_w\ g & \text{iff} & \text{either for all } k \geq 0, \Sigma, \pi^k \models f \text{ or there exists } k \geq 0 \\
 & & \quad \text{s.t. } \Sigma, \pi^k \models f \wedge g \text{ and } \Sigma, \pi^j \models f \text{ for all } 0 \leq j < k
\end{array}$$

The computation Tree Logic CTL [13] 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. The Linear Temporal Logic LTL [78] is the fragment of CTL* without path quantifiers, and where a formula is true in an FSTS if it is true on all paths. The expressiveness of LTL and CTL is incomparable.

Fairness Frequently, when dealing with systems consisting of components, we are interested only in computation paths that are "fair", in the sense that all components participate on the computation in a fair way. Appropriate fairness assumptions are often crucial for establishing that a program meets certain properties such as absence of starvation. For example, a computation is not fair if a component of the system is ignored forever.

Properties describing these requirements are expressible in CTL* but cannot be expressed directly in CTL. In order to deal with fairness in CTL, its semantics must be modified slightly. We limit our attention only to paths that satisfy *fairness constraints* expressing how the individual components should participate in the computation.

Several types of fairness are considered in the literature. For a comprehensive study of the matter see [45]. *Unconditional fairness* (also known as impartiality) requires that every process is executed infinitely often during the computation. In *weak fairness* (also known as justice) every process enabled almost everywhere must be executed infinitely often. *Strong fairness* implies that every process enabled infinitely often is executed infinitely often. Finally, in *generalised fairness* conditions about infinite computations may be expressed by means of any atomic proposition from the language. This is frequently done by using an LTL formula.

We remark that to allow verification of fairness constraints of computation paths we need to include in transition labels information about which component has participated on the state change. Therefore in the rest of the thesis labelled transition systems will be considered.

2.2 Model Checking

Model checking is a fully automatic verification method for both hardware and software systems. Essentially it is an exhaustive search of the state space of the system. Only finite state systems are considered. Model checking tools face a combinatorial blow up of the state space, commonly known as the state explosion problem. Efficient techniques have been proposed to handle this condition.

Model checking of CTL was originally proposed by Clarke, Emerson and Sistla [28]. It involves calculating satisfaction sets, $Sat(f)$, where $Sat(f) = \{s \in S : s \models f\}$. In order to calculate these sets the syntax tree of f is constructed and the subformulae are processed in a bottom-up manner, more precisely the leaves are labelled with the atomic propositions or true, while the inner nodes are labelled with $\_\neg\_, \_ \wedge \_, EX\_, E(\_U\_), EG\_$ formulae. Nodes labelled $\neg$ and X have exactly one son. Other inner nodes have two sons. The algorithm traverses the syntax tree in this (postfix) order and calculates the satisfaction sets recursively (i.e. syntax tree is not constructed explicitly). Let us suppose, the CTL formula is in the existential normal form, given by $true|a|f_1 \wedge f_2|\neg f|EXf|E(f\ U\ g)|EGf$. Satisfaction set of $true$ is the whole set of states. Satisfaction set of an atomic formula is a set of states in whose labelling this formula is included. For conjunction and negation we easily compute the satisfaction set from satisfaction sets of the subformulae. For the EXf formula, the satisfaction set consists of the states that lead to satisfaction set of f. Satisfaction set of $E(f\ U\ g)$ is computed from $Sat(g)$ by iteratively including states from $Sat(f)$ leading to this set. For $Sat(EGf)$, instead, we start with $Sat(f)$ and iteratively include states leading to this set.

These are some of more advanced techniques used in model checking of CTL.

Symbolic model checking. The state space is represented using boolean formulae. State exploration is carried out manipulating these formulae. This is possible thanks to OBDDs (Ordered Binary Decision Diagrams) that are a canonical representation for boolean formulae which is often efficient. Most notable model checkers based on OBDDs are SMV [23] and SPIN [61].

Explicit model checking. The state space is represented explicitly. State exploration is carried out using hash tables. State explosion limits the size of systems that can be handled using explicit techniques. However for protocol-like (asynchronous) systems explicit state representation usually works better than symbolic state representation. A important tool based on explicit state representation is the Murphi model checker [33].

SAT based model checking. By fixing in advance the maximum length of counterexamples to our specifications, a SAT solver can be used to look for counter examples. If we know that a counterexample will be not too long, using SAT is very efficient. In this context a system and the negation of the specification is written as a propositional boolean formula F in conjunctive normal form. Using SAT solver we look for a satisfying assignment for F. If we found one then we have a counterexample for the specification, otherwise the system satisfies the specification. Any efficient SAT solver can be used, e.g. SATO [86], GRASP [69]. SAT based model checker: BMC [15].

The power of traditional model checking is not exhaustive verification but rather its capability to generate useful diagnostic feedback in case a violation of the property is encountered. Due to this feature, model checking is seen as an effective and powerful bug-hunting technique: it does not only indicate that a property is refuted, but also indicates why.

Model checking has been recently applied in model validation in systems biology. All the properties we mentioned in Section 2.8 can be checked by using current model checking tools.

2.3 NuSMV Model Checker

The NuSMV model checker [23] is a well-established and efficient symbolic model checker for Finite State Machines (FSMs). It is particularly suitable to be used in a framework of modular verification since its input language provides for modular descriptions and because it accepts CTL (that includes ACTL$^-$, Section 4.3) as the language for property specification.

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 the last style.

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 : proc1(p2.mutex);
    p2 : proc2(p1.mutex);
```

```
MODULE proc1(other)
  VAR
    mutex : boolean;
  DEFINE
    free := !other & !mutex;
  INIT
    mutex : FALSE;
  TRANS
    free & next(mutex) | mutex & next(!mutex);

MODULE proc2(other)
  VAR
    mutex : boolean;
  DEFINE
    free := !other & !mutex;
  INIT
    mutex : FALSE;
  TRANS
    free & next(mutex) | mutex & next(!mutex);
```

The example describes two asynchronous 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.

Note that the direct specification of the transition relation works synchronously. A next state is any global state that satisfies transition relation specifications in all the modules. In this example, we demand an asynchronous execution. 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.

The asynchronicity has to be taken care of at the modelling level. It can be forced by means of an extra module which chooses at each step which module is to perform the transition at the next step. It consists of a variable with two possible values. Since there are no constraints on values of this variable, in each step it is non-deterministically decided what is its next value.

```
MODULE selector
  VAR
      select : {selP1,selP2};
```

In the module `main` we initialise the selector module. We pass the value containing the name of the module chosen for the current step as an argument to each of the modules `proc1` and `proc2` by modifying their headers.

```
MODULE main
  VAR
    sel  :  selector;
    p1  : proc1(sel.select,p2.mutex);
    p2  : proc2(sel.select,p1.mutex);
```

In the bodies of modules `proc1` and `proc2`, depending whether the module was chosen execution or not in the next step, the transition state change is specified or no change to the state is made.

```
MODULE proc1(select,other)
  VAR
    mutex : boolean;
  DEFINE
    free := !other & !mutex;
  INIT
    mutex : FALSE;
  TRANS
   ((free & next(mutex) | mutex & next(!mutex)) & next(select)=selP1)
   | (mutex=next(mutex) & next(select)!=selP1);
```

In this way the asynchronous execution is guaranteed.

A desired property for this program is that it should never be the case that the two processes `proc1` and `proc1` are at the same time in the mutex state. This property can be expressed by the following CTL formula:

```
AG! (p1.mutex & p2.mutex)
```

By running NuSMV with the commands

```
system prompt> NuSMV -int mutex.smv
NuSMV > go
NuSMV > check_ctlspec -p "AG! (p1.mutex & p2.mutex)"
```

we obtain the following output:

```
-- specification AG !(p1.mutex & p2.mutex)  is true
```

NuSMV tells us that the CTL specication is true. In the contrary case it would produce a counter-example path that can be used for debugging.

2.4 Synchronisation Skeletons

We recall the definition of the synchronisation skeleton model [27] used also by Attie and Emerson [7]. Note that for the purpose of this thesis we consider a simplified version of synchronisation skeletons without shared variables. For the purpose of easy extendability in the course of the thesis, we give definitions in a form slightly different from the one used in the above cited works.

Consider an index set I containing process indices. Let AP_i be a set of atomic propositions for process i, pairwise disjoint for all atomic propositions AP_j for all $i \neq j$. We define $I(i)$ as $I - \{i\}$.

Intuitively, the synchronisation skeleton of a process P_i with i from I is a state-machine where each state represents a region of code that performs some sequential computation and each arc represents a conditional transition (between different regions of sequential code) used to enforce synchronisation constraints.

Formally, a synchronisation skeleton is a directed graph where each node is a local state and each arc has a label that specifies an *enabling condition*. This condition describes the states of other synchronisation skeletons that make the move enabled.

Definition 2.1. A *synchronisation skeleton* SP_i^I, where i is an index and I is an index set, 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*;
- $R_i \subseteq S_i \times EC_i \times S_i$, where $EC_i \subseteq \bigcup_{L\subseteq I(i)} \Pi_{j\in L}\mathcal{P}(AP_j)$, are the *moves* between states.

A label of a move is called an enabling condition.

Definition 2.2. An *enabling condition* of a sync-skeleton SP_i^I is a label of the form $\{A_j \mid j \in L\}$ where $L \subseteq I(i)$ and A_j is a sets of atomic propositions drawn from AP_j or their negations.

We display an enabling condition $\{A_j \mid i \in L\}$, unless said otherwise, as a formula $\wedge_{j\in L} A_j$ where A_j is a conjunction of atomic propositions from AP_j. We adopt two views of A_j: one as a set of atomic propositions, one as a conjunction of these propositions. We switch between these views as convenient.

We can see an example of a synchronisation skeleton on Figure 2.1. The synchronisation skeleton SP_0^I, where $1 \in I$, is built over a set of atomic propositions $AP_0 = \{sb_0, ab_0, cm_0\}$. It contains three states, namely $\{sb_0, \neg ab_0, \neg cm_0\}$, $\neg\{sb_0, ab_0, \neg cm_0\}$ and $\{\neg sb_0, \neg ab_0, cm_0\}$, with the first chosen as initial. On the figure we display only the atomic propositions true in a state. The set of moves contains three regular and two looping moves. The enabling conditions of these moves consists of references to $AP_1 = \{sb_1, ab_1, cm_1\}$. For example, move $sb_0 \xrightarrow{sb_1} ab_0$ can be performed only if there is another synchronisation skeleton in a state that satisfies ab_1.

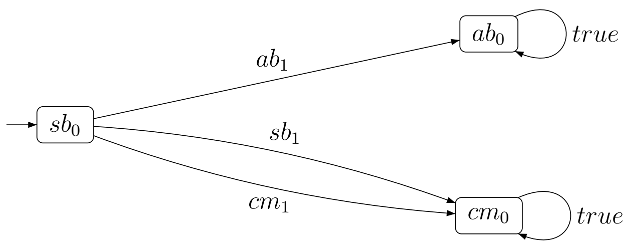


Figure 2.1: An example of sync-skeleton – SP_0^I.

A *synchronisation skeleton program* consists of a parallel composition of synchronisation skeletons related by an index set I.

Definition 2.3. Let $I = \{1, \dots, n\}$ be an index set. The *synchronisation skeleton program* is a tuple $SP^I = (S_0^I, SP_1^I || \dots || SP_n^I)$, where each SP_i^I is a synchronisation skeleton. The set $S_0^I = S_1^0 \times \dots \times S_n^0$ is the set of initial states of the synchronisation skeleton program.

The parallelism is modelled in the usual way by the non-deterministic interleaving of moves of the individual synchronisation skeletons of the processes P_i. Hence, at each step of the computation, some process with an enabled arc is non-deterministically selected to be executed next. Now we define the semantics of synchronisation skeleton formally. First, we define an I-state.

Let I be an index set where $I = \{1, \dots, n\}$. An *I-state* is a n-tuple $s = \{s_1, \dots, s_n\}$, where $s_i \in S_i$ for all $i \in I$. An I-state represents a global state composed of local states of all synchronisation skeletons in I. We denote the i-th element of an I-state s as $s\lceil i$.

The definition of semantics of synchronisation skeletons follows.

Definition 2.4. Let $I = \{1, \dots, n\}$ be an index set. The *semantics* of $SP^I = (S_0^I, SP_1^I || \dots || SP_n^I)$ is given by a *structure* $\mathcal{M}_I = (\mathcal{S}_I, \mathcal{S}_I^0, \mathcal{R}_I)$, where

- $\mathcal{S}_I = \Pi_{i \in I} S_i$ is a set of I-states
- $\mathcal{S}_I^0 \subseteq \mathcal{S}_I$ is a set of initial states

- $\mathcal{R}_I \subseteq \mathcal{S}_I \times \mathcal{P}(I) \times \mathcal{S}_I$ is a transition relation giving the transitions of SP^I. A transition (s, l, t) is in $\mathcal{R}_I$ iff there is an index i, such that
  - there is a move $s_i \xrightarrow{cc_i} t_i$ such that
    - $s_i = s\lceil i$ and $t_i = t\lceil i$
    - $cc_i = \{A_j \mid j \in L\}$ and $L \in I(i)$ and for all $j \in L$, A_j is true in s_j
  - for all j such that $j \neq i$, $s\lceil j = t\lceil j$.

A transition (s, l, t) in the transition relation $\mathcal{R}_I$ intuitively means that a move from synchronisation skeleton with index i has been performed and all the other synchronisation skeletons remained idle.

By using a syntactic projection on a synchronisation automaton, one obtains another synchronisation automaton that can be a base for modular verification. In [7] property preservation is proved, namely each ACTL$^-$ (see Section 4.3) property that can be verified in a semantics of a projected synchronisation skeleton program holds also in the semantics of the original synchronisation skeleton program.

2.5 Multisets

In mathematics, a multiset is a generalisation of a set. While each member of a set has only one membership, a member of a multiset can have more than one membership (meaning that there may be multiple instances of a member in a multiset).

Formally, a *multiset* is a 2-tuple (S, m), where S is a set and $m : S \to \mathbb{N}_{\geq 1}$ is a function from S to the set $\mathbb{N}_{\geq 1} = \{1, 2, 3, ...\}$ of positive natural numbers. The set S is called the *underlying set of elements*. For each a in S the *multiplicity* (that is, number of occurrences) of a is the number $m(a)$.

For example, in the multiset $\{a, a, b, b, b, c\}$ the multiplicities of the members a, b, and c are respectively 2, 3, and 1, and the cardinality of the multiset is 6. To distinguish between sets and multisets, we use a notation that incorporates brackets: the multiset $\{2, 2, 3\}$ can be represented as $[2, 2, 3]$.

We say that $M = (S, m)$ is a *multiset over S*. We denote the *set of all multisets over S* as $Mset(S)$. We say that $[a_1, \ldots, a_n]$ is a multiset over S iff for all $i \in \{1, \ldots, n\}$, $a_i \in S$ where some of $a_1, \ldots, a_n$ are possibly occurrences of the same element from S.

We say that an element a is a *member* of a multiset M, if a is a (set) member of the underlying set of M. We say that $M_1 = (S_1, m_1)$ is a *submultiset* of $M_2 = (S_2, m_2)$ iff $M_1 \subseteq M_2$ and for all $a \in M_1$, $m_1(a) \leq m_2(a)$. An *empty multiset* is denoted as $\emptyset$.

Now we describe several basic set operations.

Multiset intersection is specified as follows: if $M_1 = (S_1, m_1)$ and $M_2 = (S_2, m_2)$ then $M_1 \cap M_2 = (S_1 \cap S_2, m_\cap)$, where $m_\cap$ is defined on $S_1 \cap S_2$ and $m_\cap : a \mapsto min(m_1(a), m_2(a))$.

Multiset sum is specified as follows: if $M_1 = (S_1, m_1)$ and $M_2 = (S_2, m_2)$ then $M_1 \uplus M_2 = (S_1 \cup S_2, m_\uplus)$, where $m_\uplus$ is defined on $S_1 \cup S_2$ and $m_\uplus : a \mapsto m_1(a) + m_2(a)$.

Multiset difference is specified as follows: if $M_1 = (S_1, m_1)$ and $M_2 = (S_2, m_2)$ then $M_1 - M_2 = (S_1 \cup S_2, m_-)$, where m_- is defined on $S_1 \cup S_2$ and $m_- : a \mapsto max(m_1(a) - m_2(a), 0)$.

Now we specify the Cartesian product, relations and functions over multisets.

A *Cartesian product* of two multisets $M_1 = (S_1, m_1)$ and $M_2 = (S_2, m_2)$ is a multiset $M_1 \times M_2 = (S_1 \times S_2, m_\times)$, where $m_\times$ is defined on $S_1 \times S_2$ and $m_\times : (a_1, a_2) \mapsto m_1(a_1).m_2(a_2)$.

A *relation* between two multisets M_1 and M_2 is any submultiset of $M_1 \times M_2$.

A *function* $f : M_1 \to M_2$ from a multiset $M_1 = (S_1, m_1)$ to a multiset $M_2 = (S_2, m_2)$ is any relation between M_1 and M_2 with a constraint on the multiplicity function: for each $a_1 \in M_1$, $\Sigma_{(a_1,a_2) \in f} m_f((a_1, a_2)) = m_1(a_1)$. The sets S_1 and S_2 are called *domain* and *codomain*, respectively, of function f. The codomain of function is denoted as $Im(f)$.

As an example of a function over multisets, consider multisets $M_1 = [1, 1, 2]$ and $M_2 = [a, b, b]$. Then multiset $f_1 = [(1, a), (1, b), (2, a)]$ is an example of a function, note that each element in M_1 takes place of as many couples in f_1 as is its multiplicity in M_1. On the other hand, $f_1 = [(1, a), (1, b), (1, a)]$ is not a function, since there are three assignments to an element 1 from M_1, but its multiplicity in M_1 is 2.

Similarly to functions over sets, we can define when a function over multisets is injective, surjective and bijective.

We call a function $f : M_1 \to M_2$ *injective*, iff:
for each $a_2 \in M_2$, $\Sigma_{(a_1,a_2) \in f} m_f((a_1, a_2)) < m_2(a_2)$.

We call a function $f : M_1 \to M_2$ *surjective*, iff:
for each $a_2 \in M_2$, $\Sigma_{(a_1,a_2) \in f} m_f((a_1, a_2)) = m_2(a_2)$.

If a function f is both injective and surjective, then f is *bijective*.

2.6 Petri Nets

A Petri net [76] (also known as place/transition net) is a mathematical modelling language for the description of distributed systems. A Petri net consists of a Petri net graph (a bipartite graph whose vertices are divided into *places* and *transitions*) and of a *marking* (a function of places into natural numbers giving the number of *tokens* contained in each place).

Syntax A Petri net graph is a 3-tuple (S, T, W), where

- S is a finite set of *places*

- T is a finite set of *transitions*
- S and T are disjoint, i.e. no object can be both a place and a transition
- $W : ((S \times T) \cup (T \times S)) \to \mathbb{N}$ is a multiset of arcs, i.e. it defines arcs and assigns to each arc a non-negative integer arc multiplicity; note that no arc may connect two places or two transitions.

A *marking* of a Petri net graph is a multiset of its places, i.e. a function $M : S \to \mathbb{N}$. We say that a marking assigns a number of tokens to each place.

A Petri net is a 4-tuple (S, T, W, M_0), where

- (S, T, W) is a Petri net graph;
- M_0 is the *initial marking*, a marking of the Petri net graph.

Execution semantics The behaviour of a Petri net is defined as a relation on its markings, as follows.

Note that markings can be added like any multiset:
$M + M' = \{s \to (M(s) + M'(s)) \mid s \in S\}$. The *execution* of a Petri net graph $G = (S, T, W)$ can be defined as the transition relation $\to_G$ on its markings, as follows:

- for any *tr* in T: $M \to_{G,tr} M'$ iff there is $M'' : S \to \mathbb{N}$ s.t. $M = M'' + \Sigma_{s \in S} W(s, tr)$ and $M'' + \Sigma_{s \in S} W(tr, s) = M'$,
- $M \to_G M'$ iff there is $tr \in T$ s.t. $M \to_{G,tr} M'$.

In words: firing a transition *tr* in a marking M consumes $W(s, tr)$ tokens from each of its input places s, and produces $W(tr, s)$ tokens in each of its output places s.

We say that a marking M' is reachable from a marking M in one step if $M \to_G M'$. We say that it is *reachable* from M if $M \to_G^* M'$, where $\to_G^*$ is the reflexive and transitive closure of $\to_G$, that is, if it is reachable in zero or more steps.

For a Petri net $N = (S, T, W, M_0)$, we are interested in the firings that can be performed starting with the initial marking M_0. Its set of *reachable markings* is the set $R(N) = \{M' \mid M_0 \to_{(S,T,W)}^* M'\}$. The *reachability graph* of N is the transition relation $\to_G$ restricted to its reachable markings $R(N)$. The reachability graph of N represents the state space of the net.

One thing that makes Petri nets interesting is that they provide a balance between modelling power and analysability: many problems one would like to analyse about concurrent systems can be automatically determined for Petri nets. However some of those analyses are very expensive to determine in the general case or it is even unknown at all how to do it even if we do know of their general decidability. Several subclasses of Petri nets have been studied that can still model interesting classes of concurrent systems, while these problems become easier.

2.7 Selected Notions in Biology

Cell biology, the study of the morphological and functional organisation of cells, is now an established field in biochemical research. Examples of processes of interest in cell biology are gene regulatory networks, metabolic networks and signalling pathways. The main actors are proteins and nucleic acids, which interact as in a complex system to perform their activities. The study of the cell as a complex system is the topic of the recently emerged research field of systems biology.

Now we give some fundamental notions of cell biology that will be useful to understand the case studies in this thesis.

Cells There are two basic classifications of cell: procaryotic and eucaryotic. Traditionally, the distinguishing feature between the two types is that a eucaryotic cell possesses a membrane-enclosed nucleus and a procaryotic cell does not. Procaryotic cells are usually small and relatively simple, and they are considered representative of the first types of cell to arise in biological evolution. Procaryotes include, for instance, almost all bacteria. Eucaryotic cells, on the other hand, are generally larger and more complex, reflecting an advanced evolution, and include multicellular plants and animals.

Proteins A eucaryotic or procaryotic cell contains thousands of different proteins, the most abundant class of biomolecules in cells. The genetic information contained in chromosomes determines the protein composition of an organism. As is true of many biomolecules, proteins exhibit functional versatility and are therefore utilised in a variety of biological roles. A few examples of biological functions of proteins are enzymatic activity (catalysis of chemical reactions), transport, storage and cellular structure.

Although biologically active proteins are macromolecules that may be very different in size and in shape, all are polymers composed by amino acids that form a chain. The number, chemical nature, and sequential order of amino acids in a protein chain determine the distinctive structure and characteristic chemical behaviour of each protein. The native conformation of a protein is determined by interactions between the protein itself and its aqueous environment, in which it reaches an energetically stable three–dimensional structure, most often the conformation requiring the least amount of energy to maintain. In this three dimensional structure, often very complex and involving more than one chain of amino acids, it is sometimes possible to identify places where chemical interaction with other molecules can occur. This places are called interaction sites, and are usually the basic entities in the abstract description of the behaviour of a protein.

Nucleic Acids (DNA and RNA) Similarly to proteins, nucleic acids are polymers, more precisely they are chains of nucleotides. Two types of nucleic acid exist:

the deoxyribonucleic acid (DNA) and the ribonucleic acid (RNA). The former contains the genetic instructions for the biological development of a cellular form of life. In eucaryotic cells, it is placed in the nucleus and it is shaped as a double helix, while in procaryotic cells it is placed directly in the cytoplasm and it is circular. DNA contains the genetic information, that is inherited by the offspring of an organism. A strand of DNA contains genes, areas that regulate genes, and areas that either have no function, or a function yet unknown. Genes are the units of heredity and can be loosely viewed as the organisms cookbook.

Like DNA, most biologically active RNAs are chains of nucleotides forming double stranded helices. Unlike DNA, this structure is not just limited to long double-stranded helices but rather collections of short helices packed together into structures akin to proteins. Various types of RNA exist, among these we mention the Messenger RNA (mRNA), that carries information from DNA to sites of protein synthesis in the cell, and the Transfer RNA (tRNA), that transfers a specific amino acid to a growing protein chain.

The Central Dogma of Molecular Biology The description of proteins and nucleic acids we have given suggests a route for the flow of biological information in cells. In fact, we have seen that DNA contains instructions for the biological development of a cellular form of life, RNA carries information from DNA to sites of protein synthesis in the cell and provides amino acids for the development of new proteins, and proteins perform activities of several kinds in the cell. Schematically we have this flux of information:

$$\text{DNA} \xrightarrow{\text{transcription}} \text{RNA} \xrightarrow{\text{translation}} \text{protein}$$

in which transcription and translation are the activities of performing a copy of a portion of DNA into a mRNA molecule, and of building a new protein by following the information found on the mRNA and by using the amino acids provided by tRNA molecules. This process is known as the Central Dogma of Molecular Biology.

Enzymes Enzymes are proteins that behave as very effective catalysts, and are responsible for the thousands of coordinated chemical reactions involved in biological processes of living systems. Like any catalyst, an enzyme accelerates the rate of a reaction by lowering the energy of activation required for the reaction to occur. Moreover, as a catalyst, an enzyme is not destroyed in the reaction and therefore remains unchanged and is reusable.

The reactants of the chemical reaction catalysed by an enzyme are called substrate. Substances that specifically decrease the rate of enzymatic activity are called inhibitors, and, in enzymology, inhibitory phenomena are studied because of their importance to many different areas of research. Inhibitors can be classified mainly in two types, either competitive or noncompetitive. The former are substances almost always structurally similar to the natural enzyme substrates and they bind to the enzyme at the interaction site where the substrates usually bind to. The latter are substances that bear no structural relationship to the substrates and that cannot interact at the active site of the enzyme, but must bind to some other portion of an enzyme.

Enzymes perform many important activities in cells. For example, DNA transcription and RNA translation are performed by enzymes, and in the external membrane of the cell there are enzymes responsible for transporting some molecules from the outside to the inside of the cell or vice versa.

2.8 Properties of Biological Systems and Logics

Qualitative properties are properties that are observed and can generally not be measured as contrasted to quantitative properties. In biological settings, qualitative properties deal with occurrence of cellular events and relations between them.

Properties of a biological system usually deal with states of the system. A state of a system is usually defined by the states of its elements. A state of the cell is defined by the values of the actors, either the presence or absence of molecules, or their number, or their concentration in each part of the cell and by general data like pH and the temperature. Note that the set of states can be just represented by partial information on the actual values of state variables, like for instance interval or constraints between variables.

We mention several most frequent types of properties investigated in the literature as identified in [73]. Monteiro et al. created a list of questions on the dynamics of genetic, metabolic and signal transductional networks. The identified questions were grouped into four categories, depending on whether they concerned the occurrence/exclusion, consequence, sequence and invariance of cellular events. These categories include the most widely asked questions, however they do not include all.

For each category, the formalisation of of all types of formulae is shown in terms of temporal logics CTL and CTL*. Some of these formulae are in a form that will allow for the modular verification approach developed in this thesis. On the other hand, formulae with existential quantifiers are not from ACTL$^-$ and thus the approach is not applicable to them.

These formalisations form formula patterns that can help writing correct formulae by instantiating these patterns. Obviously, it is possible to formally describe also properties not falling in one of the following groups. A resource on how to write CTL formulae can be found in [35].

Occurrence/exclusion

It is possible / not possible for a state S to occur.

State S is described by some condition that is satisfied in this state. This concept represents occurrence and its negation, exclusion. An example of this kind of property is "*Wild bacteria never produce mucus by themselves when starting from a basal state*". This type encompasses also questions about mutual exclusion, specifying the negation of the undesired state, i.e. the simultaneous occurrences of two phenomena. For instance, "*It is not possible for a state to occur in which there is a high concentration of protein P_1 and P_2*".

It is possible for a state S to occur.

$$EF(S) \tag{O1}$$

It is not possible for a state S to occur.

$$\neg EF(S) \tag{O2}$$

Consequence

If a state S_1 occurs, then it is possibly/necessarily followed by a state S_2.

This type represents an ordering relation between two events. More precisely, it expresses that if the first state occurs, then it is possibly or necessarily followed by the occurrence of the second state. If the latter state necessarily follows, then the consequence pattern expresses a form of causal relation. Examples of this type are, for instance, "*If a state occurs in which protein P is phosphorylated, then it is possibly followed by a state in which the expression of gene g decreases*", or "*If a state occurs in which the concentration of protein P is below 5 μM, then it is necessarily followed by a state in which the expression of gene g is at its basal level*".

If a state S_1 occurs, then it is possibly followed by a state S_2.

$$AG(S_1 \rightarrow EF(S_2)) \tag{C1}$$

If a state S_1 occurs, then it is necessarily followed by a state S_2.

$$AG(S_1 \rightarrow AF(S_2)) \tag{C2}$$

Sequence

A state S_2 is reachable and is possibly/necessarily preceded at some time / all the time by a state S_1.

This property type represents an ordering relation between two events. It ought not to be confused with the consequence pattern, since the conditional occurrence of the second state which characterises the latter, is absent in the sequence pattern. It must be possible to observe both the first and the second state, in that order, for an instance of the sequence pattern to be true. Four variants of this pattern are distinguished, depending on whether the second state follows possibly or necessarily after the first state, and whether the system is in the first state all the time or only at some time before the occurrence of the second state. Instances are "*A steady state is reachable and is necessarily preceded all the time by a state in which nutrient N is absent*" or "*For all states at all computations, BO2 and BO3 do not occur one after another and, at some time, they will occur between BO1 and UBO1*" [77]. By Chabrier et al. [22] this property type is called checkpoint.

A state S_2 is reachable and is possibly preceded at some time by a state S_1.

$$EF(S_1 \wedge EF(S_2)) \tag{S1}$$

A state S_2 is reachable and is possibly preceded all the time by a state S_1.

$$E(S_1 \; U \; S_2) \tag{S2}$$

A state S_2 is reachable and is necessarily preceded at some time by a state S_1.

$$EF(S_2) \wedge \neg E(\neg S_1 \; U \; S_2) \tag{S3}$$

A state S_2 is reachable and is necessarily preceded all the time by a state S_1.

$$EF(S_2) \wedge AG(\neg S_1 \rightarrow AG(\neg S_2)) \tag{S4}$$

Persistence

A state S can/must persist indefinitely.

In Chabrier et al. [22] S is called stable state. The invariance pattern is used to check if the system can or must remain indefinitely in a state. In contrast with the occurrence/exclusion pattern, the question is not whether a particular state can be reached, but rather whether a particular state is invariable. Instances of the pattern are "*A state in which reaction R occurs at a high rate can persist indefinitely*" and "*A state with a basal expression of gene g must persist indefinitely*".

A state S can persist indefinitely.

$$EG(S) \tag{P1}$$

A state S must persist indefinitely.

$$AG(S) \tag{P2}$$

Oscillation

The states S_1 and S_2 oscillate.

Another interesting and frequent property scheme describes oscillatory behaviour. This pattern (as identified in [19, 9]) says that whenever state S_1 occurs, it is possibly followed by state S_2 and dually, whenever state S_2 occurs, it is possibly followed by state S_1. An example of this type is "*The states of high concentration of protein P and low concentration of P oscillate*".

The states S_1 and S_2 oscillate.

$$EG((P \to EF(Q)) \land (Q \to EF(P))) \tag{Osc1}$$

It is worth noting that the formulation is necessary but not sufficient condition for oscillations. The correct formula for oscillations is indeed a CTL* formula that cannot be expressed in CTL [19, 12]:

$$EG((P \to F(Q)) \land (Q \to F(P))) \tag{Osc2}$$

Chapter 3

Sync-programs

In this chapter we define sync-programs, an automata-based formalism of interactive systems which extends synchronisation skeletons by allowing processes to perform moves simultaneously. As an example of modelling of biological systems we apply our formalism to the well-known biological process of *lac* operon gene regulation.

3.1 Syntax

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

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 3.1. 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 with index 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.

As an example, in Figure 3.1 we show three sync-automata, P_1^I, P_2^I and P_3^I, where I is the index set {1,2,3}. Sync-automaton P_1^I consists of two states, given by two assignments to the only atomic proposition A in AP_1. We write A and $\neg A$ to denote the assignment to A of *tt* (semantic true) and *ff* (semantic false), respectively. Its moves are labelled by conditions on moves of the remaining sync-automata, P_2^I and P_3^I. The sync-automaton P_2^I has three states, built over the set $AP_2 = \{X, Y\}$ and moves conditioned by moves of P_1^I and P_2^I. Similarly, the third automaton, P_3^I is constructed over $AP_3 = \{C\}$ and has two states. The sets of initial states of the three sync-automata are $S_1^0 = \{A\}$, $S_2^0 = \{X \wedge \neg Y\}$ and $S_3^0 = \{\neg C\}$. In the graphical representation they are denoted by dangling arrows.

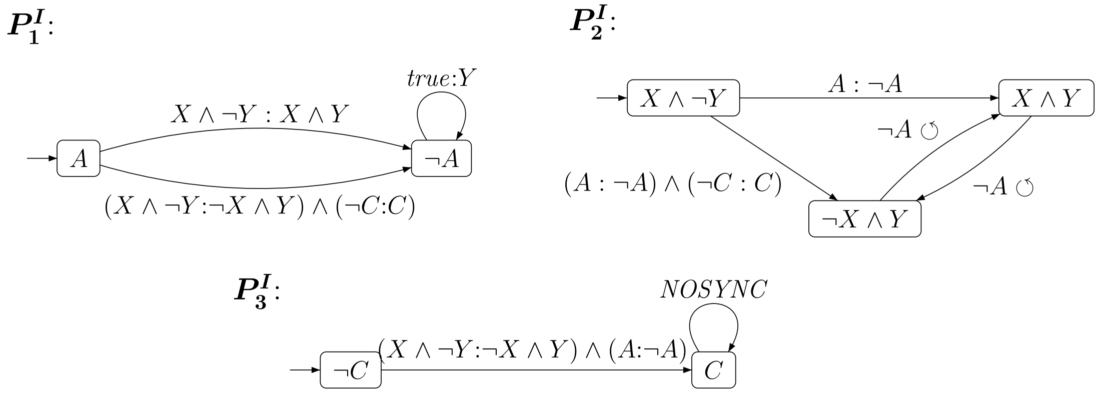

P_1^I:
$true{:}Y$
$X \wedge \neg Y : X \wedge Y$
A
$\neg A$
$(X \wedge \neg Y{:}\neg X \wedge Y) \wedge (\neg C{:}C)$
P_2^I:
$X \wedge \neg Y$
$A : \neg A$
$X \wedge Y$
$\neg A$ ↻
$(A : \neg A) \wedge (\neg C : C)$
$\neg X \wedge Y$
$\neg A$ ↻
P_3^I:
$NOSYNC$
$\neg C$
$(X \wedge \neg Y{:}\neg X \wedge Y) \wedge (A{:}\neg A)$
C

Figure 3.1: A sync-program consisting of three sync automata, P_1^I, P_2^I and P_3^I.

The synchronisation condition consists of a set of requirements against other sync-automata, each formed by a precondition and a postcondition on the move that is expected to be performed synchronously.

Definition 3.2. A *synchronisation condition* of sync-automaton P_i^I is a label of the form $\{A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}\}$ 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 $\{Aj_1{:}Bj_1, \ldots, Aj_n{:}Bj_n\}$ 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 literals 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 synchronously 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 $A_j{:}A_j$.

In the example in Figure 3.1, the sync-automaton P_1^I contains $A \xrightarrow{X \wedge \neg Y : X \wedge Y} \neg A$ that can be performed in synchronisation with a move of P_2^I that leads from a state that satisfies $X \wedge \neg Y$ to a state that satisfies $X \wedge Y$. The only such move in P_2^I is $X \wedge \neg Y \xrightarrow{A:\neg A} X \wedge Y$. Another option for changing the state from A to $\neg A$ is to synchronise with both P_2^I and P_3^I. Moreover, in the loop move in the state $\neg A$ the sync-automaton P_1^I is willing to synchronise with any move of P_2^I with the ending state satisfying Y. An example of a *NOSYNC* move can be found in P_3^I, in particular the looping move in state C.

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 3.3. Let $I = \{1, \ldots, n\}$ be an index set. The *sync-program* is a tuple $P^I = (S_0^I, P_1^I || \ldots || P_n^I)$, where each P_i^I is a sync-automaton. The set $S_0^I = S_1^0 \times \ldots \times S_n^0$ is the set of initial states of the sync-program.

Figure 3.1 shows the sync-program P^I with $I = \{1, 2, 3\}$ made up of sync-automata P_1^I, P_2^I and P_3^I.

3.2 Semantics

Let I be an index set, where $I = \{1, \ldots, n\}$. An I-state represents a configuration of a program and is defined as 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 $s = \{s_1, \ldots, s_n\}$ 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 over 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 3.4. 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 3.5. 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 $i \in I - types(MOV)$: $s\lceil i = t\lceil i$.

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

Definition 3.6. Let $I = \{1, \ldots, n\}$ be an index set. The *semantics* of $P^I = (S_I^0, P_1^I || \ldots || P_n^I)$ is given by the *I-structure* $\mathcal{M}_I = (\mathcal{S}_I, \mathcal{S}_I^0, \mathcal{R}_I)$, where $\mathcal{S}_I$ is a set of I-states, $\mathcal{S}_I^0 \subseteq \mathcal{S}_I$ is the set of initial states and $\mathcal{R}_I \subseteq \mathcal{S}_I \times \mathcal{P}(I) \times \mathcal{S}_I$ is a transition relation giving the transitions of P^I. A transition (s, l, t) is in $\mathcal{R}_I$ 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 third 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. Other automata are not included in the synchronisation, and as stated in the definition of support, they are required to stay idle. The support thus serves as a context of a synchronisation.

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 ensures that it is not possible that the synchronisation is composed of several disjoint sets, each of which could be a synchronisation alone. A more detailed discussion about the motivation for this definition and its impact on the modular verification can be found in Chapter 6. Note that since in practise it is often the case that quite few processes synchronise, the necessity of inclusion of references to all the synchronisation partners does not pose a real problem.

As an example of a transition, the program $P^I = P_1^I || P_2^I || P_3^I$ on Figure 3.1 contains a synchronisation of the moves $A \xrightarrow{X \wedge \neg Y : X \wedge Y} \neg A$ of P_1^I and $X \wedge \neg Y \xrightarrow{A : \neg A} X \wedge Y$ of P_2^I. Its support is $([A, X \wedge \neg Y, \neg C], [\neg A, X \wedge Y, \neg C])$. Thus, $\mathcal{M}_I$ contains a transition $([A, X \wedge \neg Y, \neg C], \{1, 2\}, [\neg A, X \wedge Y, \neg C])$ representing that sync-automata with indices 1 and 2 synchronise and P_3^I remains idle.

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

Definition 3.7. 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.

The dynamics of a sync-program is given through its computations. A *computation* is a fullpath in the semantics of the sync-program. We restrict ourselves to a

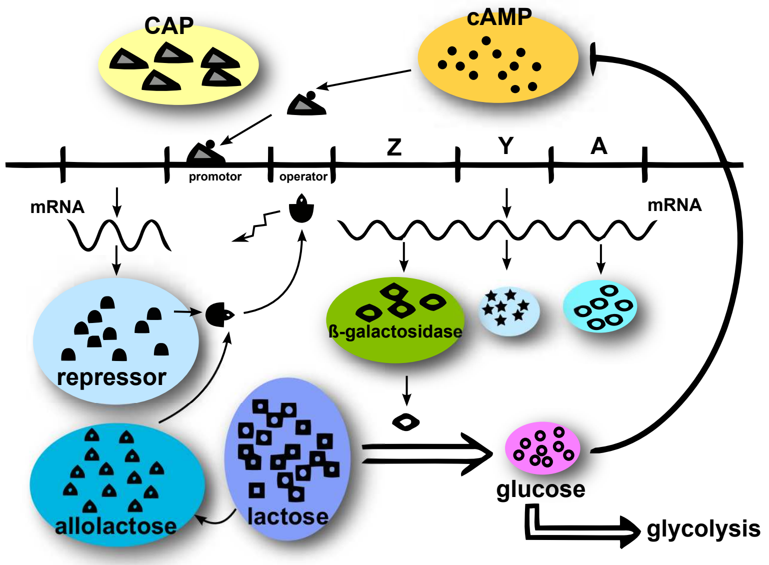


Figure 3.2: A diagram of *lac* operon regulation [34]

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 3.8. 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 the next section.

3.3 Modelling *Lac* operon regulation

In this section we formalise the *lac* operon regulation process by using sync-programs. Our model is inspired by the CCS model of the same process given in [77].

First we give a description of the *lac* operon regulation process. 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 [63].

The *lac* operon (see Figure 3.2) 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).

In order to give a formal sync-program model, we will fix the index set $I = \{lac, \beta, allo, op, repr, pos, glu\}$ 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_{repr} with $AP_{repr} = \{B1, B2, B3, Ballo\}$, 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} (Figure 3.3) 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. Note that due to fairness, 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 th 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.

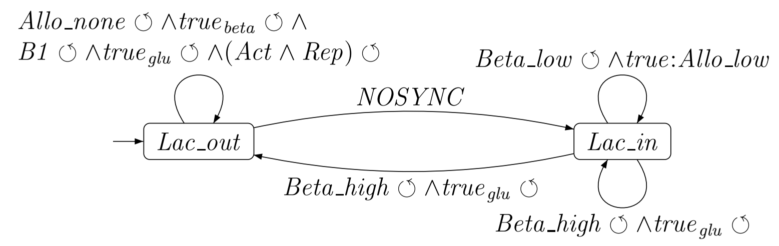

Allo_none ↺ ∧true_beta ↺ ∧
B1 ↺ ∧true_glu ↺ ∧(Act ∧ Rep) ↺
Beta_low ↺ ∧true:Allo_low
NOSYNC
Lac_out
Lac_in
Beta_high ↺ ∧true_glu ↺
Beta_high ↺ ∧true_glu ↺

Figure 3.3: P^I_{lac} – Lactose

In sync-automaton P^I_β (Figure 3.4) 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. When lactose never enters the cell, β-galactosidase level remains low.

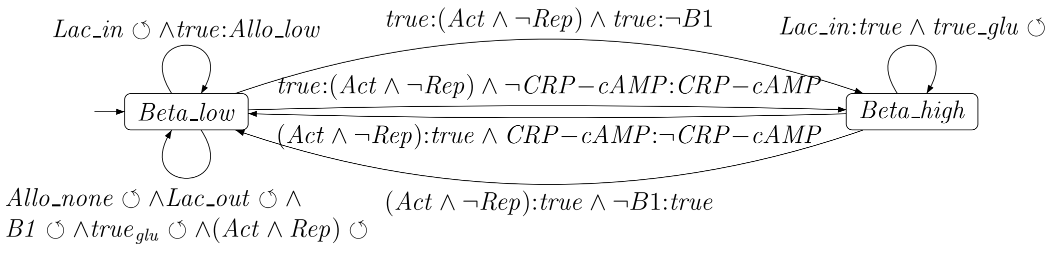

Lac_in ↺ ∧true:Allo_low
true:(Act ∧ ¬Rep) ∧ true:¬B1
Lac_in:true ∧ true_glu ↺
true:(Act ∧ ¬Rep) ∧ ¬CRP−cAMP:CRP−cAMP
Beta_low
Beta_high
(Act ∧ ¬Rep):true ∧ CRP−cAMP:¬CRP−cAMP
Allo_none ↺ ∧Lac_out ↺ ∧
B1 ↺ ∧true_glu ↺ ∧(Act ∧ Rep) ↺
(Act ∧ ¬Rep):true ∧ ¬B1:true

Figure 3.4: P^I_β – β-galactosidase

Allolactose P^I_{allo} (Figure 3.5) 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.

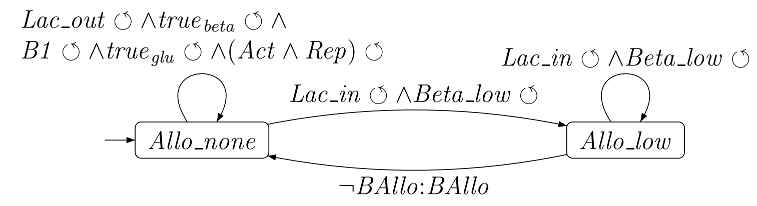


Figure 3.5: P^I_{allo} – Allolactose

The *lac* operon P^I_{op} (Figure 3.6) 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 in Figure 3.6) are determined by negative regulation P^I_{repr} while activation and inactivation (vertical moves in Figure 3.6) 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.

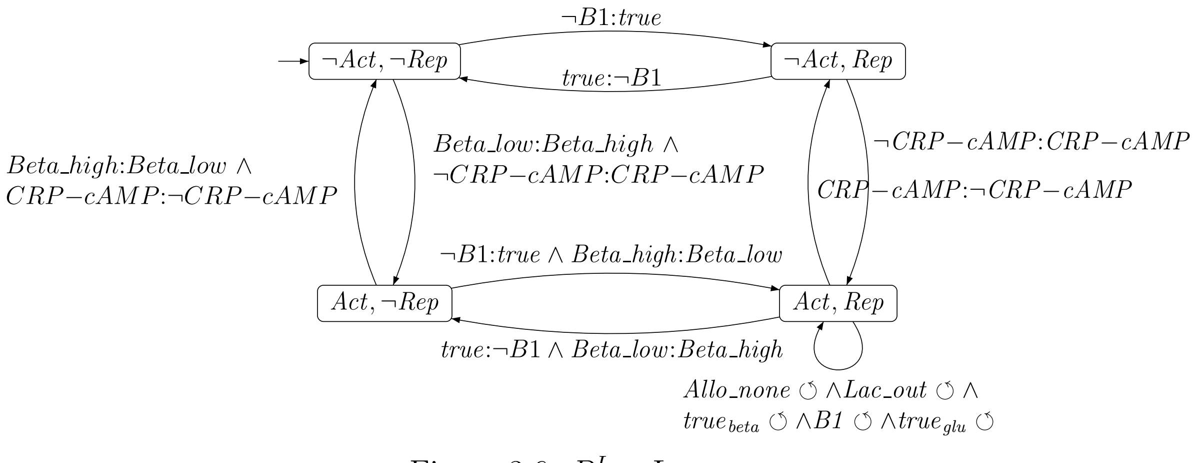


Figure 3.6: P^I_{op} – Lac operon

The *lac* repressor P^I_{repr} (Figure 3.7) has five states. After binding of *lac* repressor protein to O_1 site, it might bind either to 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} (Figure 3.8) works as follows. When glucose level is low, cAMP concentration will be increased. Coactivator CRP creates a complex

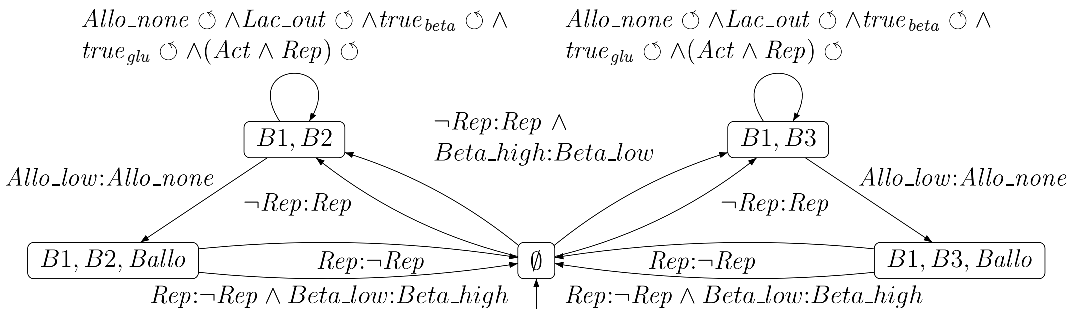

Allo_none ⥁ ∧*Lac_out* ⥁ ∧$true_{beta}$ ⥁ ∧ $true_{glu}$ ⥁ ∧(*Act* ∧ *Rep*) ⥁
Allo_none ⥁ ∧*Lac_out* ⥁ ∧$true_{beta}$ ⥁ ∧ $true_{glu}$ ⥁ ∧(*Act* ∧ *Rep*) ⥁
*B*1, *B*2
¬*Rep*:*Rep* ∧ *Beta_high*:*Beta_low*
*B*1, *B*3
Allo_low:*Allo_none*
¬*Rep*:*Rep*
¬*Rep*:*Rep*
Allo_low:*Allo_none*
*B*1, *B*2, *Ballo*
Rep:¬*Rep*
∅
Rep:¬*Rep*
*B*1, *B*3, *Ballo*
Rep:¬*Rep* ∧ *Beta_low*:*Beta_high*
Rep:¬*Rep* ∧ *Beta_low*:*Beta_high*

Figure 3.7: P^{I}_{repr} – Lac repressor protein (Negative regulation)

CRP-cAMP that binds to *lac* operon, stimulating the transcription. When the glucose concentration is increased, cAMP level will 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.

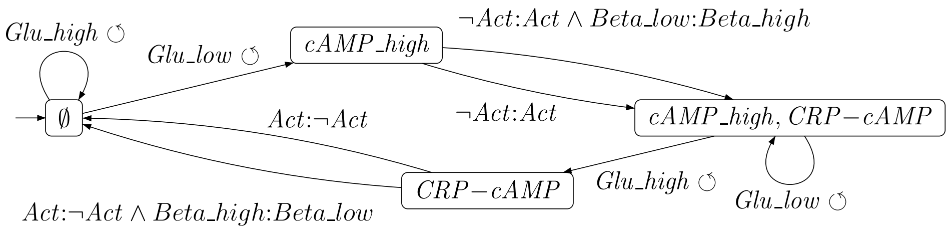


Figure 3.8: P^{I}_{pos} – CRP-cAMP (Positive regulation)

In P^{I}_{glu} (Figure 3.9) 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 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 rest unchanged.

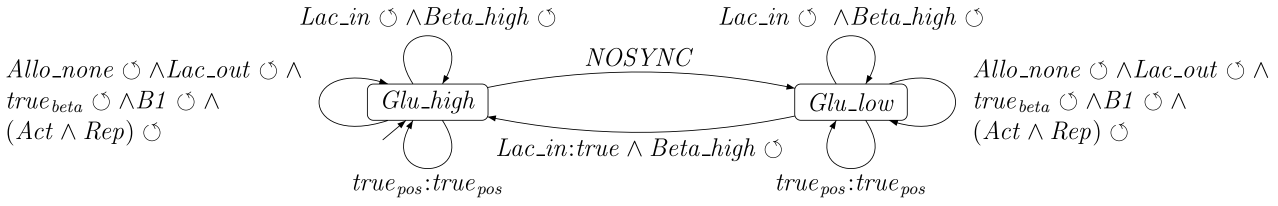

Lac_in ⥁ ∧*Beta_high* ⥁
Lac_in ⥁ ∧*Beta_high* ⥁
NOSYNC
Allo_none ⥁ ∧*Lac_out* ⥁ ∧ $true_{beta}$ ⥁ ∧*B1* ⥁ ∧ (*Act* ∧ *Rep*) ⥁
Glu_high
Glu_low
Allo_none ⥁ ∧*Lac_out* ⥁ ∧ $true_{beta}$ ⥁ ∧*B1* ⥁ ∧ (*Act* ∧ *Rep*) ⥁
Lac_in:*true* ∧ *Beta_high* ⥁
$true_{pos}$:$true_{pos}$
$true_{pos}$:$true_{pos}$

Figure 3.9: P^{I}_{glu} – Glucose

The sync-program describing the whole system is obtained by running all sync-automata in parallel $P^I = (S_0^I, P_{lac}^I || P_\beta^I || P_{allo}^I || P_{op}^I || P_{repr}^I || P_{pos}^I || P_{glu}^I)$. The set of initial states is a combination of the sets of initial states of respective sync-automata.

Chapter 4

Modular Verification of Sync-programs

In order to analyse the behaviour of a biological system we would like to verify properties of computations of a sync-program P^I representing the system. Assume that a property ϕ_J only regards a part of a system modelled as dynamic sync-program P^I, in particular the part involving only components from $J \subseteq I$. We should check satisfaction of ϕ_J on the semantics of P^I, but, in order to avoid the space explosion, we would like to check it on a smaller and more abstract semantics. Let us assume a projection operation $\upharpoonright$ that allows us to consider only a subset of the components of a sync-program. The smallest fragment of P^I we can consider is $P^J = P^I \upharpoonright J$, but in general any projection $P^{J'} = P^I \upharpoonright J'$ with $J \subseteq J' \subseteq I$ could allow us to obtain a smaller description of the system behaviour to be analysed by means of model checking and thus avoid the state explosion.

Considering a subprogram of P^I obtained through a projection means abstracting away some of the sync-automata and loosing the synchronisations with such sync-automata. This results in obtaining an overapproximation of the behaviour of P^I, since synchronisations imply constraints on the possible behaviours of the system. Hence, in order to ensure that the modular verification approach can be followed, we have to guarantee that properties holding in the semantics of a subprogram obtained through the projection operation hold also in the semantics of the whole program P^I.

The class of properties we will consider consists of the properties that can be expressed in ACTL$^-$. It has been shown in [52] that properties described with the temporal logic ACTL are preserved under composition, and this is suitable to be used in the context of modular verification. In fact, it consists of formulae with paths quantified only universally, hence formulae describing properties that must hold in all possible executions of a system. Since we consider projections that give overapproximations of the behaviours of systems, we can prove that if a property holds in all possible executions of a subprogram obtained through projection, then it holds also in the behaviour of the whole system.

This chapter is organised as follows. First we define the syntactical projection that allows to restrict the attention on a portion of a program called sync-subprogram. Then we prove that any behaviour of a sync-program P^I, when projected onto J, is present as a behaviour of the sync-subprogram P_J. This allows us to prove the property preservation for all properties from ACTL$^-$. We illustrate the usefulness of the approach by indicating what are the sync-subprograms that suffice for verification for some interesting biological properties of the *lac* operon regulation presented in Section 3.3.

4.1 Projections

In order to develop our modular verification technique we need two projection operations that allow to restrict the syntax and the semantics of a sync-program to the components that have an influence on the satisfaction of a property of interest.

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

Definition 4.1. Let $J \subseteq I$ be an index set and $J = \{j_1, \dots, j_k\}$. Let $P^I = (S_0^I, P_1^I || \dots || P_n^I)$ with $P_i^I = (S_i, S_i^0, R_i)$ for each $i \in I$.
Then $P^I \lceil J = (S_0^J, P_{j_1}^J || \dots || P_{j_k}^J)$ with $P_j^J = (S_j, S_j^0, R'_j)$ for each $j \in J$ where

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

Initial states are $S_0^J = S_{j_1}^0 \times \dots \times S_{j_n}^0$.

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 $P^I \lceil J$ is still a sync-program with index set J, hence it can be also denoted by P^J.

As an example, on Figure 4.1 there is the sync-program $P^{\{1,2,3\}} \lceil \{1, 2\}$, that is $P^{\{1,2,3\}}$ from Figure 3.1 after projection onto $\{1, 2\}$.

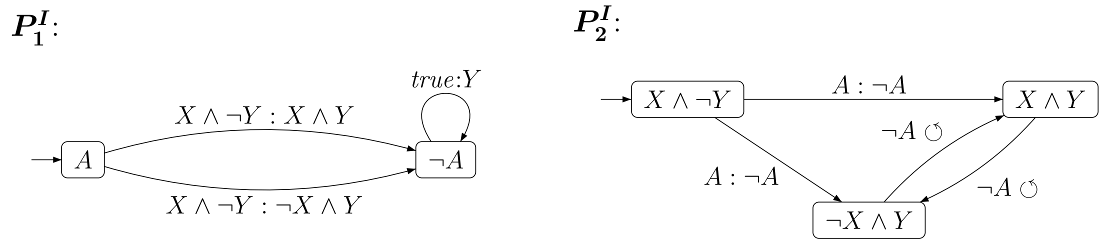

P_1^I:
$true{:}Y$
$X \wedge \neg Y : X \wedge Y$
A
$\neg A$
$X \wedge \neg Y : \neg X \wedge Y$
P_2^I:
$X \wedge \neg Y$
$A : \neg A$
$X \wedge Y$
$\neg A \circlearrowleft$
$A : \neg A$
$\neg A \circlearrowleft$
$\neg X \wedge Y$

Figure 4.1: A projection of a sync-program $P^{\{1,2,3\}}$ onto $\{1, 2\}$.

In the following we define the *semantic projection* of transitions and paths.

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: $(s, l, t)\lceil J = (s\lceil J, l \cap J, t\lceil J)$.

In order to define the path projection, we need an auxiliary definition of J-blocks in a path in an I-structure. 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: $\pi\lceil J = (Bl^1\lceil J, l^1 \cap J, Bl^2\lceil J, l^2 \cap J, \ldots)$.

4.2 Path Preservation

In this section we give some lemmas about the relationships between the semantics of a dynamic sync-program and the semantics of a sync-subprogram obtained through syntactic projection.

Given an index set I, a subset $J \subseteq I$ and a program P^I, the aim of these lemmas is to show that a fullpath in the semantics P^I that is relevant for the verification of a property concerning sync-automata from J has always a corresponding fullpath in the semantics of the sync-subprogram we obtain through syntactic projection.

To be able to perform the verification on the semantics of a sync-subprogram, 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.

Since computation of a sync-program has been defined as a fair fullpath in its semantics, we need to show that every fullpath in the semantics of P^I projected onto J is a fullpath in the semantics of $P^J = P^I\lceil J$. Firstly, we prove that every path in $\mathcal{M}_I$ projected onto J is a path in $\mathcal{M}_J$. For that, the basic building stone is the following projection preservation lemma for transitions.

Lemma 4.2 (Transition projection)**.** *Let I be an index set and $\mathcal{M}_I = (\mathcal{S}_I, \mathcal{S}_I^0, \mathcal{R}_I)$ 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 $\mathcal{M}_J = (\mathcal{S}_J, \mathcal{S}_J^0, \mathcal{R}_J)$ is the semantics of sync-program $P^J = P^I\lceil J$.*

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 $\mathit{types}(MOV) = l$. We need to show that in $\mathcal{M}_J$ there is a synchronisation MOV' with support $(s\lceil J, t\lceil J)$ such that $\mathit{types}(MOV') = l \cap J$. 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, $\mathit{types}(MOV') = l \cap J$. 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 4.3 (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 $P^J = P^I\lceil J$.*

Proof. Let $\pi = (Bl^1, l^1, Bl^2, l^2, \ldots)$ be a path in $\mathcal{M}_I$ 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 $\mathcal{M}_J$. □

With the help of the preceding lemma we can prove that the computation being a fullpath is preserved.

Lemma 4.4 (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 (Definition 3.8) follows that every fair path is infinite, i.e. it is a fullpath. □

Note that the choice made in Section 3.2 to define a computation as a fair fullpath was a necessary one. If we had not decided to include the fairness requirement, the computation preservation would not necessarily hold.

In particular, with computation defined as a fullpath, a violation of the desired computation preservation arises when an independent partition P of sync-program P^I exists whose sync-automata can be executed forever, while not allowing the execution of other 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 a computation of P^J.

Note that every fair path is infinite since 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. We believe that 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.

4.3 Logic

Now we exploit the fullpath projection lemma to prove that behavioural properties expressed in a suitable logic that hold in the semantics of a sync-subprogram also hold in the semantics of the original sync-program. The logic we will consider is a fragment of the Computation Tree Logic CTL.

Following Attie and Emerson [7], 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 4.5. 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 $A[f\ U\ g]$ 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$^-$: $AFf \equiv A[\mathit{true}\ U\ f]$ and $AGf \equiv A[f\ U_w\ \mathit{false}]$.

Properties expressible by ACTL$^-$ formulae represent a significant class of properties investigated in the systems biology literature as identified in [73], 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"*).

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 an I-structure follows. Note that only fair fullpaths are considered.

Definition 4.6. *Semantics of ACTL$^-$.* We define $\mathcal{M}_I, s \models f$ (resp. $\mathcal{M}_I, \pi \models f$) meaning that f is true in structure $\mathcal{M}_I$ at state s (resp fair fullpath π). We define $\models$ inductively:

- $\mathcal{M}_I, s \models \mathit{true}$. $\mathcal{M}_I, s \not\models \mathit{false}$. $\mathcal{M}_I, s \models p$ iff $s(p) = \mathit{tt}$.
  $\mathcal{M}_I, s \models \neg p$ iff $s(p) = \mathit{ff}$.
- $\mathcal{M}_I, s \models f \wedge g$ iff $\mathcal{M}_I, s \models f$ and $\mathcal{M}_I, s \models g$.
  $\mathcal{M}_I, s \models f \vee g$ iff $\mathcal{M}_I, s \models f$ or $\mathcal{M}_I, s \models g$.
- $\mathcal{M}_I, s \models Af$ iff for every fair fullpath $\pi = (s, l^1, \ldots)$ in $\mathcal{M}_I$:
  $\mathcal{M}_I, \pi \models f$.
- $\mathcal{M}_I, \pi \models f$ iff $\mathcal{M}_I, s \models f$, where s is the first state of π
- $\mathcal{M}_I, \pi \models f \wedge g$ iff $\mathcal{M}_I, \pi \models f$ and $\mathcal{M}_I, \pi \models g$.
  $\mathcal{M}_I, \pi \models f \vee g$ iff $\mathcal{M}_I, \pi \models f$ or $\mathcal{M}_I, \pi \models g$.
- $\mathcal{M}_I, \pi \models f\ U\ g$ iff there exists $m \in \mathbb{N}$ such that $\mathcal{M}_I, \pi^m \models g$
  and for all $m' < m$: $\mathcal{M}_I, \pi^{m'} \models f$.
- $\mathcal{M}_I, \pi \models f\ U_w\ g$ iff for all $m \in \mathbb{N}$, if $\mathcal{M}_I, \pi^{m'} \not\models g$
  for all $m' < m$ then $\mathcal{M}_I, \pi^m \models f$.

4.4 Property Preservation Theorem

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

Theorem 4.7 (Property preservation). *Let $J \subseteq I$ be an index set, s an I-state and f an ACTL^-_J formula. If $\mathcal{M}_J, s\lceil J \models f$ then $\mathcal{M}_I, s \models f$.*

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 $\mathcal{M}_J, s\lceil J \models p$ iff $\mathcal{M}_I, s \models p$. 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.

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

2. $\mathcal{M}_J, \pi\lceil J \vDash [g\ U\ h]$. 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 $\mathcal{M}_I, \pi \vDash g\ U\ h$.

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

$f = A[g\ U\ h]$. 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 $s \vDash A[g\ U\ h]$. □

4.5 Application to *Lac* Operon Regulation

In this section we investigate some known properties of the *lac* operon regulation and show their representation as ACTL$^-$ formulae.

By inspection of the sync-program, for each property we identify the minimal sync-subprogram that makes the property satisfied. This is done by starting from the fragment that contains all sync-automata whose atomic propositions are mentioned in the property. In the case that the verification finishes with a negative result, iteratively we try to add more sync-automata to the fragment in consideration. Once the satisfaction of the property is established on a fragment, then by the property preservation theorem its satisfaction in whole model of the *lac*-operon regulation is guaranteed.

For the purpose of this chapter, in this section we only indicate what is the fragment of the model that suffices for the verification of the property. Any verification method can be applied to the identified fragment in order to investigate the property in question and the result of the verification is assured. The practical verification making use of a model checker is presented in the next chapter.

The property (P1) *"The allolactose bound to the repressor implies that the operon is repressed"* represents the exclusion type as identified in [73]. The formula

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

is verifiable on the semantics of $P^{op,repr}$.

A slightly more complicated formula is needed to express that (P2) *"The increase of allolactose concentration can only be mediated by β-galactosidase in low concentration"*. The formula

$$AG(Allo\_none \wedge Beta\_high \rightarrow A[\neg Allo\_low\ U\ \ Beta\_low]) \tag{P2}$$

is true in the semantics of $P^{allo,\beta}$.

The oscillation property (P3) *"While lactose is inside the cell, the operon will necessarily oscillate between a repressed and an unrepressed state"* is expressed as follows. Consider formula (P3a) that represents the property: if operon is repressed in the current state, then operon will be eventually expressed unless lactose comes into the cell:

$$Rep \rightarrow (\ AF(\neg Rep)\ \vee\ A[AF(\neg Rep)\ U\ Lac\_in]\ ) \tag{P3a}$$

Denote (P3b) an analogous formula, claiming that a repressed state is always reachable from an unrepressed state modulo the entrance of lactose into the cell:

$$\neg Rep \rightarrow (\ AF(Rep)\ \vee\ A[AF(Rep)\ U\ Lac\_in]\ ) \tag{P3b}$$

The property (P3) therefore consists of global satisfaction of both the preceding formulae. The formula

$$AG((P3a) \wedge (P3b)) \tag{P3}$$

is true in $\mathcal{M}_I$, but the verification in $\mathcal{M}_{lac,op}$ fails. Thus the property preservation theorem cannot be invoked. However, by inspecting the model we can understand that by taking into consideration two other components that are not mentioned in the formula we could succeed in verification. Indeed, the formula is true in the semantics of $P^{\beta,lac,op,repr}$. Note the important role of fairness for satisfaction of this property that requires that each sync-automaton is executed infinitely many times. It should be noted that the present formula differs from formula Osc1 in Section 2.8 which requires oscillations along one path. Formula P3, instead, declares the oscillatory behaviour along all paths . However, we consider a weakening by allowing an action discharge the oscillations.

A property (P4) that demonstrates the correctness of the model of the *lac* operon regulation can be stated as follows: *"When the glucose concentration drops and lactose is inside the cell, the* lac *operon will eventually be fully expressed"*. Encoded as an $\mathrm{ACTL}^{-}_{glu,lac,op}$ formula

$$AG(Glu\_low \wedge Lac\_in \rightarrow AF(Act \wedge \neg Rep)) \tag{P4}$$

it holds in $\mathcal{M}_I$ but cannot be verified for any of the subprograms of P^I, as it depends on activities of every component in the model.

We can prove in a modular manner the properties stating the regulation subsystems work, both the negative and the positive ones.

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

$$AG(Lac\_in \wedge Rep \rightarrow AF(\neg Rep)) \tag{P5}$$

can be verified on the semantics of $P^{\beta,lac,op,repr}$. Note that the presence of lactose inside the cell is essential for the negative regulation.

The property (P6) *"When glucose concentration is low, the* lac *operon will eventually be activated"* written in ACTL$^-$ as

$$AG(Glu\_low \wedge \neg Act \rightarrow AF(Act)) \tag{P6}$$

can be verified on the semantics of $P^{\beta,pos,glu,op}$, marking the importance of the low level of glucose in the positive regulation.

Chapter 5

Modular Verification in Practice

In order to do the modular verification of sync-programs practically, we exploit the tool NuSMV. In order to use NuSMV we need to translate the sync-program of interest to the input language of the tool. Our translation of sync-programs, however, introduces auxiliary states that must not be considered when evaluating logical properties. Therefore we need to modify the properties so that they do not take these intermediate states into account.

We show formally the correctness of our translation approach, that is we prove that the problem of verifying the translated properties on a translation of a sync-program is equivalent to the problem of verifying the original property on the original sync-program for any ACTL^- property.

Then the correctness of the practical modular verification is guaranteed by the combination of the Property preservation theorem (Theorem 4.7) and the Verification correspondence theorem below. Indeed, when one successfully verifies an encoded property on the translation of the model fragment in NuSMV, the satisfaction of the corresponding property in the original model fragment holds and therefore the property holds also in the whole model.

5.1 Translation of Sync-Programs to Sync-Skeletons

In this section we define a function *transl* that translates a sync-program into a formalism that can be implemented directly in NuSMV. The target formalism is the synchronisation skeletons [27]. Note that by passing from sync-programs to synchronisation skeletons, the underlying concurrency model is changed. Since the latter formalism does not feature synchronisation in the sense that there is no language primitive that permits two or more processes to perform a move simultaneously.

Our solution is to add auxiliary states, and translate each transition of a sync-program to a sequence of transitions in the synchronisation skeletons.

Denote by sem and sem_{skel} the functions that assign to a sync-program its semantics and to a synchronisation skeleton program its semantics, respectively. We specify functions $transl$ that takes as an input a sync-program P^I and outputs a synchronisation skeleton SP^I and formula translation function $fortran$ such that

$$sem(P^I) \vDash \phi \text{ iff } sem_{skel}(SP^I) \vDash fortran(\phi)$$

for all ACTL$^-$ properties.

For the rest of the section, we assume an arbitrary but fixed ordering on the index set I denoted by the injective function $o : I \to N$.

Now we define the translation function compositionally.

Sync-program Let $P^I = (P_1^I || \dots || P_n^I, S_0^I)$ be a sync-program. Then the result of translation $transl(P^I)$ is a synchronisation skeleton program composed of a parallel composition of synchronisation skeletons, representing a translation of all sync-automata in P^I.

Sync-automata A sync-automaton $P_i = (S_i, S_0^i, R_i)$ is translated to a synchronisation skeleton $transl(P_i^I) = P_i^* = (S_i^*, S_{i,0}^*, R_i^*)$ which is defined as follows.

We consider an arbitrary but fixed ordering on multiple moves of the sync-automaton denoted by the injective function $moveord : R_i \mapsto N$.

- **States** If AP_i is the set of atomic propositions used for construction of S_i, then $transl(AP_i) = AP_i^*$ used for construction of S_i^* is the union of the following sets

  1. AP_i
  2. $\{to_a \mid a \in AP_i\}$
  3. $\{move_k \mid 0 < k < max_{mov}\}$, where max_{mov} is the maximal number of moves between the same states in P_i^I
  4. $\{true_i, to\_true_i\}$

  The set AP_i^* of atomic propositions of the synchronisation skeleton P_i^* includes all atomic propositions of the sync-automaton P_i^I and, in addition, it contains for each of these atomic propositions a a proposition to_a whose inclusion in a state expresses the fact that there is a successor state where a holds. The atomic proposition $move_k$ is to distinguish intermediate states. The purpose of these propositions will be clear once we define the set of states of P_i^*. Atomic propositions $true_i$ and to_true_i are to mark the set of all states and the set of intermediate states, respectively.

  The set of all states $S_i^* \subseteq \mathcal{P}(AP_i^*)$ is the smallest set for which

1. for all $s'_i \in S^*_i$ we have $s'_i(true_i) = tt$
2. for all $s'_i \in S^*_i$ we have $s'_i(to\_true_i) = tt$ iff $s'_i(to_a) = tt$ for some a
3. for each $s_i \in S_i$ there is a $s'_i \in S^*_i$ so that
   (a) for all $a \in AP_i$, $s'_i(a) = tt$ iff $s_i(a) = tt$
   (b) for all $a \in AP_i$, $s'_i(to_a) = ff$
   (c) for all $k \in N$, $s'_i(move_k) = ff$

   We construct a mapping $basicState : s_i \mapsto s'_i$.
4. for every move $s_i \xrightarrow{sc_i} t_i \in R_i$ there is a state $s'_i \in S^*_i$ such that
   (a) for all $a \in AP_i$, $s'_i(a) = tt$ iff $s_i(a) = tt$
   (b) for all $a \in AP_i$, $s'_i(to_a) = tt$ iff $t_i(a) = tt$
   (c) $s'_i(move_k) = tt$ iff $k = moveord(s_i \xrightarrow{sc_i} t_i)$ and $k > 0$

   We construct a mapping $toState : (s_i \xrightarrow{sc_i} t_i) \mapsto s'_i$.

The set of states S^*_i of synchronisation skeleton SP^*_i consists of two disjoint sets. The first set is the set of all *basic states* which corresponds one to one to the set of states of the sync-automaton P^I_i. For each state $s_i \in S_i$ there is a state $s'_i \in S^*_i$ such that all the atomic propositions in s_i have the same value in s'_i. All "to"-propositions and "move" propositions are false in these states. The mapping *basicState* maps s_i to s'_i.

The second group of states is the set of all *intermediate states.* For every move $s_i \xrightarrow{sc_i} t_i$ in P^I_i there is a state in s'_i representing that the corresponding move has been executed but the corresponding transition has not been concluded yet. The state s'_i encodes the move by having true all the atomic propositions from state s_i and satisfying to_a for all atomic propositions a from t_i. For convenience, we construct a mapping $to : \mathcal{P}(AP_i) \rightarrow \mathcal{P}(AP^*_i)$ which assigns to a set of atomic propositions the set of atomic propositions from AP^*_i such that there is a to_a for each a in the source set, i.e. $to(A) = \{to_a \mid a \in A\}$. Therefore intuitively, state s'_i can be thought of as $s_i \wedge to(t_i)$. Note that for every move between two states in the sync-automaton we need a distinct intermediate state in the synchronisation automaton. This is ensured by atomic propositions $move_k$, each of which is true exclusively in one intermediate state corresponding to a move, when multiple moves are present between the same pair of states. The mapping *toState* maps the move $s_i \xrightarrow{sc_i} t_i$ to the state s'_i.

Moreover, the proposition $true_i$ holds in all states of S^*_i and to_true_i holds only in the intermediate ones.

- **Initial states** $S^*_{i,0} = \{basicState(s_i) \mid s_i \in S^i_0\}$
  The set of initial states is the set of basic states corresponding to initial states of P^I_i.

- **Moves** The set of moves R_i^* is the union of the following sets

$$\begin{array}{rcll} R_1 & = & \{ & s'_j \xrightarrow{\bigwedge_{m \in \{1,\dots,k\}}(A_{i_m} \wedge to(B_{i_m})) \wedge \bigwedge_{m \in \{k+1,\dots,n\}} A_{i_m} \wedge \bigwedge_{w \in I \setminus \{i_1,\dots,i_k\}} \neg to\_true_w} st'_j \\ & & | & s_j \xrightarrow{\bigwedge_{m \in \{1,\dots,n\}} A_{i_m} : B_{i_m}} t_j \in R_j \\ & & & \text{and } o(i_1) < \dots < o(i_k) < o(j) < o(i_{k+1}) < \dots < o(i_n) \\ & & & \text{and } s'_j = basicState(s_j) \text{ and } st'_j = toState(s_j \xrightarrow{\bigwedge A_{i_m} : B_{i_m}} t_j)\} \end{array}$$

$$\begin{array}{rcll} R_2 & = & \{ & st'_j \xrightarrow{\bigwedge_{m \in \{1,\dots,k\}} B_{i_m} \wedge \bigwedge_{m \in \{k+1,\dots,n\}}(A_{i_m} \wedge to(B_{i_m})) \wedge \bigwedge_{w \in I \setminus \{i_1,\dots,i_k\}} \neg to\_true_w} t'_j \\ & & | & s_j \xrightarrow{\bigwedge_{m \in \{1,\dots,n\}} A_{i_m} : B_{i_m}} t_j \in R_j \\ & & & \text{and } o(i_1) < \dots < o(i_k) < o(j) < o(i_{k+1}) < \dots < o(i_n) \\ & & & \text{and } t'_j = basicState(t_j) \text{ and } st'_j = toState(s_j \xrightarrow{\bigwedge A_{i_m} : B_{i_m}} t_j)\} \end{array}$$

The set of moves of synchronisation skeleton P_i^* is constructed in such a way that only one synchronisation of the translated sync-program is simulated at a time. A synchronisation is performed by a series of separate moves of participating synchronisation skeletons, one skeleton at a time, in a specific order o defined globally.

First, all the participating synchronisation skeletons one-by-one following the order conduct the transition to the respective intermediate states. After finishing the first round, the second move leads to the respective target states. Throughout the whole operation all non-participating synchronisation skeletons cannot perform any move. Moreover, the synchronisation skeletons 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 moves of the corresponding skeleton. The first move, from the source to the intermediate state has to wait until all the suitable moves of skeletons preceding it in the order have been performed, and no other move has been performed in the system.

The formalisation in the definition can be understood with the following intuition in mind

- all sync-automata: I, total order o on I assumed without loss of generality
- I includes (not necessarily only) $i_1, \dots, i_n$ and a distinct j
- sync-automata performing the synchronisation: $P_{i_1}^I, \dots, P_{i_n}^I$ and currently performing the move is P_j^I

A move $s_j \xrightarrow{sc_j} t_j$ of P_j^I is translated to two moves of P_i^*. First, move from $basicState(s_j)$ to $toState(s_j \xrightarrow{sc_j} t_j)$ has to wait for all the moves of synchronisation skeletons that are earlier in the order, precisely following the its synchronisation condition. This is expressed by the first part of the synchronisation condition $\bigwedge_{m\in\{1,\dots,k\}}(A_{i_m} \wedge to(B_{i_m}))$. The other part of the condition says that the remaining skeletons, participating in this synchronisation but later in the order have to be ready for the transition. Moreover, all modules except for those participating in the synchronisation that precede the current module in the order, must not be in a to-state, as expressed by the following condition $\bigwedge_{m\in k+1,\dots,n\}} A_{i_m} \wedge \bigwedge_{w\in I\setminus\{i_1,\dots,i_k\}} \neg to\_true_w$.

As for the second move, which leads from $toState(s_j \xrightarrow{sc_j} t_j)$ to $basicState(s_j)$, the module has to check that all the preceding modules have arrived to the target state and the following modules are the only ones waiting for the execution, i.e. in a to-state:
$\bigwedge_{m\in\{1,\dots,k\}} B_{i_m} \wedge \bigwedge_{m\in k+1,\dots,n\}} A_{i_m} \wedge to(B_{i_m}) \wedge \bigwedge_{w\in I\setminus\{i_1,\dots,i_k\}} \neg to\_true_w$.

Now we give an estimation of the size of the translated program. Consider the sync-program $P^I = (P_1^I||\dots||P_n^I, S_0^I)$. The synchronisation skeleton program $transl(P^I)$ is composed of n synchronisation skeletons. If P_i^I is composed of n_s states and n_m moves, then $transl(P_i^I)$ has $n_s + n_m$ states, since there is one auxiliary state for each original move, and $2n_m$ moves, since every move is translated into two moves leading through the auxiliary state.

Let m_s be the number of states and m_t be the number of transitions in the LTS representing the semantics of P^I. If k_{syn} is the maximal size of a synchronisation in P^I, then the semantics of $transl(P^I)$ consists of at most $m_s + k_{syn}.m_t$ states and at most $k_{syn}.m_t$ transitions.

Moreover, for each state in the semantics of the translation not corresponding to a state in the semantics of the original program, i.e. in an auxiliary semantic state, there is only one successor state. As a consequence, the verification problem of the translated system grows asymptotically as fast as the verification problem of the original system.

An example of the translation will be presented in Section 5.4 where we translate the Program P^I from Section 3.3 representing the regulation of the *lac* operon.

5.2 Translation of Properties

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 state formula has to be implied by a formula $\neg toTrue$, where $toTrue$ is defined as $\bigvee_{i\in I} toTrue_i$ and $toTrue_i = \bigvee_{a\in AP_i} to_a$.

Formulae describe properties in terms of atomic propositions. The translation of a sync-program as defined above is composed of two types of states. Basic states are the ones directly corresponding to states of the original sync-program. All the atomic propositions from the sync-program have exactly the same value in the corresponding state in the translation. Any property referring to atomic propositions of a basic state is therefore evaluated in the same way as it is in the corresponding state of the sync-program. On the other hand, there are auxiliary intermediate states, whose original atomic propositions do not correspond to any original state of the sync program. Therefore, these states should be "skipped" when evaluating properties of translated sync-programs.

We introduce the formula translation function $fortran : ACTL^- \to ACTL^-$ that modifies a formula in such a way that the intuition described above holds. Note that the language of the formulae changes, while the first formula is an ACTL$^-$ formula built over the set of atomic propositions $AP = \bigcup_{i\in I} AP_i$, it is translated to an ACTL$^-$ formula built over a different set of atomic propositions, namely $AP^* = \bigcup_{i\in I} AP_i^*$.

Formalising the requirements mentioned above, the function *fortran* has the following two characteristics:

- in a basic state, the translation of a formula f is true iff and only if f is true in the corresponding state of $sem(P^I)$
- in a to-state the translation of an until-formula is evaluated exactly like in the successor state.

Now we define a formula translation function that meets the above specification.

Definition 5.1. The *formula translation* function *fortran* :ACTL$^-$ $\to$ACTL$^-$ is defined as follows:

$$\begin{array}{lcl}
fortran(p) & = & p \text{ iff } p \text{ is an atomic proposition or its negation}\\
fortran(f \wedge g) & = & fortran(f) \wedge fortran(g)\\
fortran(f \vee g) & = & fortran(f) \vee fortran(g)\\
fortran(A[f\ U\ g]) & = & A[(\neg toTrue \to fortran(f))\ U\ (\neg toTrue \wedge fortran(g))]\\
fortran(A[f\ U_w\ g]) & = & A[(\neg toTrue \to fortran(f))\ U_w\ (\neg toTrue \wedge fortran(g))]
\end{array}$$

Note that the translation function does neither add nor remove atomic proposition from the formula. Hence, the first point of the specification is satisfied. In order to fulfil the second requirement, the intuition behind the modifications in a until-formula can be understood from the fixed-point characterisation of formula $A[g\ U\ h]$: If $f = A[g\ U\ h]$ then $f \leftrightarrow h \vee (g \wedge AXf)$ where AX is the operator that says its argument formula holds in all successor states. It is easy to see that the evaluation of the translated formula

$$\mathit{fortran}(A[g\ U\ h]) = (\neg \mathit{toTrue} \wedge h) \vee ((\neg \mathit{toTrue} \rightarrow g) \wedge AX\mathit{fortran}(A[g\ U\ h]))$$

in an intermediate state is true if and only if $\mathit{fortran}(A[g\ U\ h])$ is true in all successor states. Effectively, the result of evaluation of the formula $\mathit{fortran}(A[g\ U\ h])$ does not depend on the intermediate state. For the formula $f = A[g\ U_w\ h]$ the fixed point characterisation is the same as for $f = A[g\ U\ h]$, the difference is that for AU the least fixed-point is considered while for AU_w the greatest one. The intuitive difference is that evaluation of the second disjunct can be postponed forever, admitting an infinite behaviour. However, the translation follows the same structure as the one for the case of AU.

To get an intuition about how the translation function works, we exemplify it on two formula patterns that appear frequently in the verification practice.

$$\begin{aligned}
\mathit{fortran}(AG(p)) &= \mathit{fortran}(A[p\ U_w\ \mathit{false}]) \\
&= A[(\neg \mathit{toTrue} \rightarrow \mathit{fortran}(p))\ U_w\ (\neg \mathit{toTrue} \wedge \mathit{fortran}(\mathit{false}))] \\
&= A[(\neg \mathit{toTrue} \rightarrow p)\ U_w\ \mathit{false}] \\
&= AG(\neg \mathit{toTrue} \rightarrow p)
\end{aligned}$$

$$\begin{aligned}
\mathit{fortran}(AF(p)) &= \mathit{fortran}(A[\mathit{true}\ U\ p]) \\
&= A[\mathit{fortran}(\mathit{true})\ U\ (\neg \mathit{toTrue} \wedge \mathit{fortran}(p))] \\
&= AF(\neg \mathit{toTrue} \wedge p)
\end{aligned}$$

5.3 Verification Problems Correspondence

In this section we prove the correspondence of the verification of a formula on a sync-program and the verification of the formula translation on the sync-program translation.

To this aim, we first show the operational correspondence of the two semantics, namely that a computation in the semantics of a sync-program corresponds to a computation in its translation. In particular, the two computations coincide on the basic states when considering the original atomic propositions.

We start with showing that a transition of a sync-program corresponds to a sequence of transitions in the corresponding synchronisation skeleton program. We override the definition of *basicState* to work also on I-states: $\mathit{basicState}(s)$ is defined as $\bigcup_{i \in I} \mathit{basicState}(s\lceil i)$.

Lemma 5.2 (Transition Correspondence)**.** *Let P^I be a sync-program with $\mathit{sem}(P^I) = (\mathcal{S}, \mathcal{S}_0, \mathcal{R})$, its translation $SP^I = \mathit{transl}(P^I)$ with $\mathit{sem}_{\mathit{skel}}(SP^I) = (\mathcal{S}^*, \mathcal{S}_0^*, \mathcal{R}^*)$. We* *have that* $(s, l, t) \in \mathcal{R}$ *where* $l = \{l_1, ..., l_k\}$ *iff there is a sequence of transitions from* $s^0 = basicState(s)$ *to* $s^{2n} = basicState(t)$*, i.e. there are* $s^1, \ldots, s^{2n-1} \in \mathcal{S}^*$ *such that* $(s^0, l_1, s^1), (s^1, l_2, s^2), ..., (s^{2n-1}, l_n, s^{2n})$ *are in* $\mathcal{R}^*$*.*

Proof. "$\Rightarrow$" Let $(s, l, t) \in \mathcal{R}$ and $l = \{l_1, ..., l_n\}$. By the definition of semantics of sync-program P^I there is a synchronisation MOV with support (s, t) and labels of the participating automata are $\{l_1, ..., l_n\}$ with $o(l_1) \leq ... \leq o(l_n)$.

First we prove that if $s^0 = basicState(s)$ and for all $0 < i \leq n$ states $s^i \in \mathcal{S}^*$ are as follows

- $s^i \lceil j = s^{i-1} \lceil j$ if $0 < j \leq n$ and $j \neq i$
- $s^i \lceil j = toState(MOV \lceil l_j)$ if $j = i$

then $(s^{i-1}, l_i, s^i) \in \mathcal{R}^*$ for all $0 < i \leq n$.

The proof proceeds by (complete) mathematical induction on i, we show that a move of $SP^I_{l_i}$ is executable in s^{i-1} and it leads to the state s^i. Let $s_{l_i} \xrightarrow{\wedge_{m\in\{1,...,i-1,i+1,...,n\}} A_{l_m}:B_{l_m}} t_{l_i}$ be the move of automaton $P^I_{l_i}$ in MOV. From the definition of translation we know that there is a move $s'_{l_i} \xrightarrow{\bigwedge_{m\in\{1,...,i-1\}} A_{l_m} \wedge to(B_{l_m}) \wedge \bigwedge_{m\in\{i+1,...,n\}} A_{l_m} \wedge \bigwedge_{w\in I\setminus\{l_1,...,l_{i-1}\}} \neg to\_true_w} st'_{l_i}$ in $SP^*_{l_i}$ and $s'_{l_i} = basicState(s_{l_i})$ and $st'_{l_i} = toState(s_{l_i} \xrightarrow{\bigwedge A_{i_m}:B_{i_m}} t_{l_i})$. Now we show that this move is enabled in s^{i-1}, that is we show that $s^{i-1} \lceil l_i = s'_{l_i}$ and the enabling condition is satisfied. Since $s^0 \lceil l_i = basicState(s \lceil l_i)$ and no move of type $l_1, ..., l_{i-1}$ changes satisfaction of atomic propositions from A_{l_i}, we have that $s^0 \lceil l_i = s^{i-1} \lceil l_i = s'_{l_i}$. By the induction hypothesis for j from 0 to $i-1$ we have that $s \lceil l_j = toState(s_{l_j} \xrightarrow{\bigwedge A_{i_m}:B_{i_m}} t_{l_j})$ and by the definition of s^{i-1} also $s^{i-1} \lceil l_j = s^j \lceil l_j$ which satisfies A_{l_j} and $to(B_{l_j})$. Since for all $i < j \leq n$ holds $s^0 \lceil l_j = s^{i-1} \lceil l_j$ then s^{i-1} satisfies A_{l_j} for all $j \in \{i+1, ..., n\}$. By the semantics of sync-automata and the translation all automata not having a move in MOV stay idle and s^0 satisfies $\neg to\_true_j$ for all $j \in I - \{l_1, ..., l_n\}$, it holds also in s^{i-1}. Moreover $\neg to\_true_j$ holds for all $i < j \leq n$ since it was so in s^0 and $s^0 \lceil l_j = s^{i-1} \lceil l_j$. Hence the enabling condition is satisfied and the move of $SP^I_{l_i}$ can be executed in s^{i-1}. Furthermore, from the definition of s^i the execution of move of $SP^I_{l_i}$ in s^{i-1} leads to s^i. Therefore then $(s^{i-1}, l_i, s^i) \in \mathcal{R}^*$ for all $0 < i \leq n$.

Second, we prove that if $s^{2n} = basicState(t)$ and for all $n < i < 2n$ states $s^i \in S^*$ are as follows

- $s^i \lceil j = s^{i-1} \lceil j$ if $0 < j \leq n$ and $j \neq i - n$
- $s^i \lceil j = basicState(t_{l_j})$ if $j = i - n$ and $s_{l_j} \xrightarrow{sc_{l_j}} t_{l_j} \in MOV$

then $(s^{i-1}, l_{i-n}, s^i) \in R^*$ for all $n < i \leq 2n$.

The proof is analogous to the case of $0 < i \leq n$, exploiting the definition of state s^i and the fact that by definition of a translation on step i there is a move whose condition is enabled and lead to state $\mathit{basicState}(t_{l_{i-n}})$.

"$\Leftarrow$" Suppose that in $\mathcal{S}^*$ there are states s^0 and s^{2n} such that $s^0 = \mathit{basicState}(s)$ for some $s \in \mathcal{S}$ and $s^{2n} = \mathit{basicState}(t)$ for some $t \in \mathcal{S}$ and a sequence of transitions from s^0 to s^{2n}, i.e. there are $s^1, \ldots, s^{2n-1} \in \mathcal{S}^*$ such that $(s^0, l_1, s^1), (s^1, l_2, s^2), ...,$ (s^{2n-1}, l_n, s^{2n}) are in $\mathcal{R}^*$. We must show that $(s, l, t) \in \mathcal{R}$, where $l = \{l_1, ..., l_n\}$. The proof is based on the following observations

- The set l of automata types that perform the synchronisation is incrementally built in the first round. The moves follow the order o: for all $0 \leq i < n$ we have that $o(l_i) < o(l_{i+1})$. Each state is created from the previous one by performing a move of a synchronisation skeleton from l from a basic state to a "to"-state. After the first round all synchronisation skeletons $SP^I_{l_1}$ to $SP^I_{l_n}$ are in a "to"-state and all other synchronisation skeletons are in a basic state.

- After the first round the second round of moves such that $l_i = l_{n+i}$ is performed. Each state is created from the previous one by performing a move of a synchronisation skeleton from l from a "to"-state to a basic state. After the second round all synchronisation skeletons are in basic states.

- Let st' be the state of SP^I after the first round. Let $\mathit{intState}$ be a set of states of synchronisation skeletons $SP^I_{l_1}$ to $SP^I_{l_n}$ after the first round, that is $\mathit{intState} = \bigcup_{i \in \{1,\ldots,n\}} st' \lceil l_i$. Denote as MOV the set $\{m \mid s \in \mathit{intState} \text{ and } \mathit{toState}(m) = s\}$ of the moves in P^I that are encoded by the moves in the set intState. The set MOV satisfies the definition of a synchronisation in the semantics of P^I and its support is (s, t). Labels of automata whose moves are in MOV are precisely those in l.

From the above it follows, that $(s, l, t) \in \mathcal{R}$. $\square$

A lemma follows that states that fair fullpaths restricted to non-intermediate states (and projected onto their basic part) of a skeleton obtained by the translation are precisely those of the original sync-program.

Now we define the path restriction operator *toBasic* that removes to-states and restricts basic states to their basic part and aggregates the labels between two basic states.

Definition 5.3. Let π' be a path in $\mathit{sem}_{\mathit{skel}}(SP^I)$. We call $\pi = \mathit{toBasic}(\pi')$ is a path obtained from π' as follows: whenever s' is a basic state in π' and t' is the first basic state along π' after s', then there is a corresponding transition in π formed by (s, l, t) where $s = \mathit{basicState}(s')$, $t = \mathit{basicState}(t')$ and l is the set composed of all labels in the portion of the path π' between s' and t'.

We are ready to state and prove the lemma.

Lemma 5.4 (Fullpath Correspondence)**.** *Let P^I be a sync-program and SP^I its translation. The following propositions are equivalent*

- *π' is a fair fullpath in $sem_{skel}(SP^I)$*
- *$\pi = toBasic(\pi')$ is a fair fullpath in $sem(P^I)$.*

Proof. Let π' be a fair fullpath in $sem_{skel}(SP^I)$. Let s' be a basic state in π' and t' the first basic state along π' after s' and there is a sequence of to-states obtained one from the previous one by moves of synchronisation skeletons of types $l_1, ..., l_n$. From the transition correspondence lemma follows that in $sem(P^I)$ there is a transition from the basic part of s' to the state representing the basic part of t' with label $l = l_1, ..., l_n$. The function *toBasic* removes the intermediate series of states and restricts states s' and t' to their basic part and collects types of automata that perform the moves through the series of states between s' and t' into l. This proves the left-to-right implication of the lemma. Since s' was an arbitrary basis state in π', the correspondence holds for the paths π' and $\pi = toBasic(\pi)$. It remains to argue, that π is a fair fullpath in $sem(P^I)$. Since π' is a fair fullpath in $sem_{skel}(SP^I)$, by the definition of fairness for synchronisation skeletons, each synchronisation skeleton participates infinitely often on transitions of the path. This is precisely the definition of fairness for sync-programs.

The converse implication is proved analogously again with the help of the transition correspondence lemma. □

Finally, a formula $fortran(f)$ is true in a $basicState(s)$ iff f is true in s.

Theorem 5.5 (Verification correspondence)**.** *For sync-program P^I and skeleton program SP^I, where $SP^I = transl(P^I)$, for all $s \in sem(P^I)$ for all $f \in ACTL^-$ it holds that*

$$sem(P^I), s \vDash f \text{ iff } sem_{skel}(SP^I), basicState(s) \vDash fortran(f).$$

Proof. By induction on the structure of the formula f.

$f = p$. By the definition of the program translation, $s \vDash p$ iff $s' \vDash p$ for all $p \in AP$. Hence, since $fortran(p) = p$, we have that $s \vDash f$ iff $s' \vDash fortran(f)$.

$f = g \wedge h$. By the semantics of ACTL$^-$, $s \vDash g \wedge h$ iff $s \vDash g$ and $s \vDash h$. By the induction hypothesis, this is equivalent to $s' \vDash fortran(g)$ and $s' \vDash fortran(h)$ where $s' = basicState(s)$ and that is by the semantics of ACTL$^-$ $s' \vDash fortran(g) \wedge fortran(h)$ which is in turn equivalent to $s' \vDash fortran(f)$ by the definition of *fortran*.

$f = g \vee h$. Analogous to the previous case.

$f = A[g\ U_w\ h]$. Let π be an arbitrary fullpath in $sem(P^I)$ starting in s. The Fullpath correspondence lemma guarantees that there is a fullpath π' in SP^I such that $toBasic(\pi') = \pi$. By assumption $\pi \vDash gUh$. By the definition of ACTL$^-$ there are two cases:

1. $sem(P^I), \pi \vDash Gg$. We have to show that any state t' of π' is satisfies $fortran(f)$ that means by the definition of $fortran$, that t' satisfies $\neg toTrue \rightarrow fortran(g)$. If t' is such that there is a $t \in \pi$ and $t' = basicState(t)$ we show that $t' \vDash \neg toTrue \rightarrow fortran(g)$. Since t' is a basic state, $t' \vDash \neg toTrue$ and from the induction hypothesis $t' \vDash fortran(g)$. If, on the other hand, there is no $t \in \pi$ such that $t' = basicState(t)$, then $t' \vDash toTrue$. Hence $t' \vDash \neg toTrue \rightarrow fortran(g)$ by the semantics of ACTL$^-$.

2. $sem(P^I), \pi \vDash g\ U\ h$. By the definition of ACTL^I there is a state $t \in \pi$ such that $t \vDash h$ and for every state u of π between s and t holds $u \vDash g$.

   From the Fullpath correspondence theorem π' starts in s' and leads through states $basicState(v)$ for each $v \in \pi$. Path π' contains $t' = basicState(t)$. We need to show that s' satisfies $fortran(f)$ that means by the definition of $fortran$, that t' satisfies $\neg toTrue \wedge fortran(h)$ and for all $v' \in \pi'$ between s' and t', it holds that $v' \vDash \neg toTrue fortran(g)$.

   Since t' is a basic state, $t' \vDash \neg toTrue$ and from the induction hypothesis $t' \vDash fortran(h)$ which proves the part for t'.

   Now consider any $v' \in \pi'$ between s' and t'. If $v' = basicState(v)$ for some $v \in \pi$ then $v' \vDash \neg toTrue$ and from the induction hypothesis $v' \vDash fortran(g)$ which by definition of ACTL$^-$ proves $v' \vDash \neg toTrue \rightarrow fortran(g)$. If, on the other hand, there is no $v \in \pi$ such that $v' = basicState(v)$, then $v' \vDash toTrue$. Hence $v' \vDash \neg toTrue \rightarrow fortran(g)$.

In both cases we showed $sem_{skel}(SP^I), toBasic(\pi) \vDash g\ U_w\ h$. Since π was an arbitrary fair fullpath starting in s, we conclude $sem(SP^I), basicState(s) \vDash A[g\ U_w\ h]$.

$f = A[g\ U\ h]$. Let π be an arbitrary fair fullpath in $sem(P^I)$ starting in s. By the fullpath correspondence lemma we have that $toBasic(\pi)$ is a fair fullpath in SP^I. From the assumption we know that $P^I, \pi \vDash g\ U\ h$. By the above case we get $SP^I, basicState(s) \vDash fortran(A[g\ U\ h])$. □

Please note, that even though the above theorem is proved for the logic ACTL$^-$ because it suffices for the purpose of the thesis, the Fair fullpath correspondence lemma guarantees a similar verification correspondence result for the entire logic CTL.

5.4 Implementation of Sync-Programs in NuSMV

In this section we demonstrate the process of implementation of a sync-program in NuSMV on an example. We sketch the translation of the sync-program P^I representing the model of the *lac* operon regulation from Section 3.3 to a synchronisation skeleton program. Subsequently, we outline the actual NuSMV code used in the encoding for the tool.

Sync-program $P^I = (S_0^I, P^I_{lac}||P^I_{\beta}||P^I_{allo}||P^I_{op}||P^I_{repr}||P^I_{pos}||P^I_{glu})$ consists of parallel composition of seven sync-automata. Its translation SP^I consists of parallel composition of seven synchronisation skeletons. We will show the translation of one sync-automaton, in particular P^I_{lac} (Figure 5.1) representing lactose.

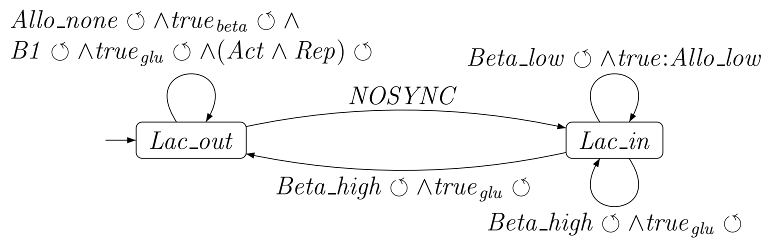

Allo_none ↺ ∧true_beta ↺ ∧
B1 ↺ ∧true_glu ↺ ∧(Act ∧ Rep) ↺
Beta_low ↺ ∧true:Allo_low
NOSYNC
Lac_out
Lac_in
Beta_high ↺ ∧true_glu ↺
Beta_high ↺ ∧true_glu ↺

Figure 5.1: P^I_{lac} – Lactose

Its translation, SP^I_{lac}, is shown on Figure 5.2.

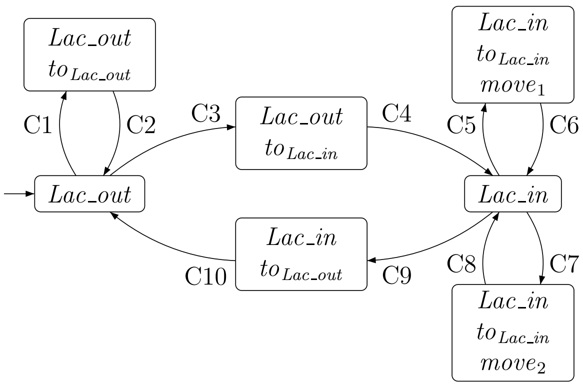


Figure 5.2: SP^I_{lac} – synchronisation skeleton for Lactose

Basic states of the synchronisation skeleton (note that we display only the atomic propositions that are true in the state) are $\{Lac\_in\}$ and $\{Lac\_out\}$. All other states are the intermediate ones, for each move of P^I_{lac} there is one intermediate state. The move enabling conditions are shown in Table 5.1.

The order o considered over I is such that $o(allo) < o(beta) < o(pos) < o(glu) < o(lac) < o(op) < o(repr)$. Consider move of P^I_{lac}
$Lac\_out \xrightarrow{Allo\_none\circlearrowleft\wedge true_{beta}\circlearrowleft\wedge B1\circlearrowleft\wedge true_{glu}\circlearrowleft\wedge(Act\wedge Rep)\circlearrowleft} Lac\_out$. This is translated in two moves of synchronisation skeleton SP^I_{lac}, namely there is one from $\{Lac\_out\}$ to $\{Lac\_out, toLac\_out\}$ with enabling condition C1 and a converse one with condition C2. On the synchronisation represented by these moves the following six components participate: $allo, \beta, glu, lac, op, repr$.

By the definition of *transl*, C1 consists of three conjuncts. The first part expresses that SP^I_{lac} has to wait for the skeletons earlier in the order to arrive in their

$$\begin{array}{lcl}
C1 & = & allo\_none \wedge toAllo\_none \wedge true_{beta} \wedge toTrue\_beta \wedge \\
 & & \wedge true_{glu} \wedge toTrue\_glu \wedge act \wedge rep \wedge b1 \wedge \\
 & & toTrue\_allo \wedge toTrue\_beta \wedge toTrue\_glu \wedge \bigwedge_{i \in I - \{allo,beta,glu\}} \neg to\_i \\
C2 & = & allo\_none \wedge true_{beta} \wedge true_{glu} \wedge act \wedge rep \wedge toAct \wedge toRep \wedge b1 \wedge toB1 \wedge \\
 & & toTrue\_op \wedge toTrue\_repr \wedge \bigwedge_{i \in I - \{op,repr\}} \neg to\_i \\
C3 & = & true \wedge \\
 & & \wedge \bigwedge_{i \in I} \neg to\_i \\
C4 & = & true \wedge \\
 & & \wedge \bigwedge_{i \in I} \neg to\_i \\
C5 & = & beta\_high \wedge toBeta\_high \wedge true_{glu} \wedge toTrue\_glu \wedge \\
 & & toTrue\_beta \wedge toTrue\_glu \wedge \bigwedge_{i \in I - \{beta,glu\}} \neg to\_i \\
C6 & = & beta\_high \wedge glu\_high \wedge \\
 & & \wedge \bigwedge_{i \in I} \neg to\_i \\
C7 & = & beta\_low \wedge toBeta\_low \wedge true_{allo} \wedge toAllo\_low \wedge \\
 & & toTrue\_beta \wedge toTrue\_allo \wedge \bigwedge_{i \in I - \{beta,allo\}} \neg to\_i \\
C8 & = & beta\_low \wedge allo\_low \wedge \\
 & & \wedge \bigwedge_{i \in I} \neg to\_i \\
C9 & = & beta\_high \wedge toBeta\_high \wedge true_{glu} \wedge toTrue\_glu \wedge \\
 & & toTrue\_beta \wedge toTrue\_glu \wedge \bigwedge_{i \in I - \{beta,glu\}} \neg to\_i \\
C10 & = & true \wedge \\
 & & \wedge \bigwedge_{i \in I} \neg to\_i
\end{array}$$

Table 5.1: Enabling conditions for SP^{I}_{lac}

respective intermediate states. Thus C1 contains the following constraints on $allo, \beta$ and glu: $allo\_none \wedge toAllo\_none \wedge true_{beta} \wedge toTrue\_beta \wedge true_{glu} \wedge toTrue\_glu$. A condition follows that the remaining skeletons, participating in this synchronisation but later in the order have to be ready for the transition: $act \wedge rep \wedge b1$. 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: $toTrue\_allo \wedge toTrue\_beta \wedge toTrue\_glu \wedge \bigwedge_{i \in I - \{allo, beta, glu\}} \neg to\_i$.

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 waiting for the execution in a to-state and no other skeleton is in an intermediate state: $allo\_none \wedge true_{beta} \wedge true_{glu}$ and $\wedge act \wedge rep \wedge toAct \wedge toRep \wedge b1 \wedge toB1$ and $toTrue\_op \wedge toTrue\_repr \wedge \bigwedge_{i \in I - \{op, repr\}} \neg to\_i$, respectively.

Now we show how to encode SP^I in the input language of NuSMV. The objective is to give one module for each synchronisation skeleton plus a selector module for simulating the asynchronous behaviour, and the `main` module.

We give one module for each of the synchronisation skeletons. For example to P^I_{lac} corresponds the module

```
MODULE lacMod(...)
```

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 the original variables corresponding to the atomic propositions of the sync-automaton.

```
VAR
    lac_out : boolean;
```

Moreover, there are variables corresponding to the "to"-propositions.

```
    toLac_in : boolean;
```

We call these variables *to-variables*. And lastly, there are variables `true_glu` and `toTrue_glu` whose function is clear.

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. To each state of the sync skeletons corresponds one macro of the following shape.

```
DEFINE
    state1 := lac_out & !toLac_out & !lac_in & !toLac_in
              & !move1 & !toTrue_lac;
```

The extra variable `move1` corresponds to the atomic proposition $move_1$ and ensures the exclusivity for intermediate states in the case of multiple moves.

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, following the order o: `alloMod`, `betaMod`, `posMod`, `gluMod`, `lacMod`, `opMod`, `reprMod`.

Transitions of a module represent the moves of the synchronisation skeleton. We express the enabling condition by means of two macros. The first one specifies the positive requirements on other synchronisation skeletons participating in the synchronisation.

This is expressed by the condition in macro

```
p12 := allo_none & toAllo_none &  true_beta & toTrue_beta &  true_glu
       & toTrue_glu & act & rep & b1;
```

The second part expresses for each module in I, whether it is in a basic or an intermediate state.

```
n12 := toTrue_allo &  toTrue_beta & _toTrue_pos &  toTrue_glu
       & _toTrue_lac & _toTrue_op & _toTrue_repr;
```

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 section.

The transition itself is expressed as a logical characterisation of the transition relation

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

There is one more module `selMod` representing the selector process taking care of the asynchronicity of the operation. The selector module is running in parallel with all other modules and in every step it selects non-deterministically one of the above modules and only that module will be permitted to change the state.

```
VAR
    select : {selAllo,selBeta,selPos,selGlu,selLac,selOp,selRepr};
```

The selector passes the name of the scheduled process to each of the modules. In compliance to this piece of information, each module either performs a transition or stays idle.

The fairness is implemented on the level of the selector. Since once a process is selected it must perform a transition, it is enough to schedule the processes infinitely often. The fairness constraint is included for each of the values of the variable `select`.

```
FAIRNESS
    select = selLac
```

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
        lac : lacMod(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 [36], along with the examples from the following section.

5.5 Modular Verification of *Lac* operon in NuSMV

In this section we show the verification of the properties given in Section 3.3 in NuSMV.

Each property has to be translated by using the function *fortran* from Section 5.2. Then the property is verified on the NuSMV implementation of the translation of the sync-subprogram identified in Section 3.3. Its successful verification on the fragment of the model implies its satisfaction in the whole model of *lac*-operon regulation, as guaranteed by the Property preservation theorem.

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.

	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 5.2: Comparison of whole model verification with modular verification

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-> (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 5.2 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.

Chapter 6

Modular Verification as a Property of Semantics

6.1 Motivation

In the previous chapter we have proved that properties of sync-subprograms are preserved in sync-programs. In this chapter, by analysing the proof of the Preservation Theorem (Theorem 4.7), we characterise the type of formalisms for which it is possible to prove a similar result.

In [52] the compositional reasoning about synchronous systems is investigated. The relationship between the whole and the part is captured by means of a homomorphism. For asynchronous systems the relationship is represented by Attie and Emerson in [7] by the system projection.

Analysing the proof, the Property Preservation Theorem (Theorem 4.7) can be restated as follows by utilising the projection of I-structures (Definition 3.6).

Theorem 6.1. *Let AP_I be s set of atomic propositions and $AP_J \subseteq AP_I$. Let M_I and M_J be I- and J-structures over AP_I and AP_J, respectively. If for each fullpath π in M_I it holds that $\pi' = \pi\lceil J$ is a fullpath in M_J then if $M_J \models f$ then $M_I \models f$ for all $f \in ACTL^-$.*

These I-structures are obtained as semantics of sync-programs. Abstractly, a semantics is a function that associates to each sync-program an I-structure. We want to see what are the conditions on the semantic function (semantics) of sync-programs such that the semantics of a sync-program and its sync-subprogram satisfy the assumption of the theorem. Note that we investigate conditions on the semantics while having a notion of sync-subprogam fixed, i.e. we do not analyse parametrisation with respect to the syntactic projection.

6.2 A Generalised Definition of Semantics

Let us first recall the definition of semantics. It is an LTS with transitions as follows. A transition from state s to state t with label l corresponds to having a selection of automata, precisely those with indices contained in the label l, which perform a move simultaneously and rest of automata stay idle. We call the set of moves performed simultaneously a synchronisation and states s and t its support.

We revisit the definition of a synchronisation and we give a new, generalised definition. The aim is to provide the most general definition that is compatible with the following intuition. A synchronisation should contain moves from distinct automata, because it is impossible that more moves of the same automaton be performed synchronously. The next stipulation is that for each move its conditions on other moves are satisfied, making the synchronisation sound. The last requirement is that only related moves should be considered. This is to guarantee the "atomicity" of the synchronisation, that is just the moves that wish to participate on the synchronisation do so. The above specification is formalised in the following definition.

Definition 6.2. 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. Let G_{MOV} be a graph, where vertices are the moves from MOV and there is a directed edge from m_1 to m_2 iff move m_1 contains a reference to m_2 in its synchronisation condition. Then G_{MOV} is weakly connected.

This definition is only slightly more general than Definition 3.4, in particular in the third point. A directed graph is weakly connected iff when replacing each directed edge by an undirected one, we obtain a connected (undirected) graph. In other words, any pair of vertices is connected through an undirected path. Note that weak connectivity corresponds to our intuition of considering only related moves, in particular MOV cannot be partitioned in two sets, such that both are correct synchronisations.

Furthermore, weak connectivity in a directed graph is satisfied in a complete graph, a condition required in Definition 3.4.

In order to illustrate the difference between the definitions 3.4 and 6.2 note, that in the semantics that uses the latter an indirect synchronisation can be performed. It is possible that two sync-automata participate in a simultaneous transition even without requiring synchronisation from each other, but both being connected through a third party. For instance, if a sync-automaton requires synchronisation with two other sync-automata, these are forced to perform a move synchronously even though they might not request synchronisation from each other directly.

6.3 The MV property

Now we state what it means for a semantic function to have a modular verification (MV) property.

Definition 6.3. Function of semantics $sem : P^I \mapsto M_I$ has the MV property iff for projection $\lceil$, for all sync-programs P^I, for all $J \subseteq I$ holds:
if π is fullpath in $sem(P^I)$ then $\pi' = \pi\lceil J$ is a fullpath in $sem(P^I\lceil J)$.

Note that a semantics with the MV property always satisfies the assumptions of the Theorem 6.1. Furthermore, note that the semantics of sync-programs as defined in Definition 3.4 has the MV property, as proved in the Fullpath Lemma (Lemma 4.4).

Fairness, as defined in Chapter 4, is preserved by the projection (Lemma4.4). From the analysis of the proof of the Fullpath Lemma we infer that if fairness is assumed, the Transition Lemma (Lemma 4.2) is a sufficient condition for the Fullpath lemma. In the following we investigate for which definitions of semantics we are able to prove the Transition lemma. Let us recall this lemma.

Lemma 6.4 (Transition projection)**.** *Let I be an index set and $\mathcal{M}_I = (\mathcal{S}_I, \mathcal{S}_I^0, \mathcal{R}_I)$ 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 $\mathcal{M}_J = (\mathcal{S}_J, \mathcal{S}_J^0, \mathcal{R}_J)$ is the semantics of sync-program $P^J = P^I\lceil J$.*

We show, that for the semantics utilising the definition of a synchronisation (Definition6.2) the Transition projection lemma not necessarily holds.

Example: Let $m_1 = A \xrightarrow{X:\neg X \wedge Z:\neg Z} \neg A$, $m_2 = X \xrightarrow{A:\neg A} \neg X$, $m_3 = Z \xrightarrow{A:\neg A} \neg Z$ where the moves m_1, m_2 and m_3 belong to sync-automaton a_1, a_2 and a_3, respectively. The tuple $MOV_1 = ([AXZ]\{a_1, a_2, a_3\}, [\neg A, \neg X, \neg Z])$ is a synchronisation according the Definition 6.2. But $MOV_1\lceil\{a_2, a_3\} = ([XZ]\{2, 3\}, [\neg X, \neg Z])$ is not a synchronisation because the moves $m_2\lceil\{a_2, a_3\} = X \xrightarrow{NOSYNC} \neg X$ and $m_3\lceil\{a_2, a_3\} = Z \xrightarrow{NOSYNC} \neg Z$ do not form a synchronisation. In particular the third point from the definition of a synchronisation (Definition 6.2) is not satisfied, as the graph $G_{MOV_1\lceil\{a_2,a_3\}} = (\{a_2, a_3\}, \emptyset)$ is not weakly connected.

As we can see from the example, weak directed connectivity of the graph from the definition of synchronisation is not preserved by the projection of the semantics. It follows that for the synchronisation from Definition 6.2 we cannot prove an analogue of the following result that holds for the synchronisation from Definition 3.4.

Lemma 6.5. *If MOV is a synchronisation of moves from P^I, then $MOV\restriction J$ is a synchronisation of moves from sync-subprogram $P^J = P^I\restriction J$.*

Proof. Follows directly from the proof of the Transition lemma (Lemma 4.2) and the definition of sync-subprogram (Definition 4.1). □

It is easy to see that the violation of this desired result comes from the third point of the definition (Definition 6.2), which is the only difference between the two definitions of synchronisation. Hence, it is necessary to impose a stronger condition as a property of the graph in the definition of synchronisation. Let *Prop* denote the desired graph property. Then these are formal requirements on the property *Prop*.

1. *Prop* implies the weak directed connectivity (if $Prop(G_{MOV})$ then G_{MOV} is weakly directed)
2. if $Prop(G_{MOV})$ then $Prop(G_{MOV\restriction J})$ where $J \subseteq I$.

The effect of projection of a synchronisation (Definition 4.1) is that it removes nodes and their induced edges, thus producing an induced subgraph.

The second requirement can be rephrased by using graph theoretical concepts. The property *Prop* is induced hereditary: *Prop* is **induced hereditary** iff if *Prop* holds for G, then it holds for all induced subgraphs H of G [16].

Some induced hereditary properties [49]: completeness, planarity, outerplanarity, bipartity, acyclicity, having max degree, interval graphs, chordal graphs. Not induced hereditary: weak connectivity (as above), connectivity (if a graph it contains a directed path from u to v or a directed path from v to u for every pair of vertices u, v), strong connectivity (if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v).

We prove that the only graph property that satisfies the above two characteristics is completeness.

Lemma 6.6. *If $Prop(G)$ implies $Prop(H)$ for all induced subgraphs H of G and $Prop(G)$ implies $WeakConnected(G)$ then $Prop(G) \leftrightarrow Complete(G)$.*

Proof. Let us suppose by contradiction, that *Prop* is different from *Complete*.

Let G be a graph such that $G \in Prop$. By the assumption G is not complete. That means that there are vertices v and v' such that the edge (v, v') is not in G. Then consider the subgraph G' induced by the vertices v and v'. Since *Prop* is induced hereditary, G' is in *Prop*. Then by the assumption G' is weakly connected. However, this contradicts to the fact that there is no edge between v and v'. Thus the assumption that *Prop* is different from *Complete* is wrong and which completes the proof. □

This means that in order to satisfy the transition lemma, a transition has to consider only moves of automata, that reference each other in the synchronisation condition.

Hence only the semantics of sync-programs that has the MV property is the one defined in Definition 3.4 as proved in the Transition lemma (Lemma 4.2).

Theorem 6.7. *A semantics has the MV property iff a synchronisation considers all and only moves that refer to all of the participants of the synchronisation.*

Proof. For the proof see reasoning above. □

6.4 Modular Verification of Other Formalisms

A semantics of a formalism with the synchronisation of automata/processes (in the following we use the name automata) has the MV property when its concept of synchronisation, that is determining which moves/actions/events/activities (in the following we use the name move) that are performed simultaneously, follows the same principles as the synchronisation in (Definition 3.4), namely:

1. moves are from distinct automata
2. for each move its conditions on other moves are satisfied
3. each move is willing to synchronise with all the participating moves.

There are some well known formalisms that enable synchronisation of multiple processes through shared actions, for example CSP [60] and PEPA [58]. In the synchronisation, action names are used without a reference to the process names. In order to satisfy the third condition, it is necessary that all processes currently prepared to perform an action really do so.

In CSP, even a stronger condition is satisfied, namely if an action is in the alphabet of a process then its participation on the action execution is necessary [50].

In PEPA (by extension Bio-PEPA [25]) we consider a subset of the language, in particular we consider the version where all rates r of all activities$(\alpha, r).S$ are infinite and we denote it PEPA$_\infty$. In a way this is equivalent to a qualitative version of the calculus where all activities are instantaneous. The synchronisation of two processes $C_1 \bowtie_L C_2$ is specifies as a synchronisation on all activities from L, other activities are independent. Again, the satisfaction of the third condition is guaranteed.

As a result, since implicitly all three requirements for the synchronisation in the semantics of all CSP and PEPA$_\infty$ are satisfied, their semantics have the MV property. Thus it is possible to prove the preservation of all ACTL$^-$ formulae for CSP and PEPA$_\infty$.

In CCS [71], processes may interact on two complementary actions a and $\bar{a}$. Only two agents may participate in each interaction. Since processes cannot choose to perform a or $\bar{a}$ without the partner, our syntactical projection does not apply here. Thus we cannot apply our modular verification technique in the case of CCS. It might be, however, possible to create a more sophisticated syntactical projection and prove the modular verification for this projection.

6.5 Equivalence of state-based synch. conditions and action sharing

In this section we argue that the synchronisation principle of sync-automata and the action sharing in the style of CSP are equivalent.

Synchronisation conditions of sync-programs have been introduced with verification in mind. Since properties of behaviours refer to states and not to moves (actions), synchronisation conditions of a move specify preconditions and postconditions of the synchronously performed moves.

As it turns out, in order to be able to apply the modular verification, each move has to indicate all the moves with which it is willing to synchronise. This is a feature that is present implicitly also in the concept of action sharing in the style of Team automata [65] or Communities of interacting automata [67]. Now we show that there is a procedure for obtaining one form of synchronisation from the other while maintaining the semantics (resulting labelled transition system) of the program. For this purpose we consider a fictitious formalism of shared action automata (SAA). Moves between states are labelled by action symbols. A shared action can be executed only if it is performed by all the processes that contain this action in their alphabet.

First take a sync-program P^I for an index set I. We will show how to construct an equivalent shared action automaton. Consider a mapping that to each move of each sync-automaton from P^I assigns a unique identifier, let M be a set of all these identifiers.

We construct a propositional formula $fsync_{P^I}$ that characterises the synchronisation requests of the program P^I. For move with identifier $m \in M$ we define

- $move_m \leftrightarrow m \rightarrow (moving_m \wedge idle_m)$
- $moving_m \leftrightarrow \bigwedge_{c \in sc(m)} \bigvee_{sat(m',c)} m'$
- $idle_m \leftrightarrow \bigwedge_{\neg sat(m',sc(m))} \neg m'$

where $sc(m)$ denotes the synchronisation condition of move m. Furthermore $sat(m, c)$ is a predicate that is true when move m satisfies the synchronisation condition c in both precondition and postcondition or, in extension, at least one of the set of conditions.

The formula $move_m$ intuitively means that whenever move m is performed then also for each automaton referred to in the synchronisation condition of m one of the moves has to be performed too, as specified by the formula $moving_m$. Moreover, all the other moves must not be performed, as described in the formula $idle_m$.

For the sync-automaton P_i, we have a formula $aut_i = NAND_{m \in P_i} move_i$, that specifies that only one of the moves is permitted to be performed at a time. Finally, formula

$$fsync_{P^I} = \wedge_{i \in I} aut_i$$

puts into conjunction specification of all automata. This formula represents the description of the move dependencies.

Now, all models of formula fsync_{P^I} represents all correct synchronisations of the program P^I.

The equivalent shared action automaton has the same states and moves between them as P^I. We describe how to obtain the shared action labels.

Take a model X of formula fsync_{P^I}. The identifiers that are true in X represent moves constituting the corresponding synchronisation. We label these moves with a fresh action name a_X. In this way the SAA performs the synchronisation if and only if all the moves labelled with this symbol participate on the synchronisation.

After labelling the moves of the SAA with the symbols for all the models of formula fsync_{P^I}, some moves might be labelled with more action names. Since the partners of a move in the synchronisation in the sync-programs are fixed for each move, such a situation corresponds to being able to synchronise with different tuples of moves from the same tuple of sync-automata. Since each automaton can perform only one move at a time, it cannot introduce incorrect synchronisation when we unify the symbols. Thus if two symbols appear as a label of the same move, they are substituted with the one that is earlier in the global order of action symbols. Hence we have constructed an SAA that is equivalent to P^I.

In order to construct a sync-automaton from a SAA P', first we construct a propositional formula characterising the synchronisations of P' analogously to the converse case. The sync-automaton equivalent to P' again consists from the same states as P'. Then for a synchronisation corresponding to a model of the above mentioned formula, we include in the sync-automata fresh moves that reference in the synchronisation conditions each other in an absolute way, that is preconditon and postcondition is the full state description. It this way for each synchronisation there is a fresh tuple of moves that make the synchronisation sound. For quite general synchronisations, where a move of the SAA can synchronise with more moves of a partner automaton, this approach can generate quite many moves of the resulting equivalent sync-automaton. Hence, we have constructed a sync-program that is equivalent to the SAA P'.

Note that these algorithms might not be efficient, but their aim is demonstrate the equivalence of the forms of synchronisation.

Chapter 7

Dynamic Sync-Programs

In this chapter we extend the approach by allowing sync-automata (the components of a sync-program) to be created dynamically by other already running sync-automata. Moreover, we allow several instances of the same sync-automata to be executed concurrently, without any bound of the number of concurrent instances of the same sync-automaton. This extension is hence a new formalism that we call *dynamic sync-programs.*

This extension is biologically motivated. Indeed, in biological systems it is often the case that entities involved in the processes of interest (e.g. proteins) are synthesised by other active entities (e.g. the DNA). Moreover, it is very common that a number of copies of the same entities (e.g. proteins) are active at the same time.

A Petri net representation of dynamic sync-programs is given. The existence of such a representation implies that the formalism is not Turing-complete, that verification of the properties of interest is decidable, and that analysis tools for Petri nets can be used to analyse system descriptions given in terms of sync-programs.

We develop a modular verification technique for dynamic sync-programs. In order to be able to express properties of different instances of sync-automata in a dynamic sync-program we define an extension of the logic ACTL$^-$ that we call Dynamic ACTL$^-$. We prove, in the line of Chapter 4, that DACTL$^-$ properties that hold in the semantics of a portion of a dynamic sync-program obtained by a suitable projection operation, hold also in the semantics of the whole dynamic sync-program.

We apply the approach to a biological case study, namely the EGF signalling pathway.

7.1 Syntax

As in the case of sync-programs, dynamic sync-programs are founded on the component-based approach. We assume a finite *index set* I representing the possible indices. Sometimes indices are referred to as types of sync-automata. Whereas in the static case there was a single component of a given type, in the dynamic case there can be also none or more components with the same *index* from I.

Again, with every index i a set AP_i of *atomic propositions* is associated. Sets of atomic propositions are pairwise disjoint for all the types, i.e. if $i \neq j$ then $AP_i \cap AP_j = \emptyset$.

An individual component with an index from I is modelled by a finite state machine called a dynamic sync-automaton.

Definition 7.1. A *dynamic 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*;
- $R_i \subseteq S_i \times SC_i \times CC_i \times S_i$, where $SC_i \subseteq Mset(\bigcup_{j \in I}(\mathcal{P}(AP_j) \times \mathcal{P}(AP_j)))$, and $CC_i \subseteq Mset(\bigcup_{j \in I} \mathcal{P}(AP_j))$ are labelled *moves* between states.

There are essentially two differences between the definitions of a sync-automaton and a dynamic sync-automaton. The first is that in a synchronisation condition of a move of a sync-automaton of type i there can be multiple references to the components of a type j, referring to different instances of sync-automata of such a type. References to other instances of the same type i are also allowed.

As second difference with respect to a sync-automaton, a move of a dynamic sync-automaton contains an extra condition called a *creation condition.* We denote a move from state s_i to state t_i with labels sc_i and cc_i by $s_i \xrightarrow[cc_i]{sc_i} t_i$ where sc_i is the synchronisation condition and cc_i the creation condition. Performing this move amounts to synchronising with sync-automata specified by the sc_i while sync-automata automata described by cc_i are created.

An example of a dynamic sync-automaton can be seen in Figure 7.1. The component's state space is built over two atomic propositions, A and B. The two states of the automaton are $\{A, B\}$ and $\{\neg A, B\}$ with the former chosen to be initial. Moves between states are labelled by synchronisation and creation conditions.

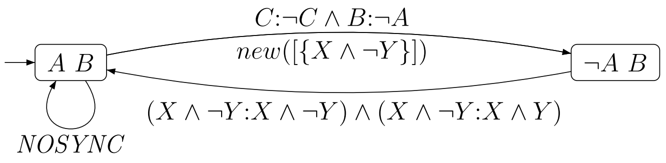

$C{:}\neg C \wedge B{:}\neg A$
$new([\{X \wedge \neg Y\}])$
$A\ B$
$\neg A\ B$
$(X \wedge \neg Y{:}X \wedge \neg Y) \wedge (X \wedge \neg Y{:}X \wedge Y)$
$NOSYNC$

Figure 7.1: Example of a dynamic sync-automaton and move conditions.

The synchronisation condition consists of a set of requirements against other dynamic sync-automata, each formed by a precondition and a postcondition on the move that is expected to be performed synchronously. As mentioned above, more (instances of) automata with the same index can participate on a synchronisation. Moreover, a dynamic sync-automaton can synchronise with other instances with indices equal to its own index. It should be noted that since referencing instances by indices is no longer unambiguous, any of the instances that matches the requirement suffices. Apart from these modifications, the definition does not differ from the static one.

Definition 7.2. A *synchronisation condition* of sync-automaton P_i^I is a label of the form $[A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}]$ where $\{j_1, \ldots, j_n\} \subseteq \mathbb{N}$ and $[A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}]$ is a multiset over $\bigcup_{j \in I}(\mathcal{P}(AP_j) \times \mathcal{P}(AP_j))$ and A_j, B_j are sets of atomic propositions drawn from AP_j or their negations.

Let us denote a synchronisation condition $[A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}]$ as a formula $A_{j_1}{:}B_{j_1} \wedge \ldots \wedge A_{j_n}{:}B_{j_n}$, where A_j, B_j are conjunctions of atomic propositions from A_j and B_j, respectively. Again, an empty synchronisation condition is denoted as $NOSYNC$.

In Figure 7.1 an example of synchronisation conditions of a dynamic sync-automaton can be seen. In particular, the move from $\{A, B\}$ to $\{\neg A, B\}$ requests synchronisation from another instance of its own type. In the converse move the dynamic sync-automaton expresses an intention to synchronise with two automata instances of the same dynamic sync-automaton.

Another important novelty of dynamic sync-automata are creation conditions.

Definition 7.3. A *creation condition* of the dynamic sync-automaton P_i^I is a label of the form $new([C_{j_1}, \ldots, C_{j_n}])$, where $[C_{j_1}, \ldots, C_{j_n}]$ is a multiset over $\bigcup_{j \in I} \mathcal{P}(AP_j)$ and C_j are sets of atomic propositions drawn from AP_j or their negations.

The creation condition specifies dynamic sync-automata that are to be created when performing the move. For each j the set of atomic propositions (or their negations) C_j unambiguously encodes an initial state of a dynamic sync-automaton. In case the multiset of conditions is empty, the creation condition is not displayed. For an example see the move in Figure 7.1 from $\{A, B\}$ to $\{\neg A, B\}$ which specifies a creation of an automaton with initial state $\{X, \neg Y\}$.

By running in parallel dynamic sync-automata of types related by an index set, we obtain a dynamic sync-program. Note that a dynamic sync-program can contain none or multiple dynamic sync-automata of any index. In what follows, we also refer to individual sync-automata in a dynamic sync-program as to sync-automata instances.

Definition 7.4. Let I be an index set. A *dynamic sync-program* is a tuple $P^I = (S_0^I, P_1^I || \ldots || P_n^I)$, where $[P_1^I, \ldots, P_n^I]$ is a multiset of dynamic-sync automata over $\bigcup_{j \in I} P_j^I$. The set $S_0^I = S_1^0 \times \ldots \times S_n^0$ is the multiset of initial states of the sync-program.

The creation conditions on the moves of all sync-automata are well formed, i.e. for every creation condition $new([C_{j_1},\ldots,C_{j_n}])$, each C_j describes a unique state s' of a sync-automaton of type in I and s' is initial.

7.2 Semantics

Let I be an index set, where $I = \{1,\ldots,n\}$. An I-multistate represents a configuration of a program and is defined as a multiset over $S* = \bigcup_{i\in I} S_i$, where S_i is the set of states states of the sync-automaton P_i^I.

As an example consider $I = \{1,2\}$, the sets of states $S_1 = \{a \wedge b, \neg a \wedge b\}$ and $S_2 = \{x \wedge y, \neg x \wedge \neg y\}$. Some examples of I-multistates are: $[a \wedge b, x \wedge y]$, $[a \wedge b, a \wedge b, x \wedge y]$ and $[a \wedge b, a \wedge \neg b, x \wedge y]$.

An I-multistate s *projected* onto a subset J of I is defined as the greatest submultiset of s where all elements are members of $S_J = \bigcup_{i\in J} S_i$. We denote the projection by the projection operator $\lceil J$. For instance, $[a \wedge b, a \wedge b, x \wedge y]\lceil\{1\} = [a \wedge b, a \wedge b]$.

We assume a function *type* : $R* \to I$ where $R* = \bigcup_{i\in I} R_i$ is the set of moves of all dynamic sync-automata in I, that for an move from $R*$ gives its index i. Analogously, function *types* yields a multiset of types of a set of moves.

Now we can proceed to defining the semantics of a sync-program as a labelled transition system over I-multistates, called an I-multistructure. 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 7.5. Let I be an index set. We call a multiset of moves MOV a *dynamic synchronisation* iff
whenever $m \in MOV$ is of type i and has the form $s_i \xrightarrow[cc_j]{A_{j_1}:B_{j_1}\wedge\ldots\wedge A_{j_n}:B_{j_n}} t_i$ there is a bijection *inst* from the multiset $[A_{j_1}{:}B_{j_1},\ldots,A_{j_n}{:}B_{j_n}]$ to the multiset $MOV - \{m\}$ of moves such that for each $(A_j{:}B_j, m_k) \in inst$ if m_k has the form $s_k \xrightarrow[cc_k]{sc_k} t_k$ then for all $p \in A_j$: $s_k(p) = tt$ and for all $p \in B_j$: $t_k(p) = tt$.

A dynamic synchronisation is composed of a multiset of moves. For each move the bijective multiset mapping *inst* between the multiset of synchronisation conditions and the multiset of all other moves serves as an assignment of the conditions to moves within the dynamic synchronisation that satisfy them.

We remark that the dynamic synchronisation is based on analogous principles as its counterpart synchronisation in Chapter 3. In particular, instead of an explicit bijective mapping there is an implicit bijection in the synchronisation since there is only one sync-automaton of a given type. Furthermore, the completeness of the synchronisation condition graph explicitly required in a synchronisation is satisfied in a dynamic synchronisation implicitly. Note the elimination of the constraint of moves from distinct automata, actually being the basic feature of dynamic-sync programs.

As an example consider multiset $MOV = [m_1, m_1, m_2]$ where $m_1 = A \xrightarrow[cc_1]{sc_1 \wedge sc_2} B$, $m_1 = A \xrightarrow[cc_1]{sc_1 \wedge sc_2} B$ and $m_2 = X \xrightarrow[cc_1]{sc_1 \wedge sc_1} Y$, where $sc_1 = A{:}B$ and $sc_2 = X{:}Y$. MOV satisfies the definition of a dynamic synchronisation. In particular

- for m_1, the function $inst = [(sc_1, m_1), (sc_2, m_1)]$ and
- for m_2, the function $inst = [(sc_1, m_1), (sc_1, m_1)]$.

A pair of I-multistates such that the second multistate can be reached from the first one by carrying out the synchronisation MOV while creating automata specified by creation conditions of the moves in MOV is called a support of the synchronisation MOV.

Definition 7.6. Let I be an index set. Consider a dynamic synchronisation MOV. We call a couple of I-multistates (s, t) the *support of the dynamic synchronisation* MOV iff s and t are I-multistates and

1. if s_{MOV} is the multiset of all starting states of all moves in MOV, then $s_{MOV} \subseteq s$
2. if t_{MOV} is the multiset of all ending states of all moves in MOV, then $t_{MOV} \subseteq t$
3. $(t - t_{MOV}) - (s - s_{MOV})$ is equal to the multiset of states that correspond to the creation conditions of moves in MOV.

For the dynamic synchronisation MOV from the example above, the couple (s, t), where $s = [A, A, X, X, C]$ and $t = [B, B, Y, X, C, D]$, is a support of the dynamic synchronisation MOV. Intuitively, in the synchronisation two As are changed to B, one X is transformed to Y, D is created and the rest of the multistate is left intact.

Thus the semantics is a labelled transition system over I-multistates where the transitions between two multistates s and t are obtained from the dynamic synchronisations with support (s, t).

Definition 7.7. Let $I = \{1, \dots, n\}$ be an index set. The *semantics* of a dynamic sync-program $P^I = (S^0_I, P^I_1 || ... || P^I_n)$ is given by the I-*multistructure* $\mathcal{M}_I = (\mathcal{S}_I, \mathcal{S}^0_I, \mathcal{R}_I)$, where $\mathcal{S}_I$ is a set of I-multistates, $\mathcal{S}^0_I \subseteq \mathcal{S}_I$ is the set of initial multistates and $\mathcal{R}_I \subseteq \mathcal{S}_I \times Mset(I) \times \mathcal{S}_I$ is a transition relation giving the transitions of P^I. A transition (s, l, t) is in $\mathcal{R}_I$ iff there is a nonempty multiset MOV of moves such that $l = types(MOV)$ and MOV is a dynamic synchronisation with support (s, t).

A transition of the form (s, l, t) corresponds to the situation where dynamic sync-automata with indices in l with multiplicity corresponding to the multiplicity of indices from I in l perform moves, while potentially creating some dynamic sync-automata and the rest of the automata stays idle.

A concept that will allow us to reason about properties of programs is that of path. A *path* in an I-multistructure $\mathcal{M}_I$ is a sequence of I-multistates and transition labels $\pi = (s^1, l^1, s^2, l^2, \ldots)$ such that for all m, $(s^m, l^m, s^{m+1}) \in \mathcal{M}_I$. A *fullpath* is a maximal path. Let π^m denote the suffix of π starting in m-th I-multistate.

7.3 Dynamic Sync-Programs as Petri nets

In this section we define an alternative representation of dynamic sync-programs in terms of Petri nets.

Intuitively, in the Petri net representation of a sync-program, a place represents a state of an index and a transition embodies synchronisation of sync-automata and creation of new ones. Moreover, the number of tokens in a place represents the number of automata currently in that state, and the initial marking of the net is constructed according to the initial state of the sync-program. Note that the Petri net graph is created from the set of all indices and is always finite.

Petri nets are very intuitive and allow the modeller to have a comprehensive view of the system. Even though Petri nets can be constructed by following a modular methodology, they are not directly suitable for modular verification. We shall exploit the Petri net representation of sync-programs and the formal semantics of Petri nets to define the formal semantics of sync-programs. Moreover, the existence of a representation of sync-programs as Petri nets is very important since it implies that the formalism is not Turing-complete, that verification of the properties of interest is decidable, and that analysis tools for Petri nets can be used to analyse system descriptions given in terms of sync-programs.

The Petri net representation of dynamic sync-programs is as follows.

Definition 7.8. Given a set of dynamic sync-automata $P_I = \{P_i^I \mid i \in I\}$, we construct a *Petri net graph* $G_I = (S_I, T_I, W_I)$ as follows. The set of places is $S_I = \bigcup_{i \in I} S_i$.

The set of transitions $T_I = \{t_{MOV} \mid MOV \text{ is a dynamic synchronisation from } P_I\}$

The set of arcs $W_I \subseteq (S_I \times T_I) \cup (T_I \times S_I)$ is as follows: if MOV is a dynamic synchronisation from P_I, denote as $st(MOV)$ the multiset of starting states of all moves in MOV and $en(MOV)$ the multiset of ending states of all moves in MOV.

There is an arc from a state s to t_{MOV} iff s belongs to the multiset $st(MOV)$ and the multiplicity of the arc is equal to the multiplicity of s in $st(MOV)$.

Morevoer, there is an arc from t_{MOV} to a state s iff s belongs to the multiset $en(MOV)$ and the multiplicity of the arc is equal to the multiplicity of s in $en(MOV)$.

We argue that the structure obtained as a representation of a dynamic sync-program is a Petri net. The set of places is finite because it is a finite union of finite sets. The set of transitions is finite because there is a transition for each dynamic synchronisation of moves in the dynamic sync-program. Each dynamic synchronisation is composed of a finite multiset of moves, since there is a bijection between the synchronisation conditions and the moves and the conditions are always finite. Finally, it is easy to see that the set of places and the set of transitions are disjoint.

Each transition of the Petri net represents one dynamic synchronisation. There are arcs from the places representing the starting states of the moves constituting the synchronisation to the transition representing the synchronisation, with the respective multiplicity equal to the multiplicity of the move in the synchronisation. Similarly, there are arcs from the places representing the ending states of the moves in the synchronisation with the respective multiplicity. Moreover, there are arcs from the transition to places representing the initial states of automata that were created by performing the synchronisation corresponding to the transition, again respecting the multiplicity.

A Petri net $PN_I = (G_I, M_0)$ consists of a Petri net graph G_I defined above and of an initial marking M_0.

Definition 7.9. Given a dynamic sync-program $P^I = (S_0^I, P_1^I || \dots || P_n^I)$, the initial marking M_0 of G_I as follows: a place representing an initial state of a dynamic sync-automaton from P_i^I contains as many tokens in the initial marking M_0 as there are instances of this initial state in the program P^I.

Now we state the equivalence result that relates dynamic sync-programs to their Petri net representation.

Theorem 7.10. *Let P^I be dynamic sync-program and N_I its Petri net representation. Then these two representations are equivalent, namely the dynamic sync-program semantics of P^I and the Petri net semantics of N^I are the same labelled transition systems.*

We do not give the proof of this equivalence result here. It is fairly straightforward, but quite laborious.

7.4 Dynamic ACTL$^-$

We define a logic for specification of properties of dynamic sync-programs. We adapt the logic ACTL$^-$ for our purpose.

Our dynamic version of universal computation tree logic differs from ACTL$^-$ in that it needs to be able to reason about instances (multisets of automata). For this reason we give an enriched propositional formula that associates with every atomic proposition an identifier and an index.

For example the propositional formula $\exists k{:}i.a(k)$ asserts the existence of a distinct instance of a dynamic sync-automaton with index i that is in a state that satisfies the atomic proposition a. The identifier k serves only for distinction from atomic propositions of the other instances of automata with the same index. For example, the property $\exists k{:}i.(\exists k'{:}i.(a(k) \wedge a(k')))$ refers to two instances of sync-automata of the same type i – the formula says that there are two distinct dynamic sync-automata and both of them need to be in a state that satisfies a.

As we could see, identifiers are bound by existential and universal operators which allow reasoning with instances. We are not able to reference a particular automaton amongst the automata of a given type, we can rather specify existence and properties of different instances. Therefore, the property $\exists k{:}i.\exists k'{:}i.(a(k) \wedge \neg a(k'))$ is a consistent property that states that there are two distinct copies of dynamic sync-automata with index i, one of which is in a state satisfying a and the other in a state satisfying $\neg a$.

Now we give the syntax of DACTL$^-$ formally. Note that the path operators mimic the path operators of ACTL$^-$.

Definition 7.11 (Syntax of DACTL$^-$)**.** Let AP be a set of atomic propositions, I be an index set and suppose that there is an infinite set Id of identifiers. The logic DACTL$^-$ consists of formulae that are defined inductively as follows.

Propositional formulae

- $\mathit{true}, \mathit{false}$ are propositional formulae
- $a(k)$ is a propositional formula, where $a \in AP$ and $k \in Id$
- $\mathit{true}_i(k)$ and $\mathit{false}_i(k)$ are propositional formulae, where $k \in Id$ and $i \in I$
- if f and g are propositional formulae, then so are $\neg f$, $f \wedge g$, $f \vee g$, $f \rightarrow g$ and $f \leftrightarrow g$
- if f is a propositional formula, then so are $\forall k{:}i.f$ and $\exists k{:}i.f$, where $k \in Id$ and $i \in I$

Formulae

- each propositional formula is a formula
- if h_1 and h_2 are formulae, then so are $h_1 \wedge h_2$ and $h_1 \vee h_2$
- if h_1 and h_2 are formulae, then so are $A[h_1\ U\ h_2]$ and $A[h_1\ U_w\ h_2]$.

On the level of state, formulae can be created by using the boolean operators out of atomic propositions with associated identifiers. These identifiers can be bound by quantifiers $\forall$ and $\exists$. Intuitively, $\forall k{:}i.f$ means that for all instances k of index

i state formula f holds. Dually, $\exists k{:}i.f$ says that there is an instance k of index i where state formula f holds.

The scope of quantifier $\forall$ in formula $\forall k{:}i.f$ is formula f. Similarly the scope of quantifier $\exists$ in formula $\exists k{:}i.f$ is formula f. We say that quantifiers $\forall k{:}i$ and $\exists k{:}i$ bind an occurrence of index k, if this occurrence is in the scope of the quantifier. We consider formula f *well formed*, if its each index occurrence is bound by at most one quantifier, formulae $true_i(k)$ and $false_i(k)$ are bounded only by quantifiers $\forall k{:}i$ or $\forall k{:}i$ with corresponding types and $a(k)$ is bound only by quantifiers $\forall k{:}i$ or $\forall k{:}i$ where $a \in AP_i$. We say that formula f is *closed*, if its each index occurrence is bound by at least one quantifier. In what follows, we will only consider closed well formed formulae.

On the level of paths, this logic is able to express only formulae that hold for all paths, by using the universally quantified until operator. Abbreviations in DACTL$^-$: $AFf \equiv A[true\ U\ f]$ and $AGf \equiv A[f\ U_w\ false]$. Properties expressible by DACTL$^-$ formulae are similar to ACTL$^-$ properties and thus are able to represent a significant class of biologically relevant properties.

We define the logic DACTL^-_J to be DACTL$^-$ where the atomic propositions are drawn from $AP_J = \bigcup_{j\in|J|} AP_j$.

DACTL$^-$ formulae are evaluated on I-multistructures. The definition of the semantics of DACTL$^-$ follows. Note that only fair fullpaths are considered. The concept of fairness is the one of Chapter 4.

Definition 7.12 (Semantics of DACTL$^-$)**.** We define $\mathcal{M}_I, s \models f$ (resp. $\mathcal{M}_I, \pi \models f$) meaning that f is true in multistructure $\mathcal{M}_I$ at state s (resp fair fullpath π).

For any formula f we define $\mathcal{M}_I, s^1 \models f$ as $\mathcal{M}_I, \emptyset, s^1 \models f$.

For a multiset mapping $\sigma : Id \to Mset(I)$ we define $\mathcal{M}_I, \sigma, s \models f$ inductively: Let f and g be propositional formulae.

- $\mathcal{M}_I, \sigma, s^1 \models true$.
- $\mathcal{M}_I, \sigma, s^1 \not\models false$.
- $\mathcal{M}_I, \sigma, s^1 \models a(k)$ iff $s'(a)$ where $\sigma(k) = s'$.
- $\mathcal{M}_I, \sigma, s^1 \models true_i(k)$ iff $\sigma(k) \neq \emptyset$.
- $\mathcal{M}_I, \sigma, s^1 \models false_i(k)$ iff $\sigma(k) = \emptyset$.
- $\mathcal{M}_I, \sigma, s^1 \models \neg f$ iff not $\mathcal{M}_I, \sigma, s^1 \models f$.
- $\mathcal{M}_I, \sigma, s^1 \models f \wedge g$ iff $\mathcal{M}_I, \sigma, s^1 \models f$ and $\mathcal{M}_I, \sigma, s^1 \models g$.
  The cases for $f \vee g$, $f \to g$ and $f \leftrightarrow g$ are analogous.
- $\mathcal{M}_I, \sigma, s^1 \models \forall k{:}i.f$ iff for all elements s' from $\mathcal{P}(AP_i)$ in multiset $s^1 - Im(\sigma)$ holds $\mathcal{M}_I, \sigma \uplus (k, s'), s^1 \models f$.

- $\mathcal{M}_I, \sigma, s^1 \models \exists k{:}i.f$ iff there exists an element s' from $\mathcal{P}(AP_i)$ in multiset $s^1 - Im(\sigma)$ such that $\mathcal{M}_I, \sigma \uplus (k, s'), s^1 \models f$.

Let h_1 and h_2 be formulae.

- $\mathcal{M}_I, \sigma, s^1 \models h_1 \wedge h_2$ iff $\mathcal{M}_I, \sigma, s^1 \models h_1$ and $\mathcal{M}_I, \sigma, s^1 \models h_2$.
  The case for $h_1 \vee h_2$ is analogous.
- $\mathcal{M}_I, \sigma, s^1 \models A[h_1\ U\ h_2]$ iff for every fair fullpath $\pi = (s^1, l^1, \ldots)$ in $\mathcal{M}_I$: there exists $m \in \mathbb{N}$ such that $\mathcal{M}_I, \sigma, \pi^m \models h_2$ and for all $m' < m$: $\mathcal{M}_I, \sigma, \pi^{m'} \models h_1$.
- $\mathcal{M}_I, \sigma, s^1 \models A[h_1\ U_w\ h_2]$ iff for every fair fullpath $\pi = (s^1, l^1, \ldots)$ in $\mathcal{M}_I$: for all $m \in \mathbb{N}$, if for all $m' < m$: $\mathcal{M}_I, \sigma, \pi^{m'} \not\models h_2$ then $\mathcal{M}_I, \sigma, \pi^m \models h_1$.

The main difference of DACTL$^-$ with respect to CTL (and its subset ACTL$^-$) lies in propositional formulae, namely it is the possibility of dealing with instances. In fact, in the semantics of propositional formulae there is a multiset mapping σ associating identifiers with particular instances of dynamic sync-automata of type chosen by the quantifier. The satisfaction of an atomic proposition is resolved in the instance associated with the proposition by its binding quantifier. Since we consider only closed well-formed formulae, exactly one binding quantifier exists for each atomic proposition. When evaluating a closed well-formed formula, the function σ is empty.

We illustrate the definition of the semantics on an example. Consider index set I which includes 0 and $AP_0 = \{a, b\}$. Consider an I-multistate $s^1 = [\{a, b\}, \{a, b\}, \{\neg a, b\}]$ which can be represented also as $[a \wedge b, a \wedge b, \neg a \wedge b]$. Now we show that s^1 contains two distinct states of dynamic sync-automata P_0^I both of which satisfy a:

- $\mathcal{M}_I, s^1 \models \exists k{:}0.\exists k'{:}0.(a(k) \wedge a(k'))$ iff
- $\mathcal{M}_I, \emptyset, s^1 \models \exists k{:}0.\exists k'{:}0.(a(k) \wedge a(k'))$ iff
- there exists an element s' from $\mathcal{P}(AP_0)$ in s^1 such that
  $\mathcal{M}_I, [(k, s')], s^1 \models \exists k'{:}0.(a(k) \wedge a(k'))$ iff
- there exists an element s' from $\mathcal{P}(AP_0)$ in s^1 such that there exists an element s'' from $\mathcal{P}(AP_0)$ in $s^1 - [s']$ such that $\mathcal{M}_I, [(k, s'), (k', s'')], s^1 \models a(k) \wedge a(k')$ iff
- there exists an element s' from $\mathcal{P}(AP_0)$ in s^1 such that there exists an element s'' from $\mathcal{P}(AP_0)$ in $s^1 - [s']$ such that $\mathcal{M}_I, [(k, s'), (k', s'')], s^1 \models a(k)$ and $\mathcal{M}_I, [(k, s'), (k', s'')], s^1 \models a(k')$
- there exists an element s' from $\mathcal{P}(AP_0)$ in s^1 such that there exists an element s'' from $\mathcal{P}(AP_0)$ in $s^1 - [s']$ such that $s'(a)$ and $s''(a)$
- since there are $s' = a \wedge b$ and $s'' = a \wedge b$ and they satisfy the requirements, the above satisfaction holds and therefore the original satisfaction holds.

For further examples of the satisfaction of propositional formulae, consider $AP_t = \{a, b\}$ and let s be the multistate $[a \wedge b, a \wedge \neg b]$ of an I-multistructure $\mathcal{M}_I$. Now we have that $\mathcal{M}_I, s \vDash \exists k{:}t.\exists k'{:}t.(b(k) \wedge \neg b(k'))$ holds, since there exists a function σ that maps k and k' to different elements of s and such that $s_{\sigma(k)}$ satisfies b and $s_{\sigma(k')}$ satisfies $\neg b$. Note that the requirement that k and k' are mapped to different elements of s comes from the fact that the formula $b(k) \wedge \neg b(k')$ is in the scope of both k and k'. In fact, we have that also $\mathcal{M}_I, s \vDash \exists k{:}t.b(k) \wedge \exists k'{:}t.b(k')$ holds, since this time k and k' can be mapped to the same element of s (the one satisfying b). For the same reason we have that $\mathcal{M}_I, s \vDash \exists k{:}t.\exists k'{:}t.(b(k) \wedge b(k'))$ does not hold, since in s there are not two different elements satisfying b.

This example shows that usual properties that allow to change the position of existential and universal quantifiers inside a formula do not hold in this case, since the scope of quantifiers is important to determine whether two different portions of a formula can be satisfied by the same instance of a sync-automaton or must be satisfied by two different instances.

Let us consider a few other interesting examples of propositional formulae. The first is $\exists k{:}t.true_t(k)$, that holds if and only if there exists at least one dynamic sync-automaton of type t. This formula can be used inside a temporal formula to test whether a sync-automaton of type t will be eventually created (e.g. in $A[true\ U\ \exists k{:}t.true_t(k)]$). Similarly, we have that $\forall k{:}t.false_t(k)$ corresponds to the negation of the previous state formula and holds if and only if there exist no sync-automata of type t. Finally, we can construct a state formula stating that there exists exactly one sync-automaton of given type t as follows: $\exists k{:}t.\forall k'{:}t.(true_t(k) \wedge false_t(k'))$.

We remark that even though the presented logic is of universal nature because of its role in the modular verification which is the main topic of the thesis, it would be possible to give a dynamic logic similar to the entire logic CTL. Indeed, the difference between DACTL$^-$ and ACTL$^-$ lies in the propositional formulae, keeping the temporal and path operators identical to the CTL case.

We have extended the logic ACTL$^-$ for this dynamic scenario allowing to refer to automata instances. The logic could be further improved, by allowing the instances to be tracked over time. Technically this would be done by moving the existential and universal quantifiers from the level of state formulae to that of formulae. Note, however, that this is not possible without further information included in the semantics of sync-programs.

7.5 Modular verification

We develop a modular verification technique for dynamic sync-programs. We prove, in the line of Chapter 4, that satisfaction of a DACTL$^-$ property on a projected model entails its satisfaction in the original model.

In order to prove the property preservation, we need the fullpath preservation: every fair fullpath representing a computation of the entire system when projected onto J is present also as a computation of the projected system P^J. This is ensured by the Transition projection and Path projection lemmas and a notion of fairness. Finally we show the preservation of the logic DACTL$^-$ by resorting to the previously proved lemmas.

Note that there are two main differences of dynamic sync-programs with respect to sync-programs, namely the reasoning with instances and the creation.

The former is dealt with by exploiting the multiset representation and by modifying all structures and operations to manipulate this representation.

The latter, the dynamic creation, influences the modular verification. In particular, the subprogram that is considered for the modular verification of the properties must necessarily consider not only a portion J of interest, but also the part of I containing sync-automata that are able to create dynamic-sync automata within J. The reason is, that the dynamic sync-automata that are able to create instances of J, even by transitivity, might have an effect on the properties of interest concerning the dynamic sync-program P^J. Therefore, the modular verification has to take into account all the mentioned dynamic-sync automata.

7.5.1 Subprograms and projections

In this section we formally define the syntactical and semantic projections that will be useful for the purpose of modular verification.

A *dynamic sync-subprogram* represents the behaviour of a portion of a sync-program in isolation. A dynamic sync-subprogram is obtained from a dynamic sync-program P^I by projection operator $\lceil J$ which is much the same as $\lceil J$ in Chapter 4.

Definition 7.13. Let I be an index set and $J \subset I$ and $J = \{j_1, \ldots, j_k\}$. Let $P^I = (S_0^I, P_1^I || \ldots || P_n^I)$ where $[P_1^I, \ldots, P_n^I]$ is a multiset of dynamic-sync automata over $\bigcup_{j \in I} P_j^I$ with $P_i^I = (S_i, S_i^0, R_i)$ and for each $i \in I$. Then $P^I \lceil J = (S_0^J, P_{j_1}^J || \ldots || P_{j_k}^J)$ where $[P_{j_1}^I, \ldots, P_{j_k}^I]$ is a maximal submultiset of dynamic-sync automata over $\bigcup_{j \in J} P_j^I$ with $P_j^J = (S_j, S_j^0, R'_j)$ for each $j \in J$ where S_j and S_j^0 are as in P_j^I and

$$\begin{array}{rcl} R'_j & = & \{ \;\; s_j \xrightarrow[new([C_{j_1},\ldots,C_{j_n}] \cap Mset(\bigcup_{j \in J} \mathcal{P}(AP_j)))]{[A_{k_1}:B_{k_1},\ldots,A_{k_m}:B_{k_m}] \cap Mset(\bigcup_{j \in J}(\mathcal{P}(AP_j) \times \mathcal{P}(AP_j)))} t_j \\ & & \mid \;\; s_j \xrightarrow[new([C_{j_1},\ldots,C_{j_n}])]{[A_{k_1}:B_{k_1},\ldots,A_{k_m}:B_{k_m}]} t_j \in R_j \}. \end{array}$$

Initial states are $S_0^J = S_{j_1}^0 \times \ldots \times S_{j_n}^0$.

The projection contains only dynamic sync-automata from J and their synchronisation and creation conditions references to dynamic sync-automata outside J are removed. Again a dynamic sync-subprogram $P^I \upharpoonright J$ is still a dynamic sync-program with index set J, hence it can be also denoted by P^J.

In what follows we will always use the syntactical projection in connection with a notion of *creation set* of J. It can be obtained through syntactic analysis of the program P^I as follows.

Let P^I be a dynamic sync-program and J be a subset of I. Consider an oriented graph G with vertices from I. There is an oriented edge from i to j if there is a move in in P_i^I whose creation condition contains an initial state of automaton of type j. The creation set of J is defined as

$$J \cup \{i \in G \mid \text{there is an oriented path from } i \text{ to some } j \in J\}.$$

Intuitively, the projection of a dynamic sync-program onto creation graph J' of J contains all automata that are from J or that create (even by transitivity) an automaton from J.

To define the *semantic projection*, let us denote with $s\lceil J$ the projection of an I-multistate s onto a $J \subseteq I$. It is the greatest submultiset of s consisting only of states of dynamic sync-automata from J. A transition (s, l, t) in an I-multistructure can be projected onto $J \subseteq I$ as follows: $(s, l, t)\lceil J = (s\lceil J, l \cap J, t\lceil J)$. Projection $\mathcal{M}_I\lceil J$ of an I-multistructure $\mathcal{M}_I$ onto J can now be defined as the transition relation consisting of the projections onto J of the transitions in $\mathcal{M}_I$. Note that it is possible that the transition label is an empty set, and the corresponding projected multistates are both equal (synchronisation outside J was removed) or different (because of creation by an automaton outside J). However this does not cause problems, as for the purpose of verification the transitions of the former type are removed.

Path projection $\pi\lceil J$ is obtained by projecting every transition of the path π onto J.

7.5.2 Path Preservation

The projection of a dynamic sync-program onto creation set J' of J contains all automata that are from J or that create (even by transitivity) an automaton from J.

To be able to perform the verification on the semantics of a sync-subprogram, 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.

Since a computation of a sync-program has been defined as a fair fullpath in its semantics, we need to show that every fullpath in the semantics of P^I projected onto J is a fullpath in the semantics of $P^J = P^I \upharpoonright J$. However, we need the syntactic projection to be done not onto J, but onto its creation set J'. This is necessary since sync-automata from $J' - J$ may cause the creation of some sync-automata from J that may be necessary to satisfy the property of interest.

The Transition projection lemma states that each individual transition in the semantics of P^I that involves sync-automata from J (either because they perform some moves or because they are created) has a corresponding transition in the semantics of $P^I \lceil J'$.

Lemma 7.14 (Transition projection). *Let I be an index set and $\mathcal{M}_I = (\mathcal{S}_I, \mathcal{S}_I^0, \mathcal{R}_I)$ the semantics of dynamic sync-program P^I. For all I-multistates s, t in $\mathcal{S}_I$ and all $l \in Mset(I)$,*

$$(s, l, t) \in \mathcal{R}_I \text{ iff } (s, l, t)\lceil J \in \mathcal{R}_{J'}\lceil J$$

for all $J \subseteq I$ such that if $l \cap J \neq \emptyset$ then $s\lceil J \neq t\lceil J$, where J' is the creation set of J and $\mathcal{M}_{J'} = (\mathcal{S}_{J'}, \mathcal{S}_{J'}^0, \mathcal{R}_{J'})$ is the semantics of sync-program $P^{J'} = P^I \lceil J'$.

Proof. Direction right to left. Suppose that for any $J \subseteq I$ such that if $l \cap J = \emptyset$ then $s\lceil J \neq t\lceil J$ holds, $(s, l, t)\lceil J \in \mathcal{M}_{J'}\lceil J$. By taking $J = I$ we get $(s, l, t) \in \mathcal{M}_I$.

Direction left to right. Suppose that $(s, l, t) \in \mathcal{R}_I$, we will show $(s, l, t)\lceil J \in \mathcal{R}_{J'}\lceil J$ for any $J \subseteq I$ such that if $l \cap J \neq \emptyset$ then $s\lceil J \neq t\lceil J$.

From the definition of the semantics of the dynamic sync-program P^I we have that there is a dynamic synchronisation MOV with support (s, t) s.t. $types(MOV) = l$. We need to show that in $\mathcal{M}_{J'}$, where J' is the creation set of 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 multiset

$$\begin{aligned} MOV' \;=\; \{ \; & s_j \xrightarrow[new([C_{j_1},\ldots,C_{j_n}] \cap Mset(\bigcup_{j\in J'} \mathcal{P}(AP_j)))]{[A_{k_1}:B_{k_1},\ldots,A_{k_m}:B_{k_m}] \cap Mset(\bigcup_{j\in J'}(\mathcal{P}(AP_j)\times\mathcal{P}(AP_j)))} t_j \\ \mid \; & s_j \xrightarrow[new([C_{j_1},\ldots,C_{j_n}])]{[A_{k_1}:B_{k_1},\ldots,A_{k_m}:B_{k_m}]} t_j \in MOV \text{ and } j \in J' \}. \end{aligned}$$

Since $l \cap J \neq \emptyset$, and J' is the creation set of J, MOV' is not empty. Now we prove that MOV' satisfies the definition of a dynamic synchronisation.

Let m' be any move in MOV' and it has the form $s_i \xrightarrow[cc_j]{[A_{j_1}:B_{j_1}\wedge\ldots\wedge A_{j_r}:B_{j_r}]} t_i$. By the definition of MOV' there is a move m such that $s_i \xrightarrow[cc_j]{[A_{j_1}:B_{j_1}\wedge\ldots\wedge A_{j_n}:B_{j_n}]} t_i$ in MOV. By the definition of dynamic synchronisation in $\mathcal{M}^I$ there is a bijection $inst$ from the multiset $[A_{j_1}{:}B_{j_1}, \ldots, A_{j_n}{:}B_{j_n}]$ to the multiset $MOV - \{m\}$ of moves such that for each $(A_j{:}B_j, m_k) \in inst$ if m_k has the form $s_k \xrightarrow[cc_k]{sc_k} t_k$ then for all $p \in A_j$: $s_k(p) = tt$ and for all $p \in B_j$: $t_k(p) = tt$.

It is easy to see that $inst \cap (Mset(\bigcup_{j\in J'}(\mathcal{P}(AP_j) \times \mathcal{P}(AP_j))) \times R_{J'})$ is a bijection from the multiset $[A_{j_1}{:}B_{j_1}, \ldots, A_{j_r}{:}B_{j_r}]$ to the multiset $MOV' - \{m'\}$ of moves such that for each $(A_j{:}B_j, m_k) \in inst$ if m_k has the form $s_k \xrightarrow[cc_k]{sc_k} t_k$ then for all $p \in A_j$: $s_k(p) = tt$ and for all $p \in B_j$: $t_k(p) = tt$.

Hence MOV' satisfies the definition of a dynamic synchronisation in $P^{J'}$. It remains to show that, $(s\lceil J', t\lceil J')$ is a support for MOV'.

Let s_{MOV} be the multiset of all starting states of all moves in MOV and t_{MOV} be the multiset of all ending states of all moves in MOV. From the assumption we have that $(t - t_{MOV}) - (s - s_{MOV})$ is equal to the multiset of states that correspond to the creation conditions of moves in MOV. It is easy to see that if $s_{MOV'}$ is the multiset of all starting states of all moves in MOV' and t'_{MOV} is the multiset of all ending states of all moves in MOV' then $(t\lceil J' - t_{MOV'}) - (s\lceil J' - s_{MOV'})$ is equal to the multiset of states that correspond to the creation conditions of moves in MOV'.

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 $\mathcal{R}_{J'}\lceil J$. □

By exploiting the Transition projection lemma we can prove a lemma dealing with paths, and called Path projection lemma.

Lemma 7.15 (Path projection)**.** *Let I be an index set and $\mathcal{M}_I$ be the semantics of dynamic 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 $\mathcal{M}_{J'}\lceil J$, where J' is the creation set of J and $\mathcal{M}_{J'}$ is the semantics of sync-program $P^I\lceil J'$.*

Proof. Let $\pi = (Bl^1, l^1, Bl^2, l^2, \ldots)$ be a path in $\mathcal{M}_I$ and Bl^m J'-blocks for all m where J' is the creation set of J. By s^m and t^m denote first and last state of Bl^m, respectively. By definition of I-multistructure 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 $\mathcal{M}_{J'}\lceil J$ for all m. Now every J'-block projected on J collapses to one state, i.e. $s^m\lceil J = t^m\lceil J$ for all m. Therefore $(s^m\lceil J, l^m \cap |J|, s^{m+1}\lceil J)$ is 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 $\mathcal{M}_{J'}\lceil J$. □

Now we want to exploit the Path projection lemma to prove a similar result on fair fullpaths, which are the most natural descriptions of complete system executions. We adopt the concept of fairness from Chapter 4, where a fair path contains infinitely many transitions performed by each dynamic sync-automaton.

We remark that this concept of fairness in connection with the reasoning about instances is a little different from the concept of fairness for sync-programs in the static case. In the dynamic setting, it says that for each type i of dynamic sync-automata it is infinitely often the case, that some instance of type i performs the move. No that there is no guarantee of an infinite behaviour of all instances, which might not entirely correspond to our intuitive interpretation of the fairness. However, for specifying the fairness as a requirement of an infinite behaviour of each instance, there is not enough information in the semantics of the dynamic sync-program, since the semantics is built over multisets and no distinction of individual instances is made. Nevertheless, the present notion of fairness is sufficient for the scope of the modular verification and permits to prove the Fullpath projection lemma.

Lemma 7.16 (Fullpath projection)**.** *Let I be an index set and $\mathcal{M}_I$ semantics of dynamic sync-program P^I. For every $J \subseteq I$ if π is a fair fullpath in $\mathcal{M}_I$ then $\pi\lceil J$ is a fair fullpath in $\mathcal{M}_{J'}\lceil J$, where J' is the creation set of J and $\mathcal{M}_{J'}$ is the semantics of sync-program $P^I\upharpoonright J'$.*

Proof. By path projection lemma $\pi\lceil J$ it is a path in $\mathcal{M}_{J'}\lceil J$. 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 $\mathcal{M}_{J'}\lceil J$. From the definition of fairness follows that every fair path is infinite, i.e. it is a fullpath. □

7.5.3 Property preservation theorem

The following theorem guarantees that successful verification of a DACTL^-_J property in $M_{J'}\lceil J$, where J' is the creation set of J, amounts to its successful verification in M_I. We first prove an auxiliary lemma, that shows the preservation of propositional formulae. Then we prove the theorem itself.

Lemma 7.17 (Preservation of propositional formulae)**.** *Let $J \subseteq I$ be an index set, $\mathcal{M}_I$ be an I-multistructure and s be an I-multistate. If f is a state formula from DACTL^-_J, $\mathcal{M}, s \vDash f$ implies $\mathcal{M}, s\lceil J \vDash f$.*

Proof. By induction on the construction of f for all σ.

The cases of $f = true$ and $f = false$ are trivial.

The case of $f = a(k)$ where $a \in I$ and $k \in Id$ follows. Suppose that $\mathcal{M}, \sigma, s \vDash a(k)$. By definition of DACTL$^-$ we have that $s'(a)$ where $s' = \sigma(k)$. Since all elements of multiset s belonging to states of sync-automata of type i are preserved in $s\upharpoonright J$, we have that $\mathcal{M}, \sigma, s\upharpoonright J \vDash a(k)$. The cases of $f = true_i(k)$ and $f = true_i(k)$ are analogous.

The cases of $f = g_1 \wedge g_2$, $f = g_1 \vee g_2$, $f = g_1 \rightarrow g_2$ and $f = g_1 \leftrightarrow g_2$ are immediate from the induction hypothesis.

The case of $f = \forall k{:}i.g$. Suppose that $\mathcal{M}, \sigma, s \vDash \forall k{:}i.g$. By the definition of DACTL$^-$ for all elements s' from $\mathcal{P}(AP_i)$ in multiset $s - Im(\sigma)$ holds $\mathcal{M}, \sigma \uplus (k, s'), s \vDash f$. Since all elements of multiset s belonging to states of sync-automata of type i are preserved in $s\upharpoonright J$, we have that $\mathcal{M}, \sigma \uplus (k, s'), s\upharpoonright J \vDash f$. Then by definition of DACTL$^-$ $\mathcal{M}, \sigma, s\lceil J \vDash \forall k{:}i.g$. The case for $f = \exists k{:}i.g$ is analogous. □

The main theorem follows.

Theorem 7.18 (Property preservation)**.** *Let I be an index set and $\mathcal{M}_I$ semantics of sync-program P^I and s an I-multistate. Let $J \subseteq I$ be an index set, J' its creation set and $\mathcal{M}_{J'}$ is the semantics of sync-program $P^I\upharpoonright J'$. For every DACTL^-_J property f,*

$$\textit{if } \mathcal{M}_{J'}\lceil J, s\lceil J \vDash f \textit{ then } \mathcal{M}_I, s \vDash f.$$

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

f is a propositional formula. Follows from the lemma about preservation of propositional formulae.

$f = g \wedge h$. From the assumption $\mathcal{M}_{J'}\lceil J, s\lceil J \vDash_\Phi g \wedge h$ by DACTL$^-$ semantics, $\mathcal{M}_{J'}\lceil J, s\lceil J \vDash_\Phi g$ and $\mathcal{M}_{J'}\lceil J, s\lceil J \vDash_\Phi h$. By induction hypothesis $\mathcal{M}_I, s \vDash_\Phi g$ and $\mathcal{M}_I, s \vDash_\Phi h$. Hence, $\mathcal{M}_I, s \vDash_\Phi g \wedge h$. Case $f = g \vee h$ is proved analogously.

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

1. $\mathcal{M}_{J'}\lceil J, \pi\lceil J \vDash_\Phi Gg$. Let t be any state along π. By DACTL$^-$ semantics $\mathcal{M}_{J'}\lceil J, t\lceil J \vDash_\Phi g$. by induction hypothesis we have $\mathcal{M}_I, t \vDash_\Phi g$. Since t was an arbitrary state of π, we get $\mathcal{M}_I, \pi \vDash_\Phi Gg$ and thus $\mathcal{M}_I, \pi \vDash_\Phi g\ U_w\ h$.

2. $\mathcal{M}_{J'}\lceil J, \pi\lceil J \vDash_\Phi [g\ U\ h]$. 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 $\mathcal{M}_I, s^{m'} \vDash_\Phi h$. 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 $\mathcal{M}_{J'}\lceil J, s^m\lceil J \vDash_\Phi g$, hence by induction hypothesis $\mathcal{M}_I, s^m \vDash_\Phi g$. By DACTL$^-$ semantics we get $\mathcal{M}_I, \pi \vDash_\Phi g\ U\ h$.

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

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

7.6 Case Study: EGF Signalling Pathway

In Biology, signal transduction refers to any process by which a cell converts one kind of signal into another. Signals are typically proteins that may be present in the environment of the cell. In order to recognise that a signal is available in the environment, a cell exposes some receptors on its external membrane. A complex signal transduction cascade (Fig. 7.2) that modulates cell proliferation is based on a family of receptors called epidermal growth factor receptors (EGFRs) that are produced by specific genes in the DNA (through the RNA) and are located on the cell surface. Receptors are activated by the binding with a specific ligand (epidermal growth factor, EGF) to form a ligand-receptor complex. After activation, EGFR undergoes a transition from a monomeric form to a dimeric one (consisting of two ligand-receptor complexes). EGFR dimerisation enables activation of effector proteins, which initiates several signal transduction cascades, leading to cell proliferation. Occasionally, ligand-receptor dimers are internalised in endosomes and consequently targeted for degradation inside the lysosomes.

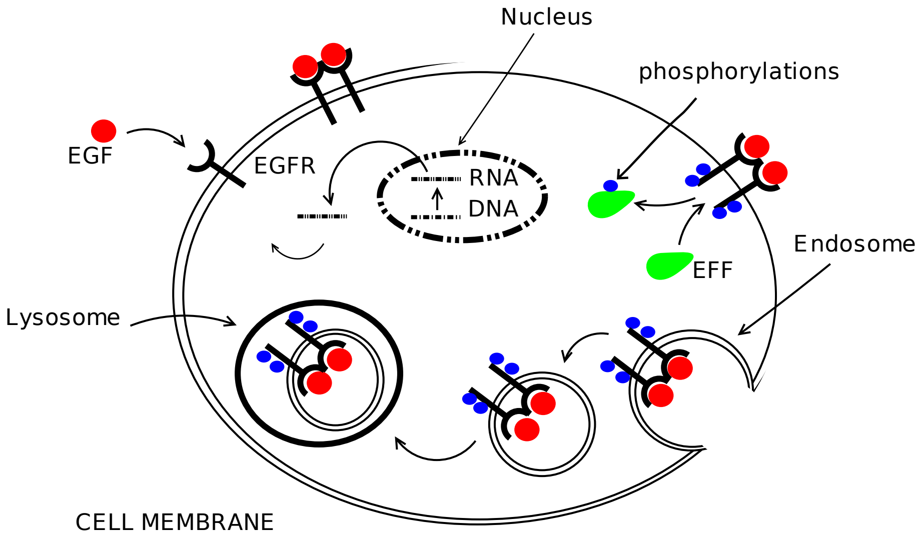


Figure 7.2: The first steps of the EGF pathway

The result of modelling the described steps of the EGF signalling pathways (with some simplifications) is dynamic sync-program P^I.

The index set I consists of five elements $\{nucl, egfr, endo, lyso, eff\}$. There is a dynamic sync-automaton for each instance of each biological component. In particular the nucleus is modelled by P^I_{nucl}, with $AP_{nucl} = \{Dna\}$; EGFR by P^I_{egfr} with $AP_{egfr} = \{Egfr, On\_membr, Bound, Dim, Cr\_endo\}$; effectors by P^I_{eff} with $AP_{eff} = \{Effp\}$; endosomes P^I_{endo}, with $AP_{endo} = \{Endo, In\_lyso\}$; and finally lysosomes by P^I_{lyso}, with $AP_{lyso} = \{Lyso\}$.

Sync-automaton A_{nucl} (depicted in Figure 7.3) describes the DNA/RNA activity in the nucleus, that is the production of receptors and effectors encoded as creation of the respective dynamic sync-automata bound to $NOSYNC$ moves.

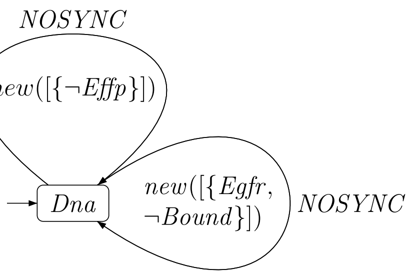


Figure 7.3: Nucleus – A_{nucl}.

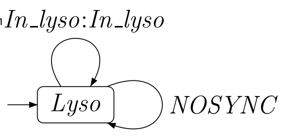


Figure 7.4: Lysosome – A_{lyso}.

Sync-automaton A_{egfr} (Figure 7.5) describes a receptor. It starts by binding an EGF signal, as modelled by the first $NOSYNC$ move. (Absence of EGF signals in the environment is modelled by the self-looping $NOSYNC$ move in the initial state.) Subsequently, it synchronises with another sync-automaton of the same type to form a dimer (represented by two automata in a state in which *Dim* holds). The dimer can then interact with effectors. A dimer can be internalised by an endosome. The endosome is created by an automaton in a state in which *Dim* holds, by synchronising with another automaton in the same state (assumed to be the other component of the dimer). In order to ensure that only one endosome is created, the moves of the two components of the dimer are kept distinct. Finally, a dimer synchronises with an endosome in order to be destroyed (inside the lysosome).

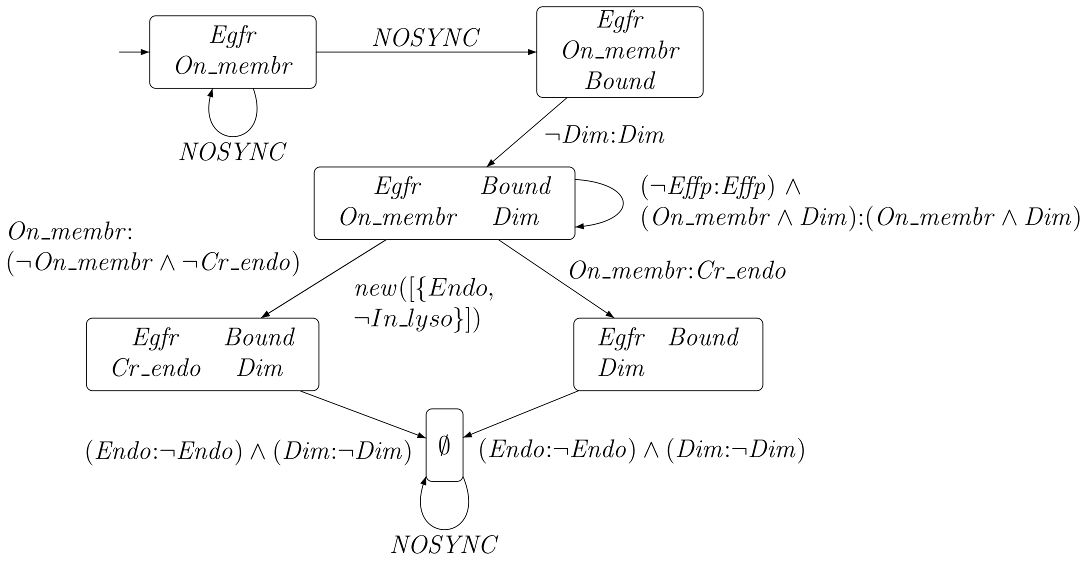


Figure 7.5: EGFR – A_{egfr}.

An effector protein is modelled by sync-automaton A_{eff} (Figure 7.6) that, once created, can either do nothing (the *NOSYNC* self-looping move) or synchronise with a dimer in order to move to a state in which it is phosphorylated.

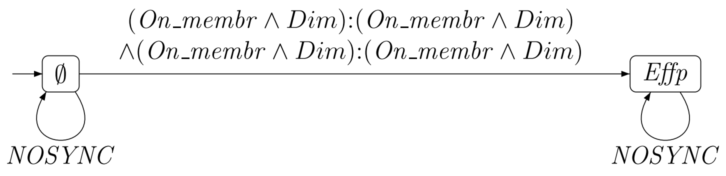

(On_membr ∧ Dim):(On_membr ∧ Dim)
∧(On_membr ∧ Dim):(On_membr ∧ Dim)
∅
Effp
NOSYNC
NOSYNC

Figure 7.6: Effector – A_{eff}.

Finally, sync-automaton A_{endo} (Figure 7.7) describes an endosome that synchronises first with a sync-automaton A_{lyso} (Figure 7.4) to model entrance into the lysosome and then with two automata assumed to represent the dimer it is carrying.

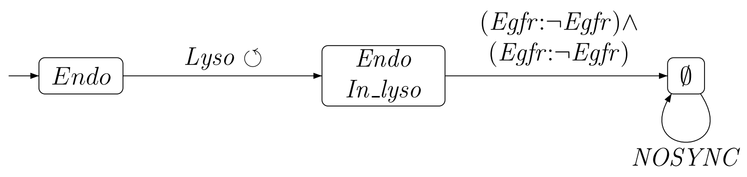


Figure 7.7: A_{endo}.

The sync-program corresponding to the initial configuration of the model is $P^I = (\{\{Dna\}, \{Lyso\}\}, P^I_{nucl} || P^I_{lyso})$.

To illustrate the Petri net representation dynamic sync-programs, we provide the Petri net representing dynamic sync-program P^I on Figure 7.8. Note that the colours are used only to visually differentiate places that originate from one component. These colours are, however, not a part of the formalism. The figure was prepared by using the tool Snoopy [81].

Now we discuss some properties that could be verified on the model in a modular way. We indicate what is the fragment of the model that suffices for the verification of the property. Any verification method can be applied to the identified fragment in order to investigate the property in question and the result of the verification is assured.

We would like to verify the property *"The number of instances of A_{egfr} in which Dim holds is always even"*. Since we cannot reason on the numbers of instances, what we can actually prove is the following weaker property (D1): *"In every reachable state Dim holds in either zero or at least two instances of A_{egfr}"*. This property is expressed by the formula

$$AG(\ (\forall k{:}egfr.false_{egfr}(k)) \vee (\forall k{:}egfr.\neg Dim(k)) \vee \\ (\exists k{:}egfr.\exists k'{:}egfr.(Dim(k) \wedge Dim(k')))\ ) \tag{D1}$$

that can be verified on the semantics of P^{egfr}.

A property (D2) *"If an endosome is destroyed, then at the same time two receptors are destroyed"* is expressed by the formula

$$A[\ (\forall k{:}endo.false_{endo}(k))\ U_w \\ A[\ (\exists k_1{:}endo.\exists k_2{:}egfr.\exists k_3{:}egfr.Endo(k_1) \wedge Egfr(k_2) \wedge Egfr(k_3))\ U \\ (\exists k_1{:}endo.\exists k_2{:}egfr.\exists k_3{:}egfr.\neg Endo(k_1) \wedge \neg Egfr(k_2) \wedge \neg Egfr(k_3))\ ]\ ] \tag{D2}$$

that can be verified on the semantics of $P^{endo,egfr}$.

A property (D3) *"If EGF signals are present in the environment, eventually either effectors are phosphorylated or endosomes are created."* is a property stating the correct functioning of the model. It is expressed as the formula:

$A[\ ((\forall k{:}endo.false_{endo}(k)) \vee (\exists k{:}egfr.\neg Bound(k)))\ U_w$

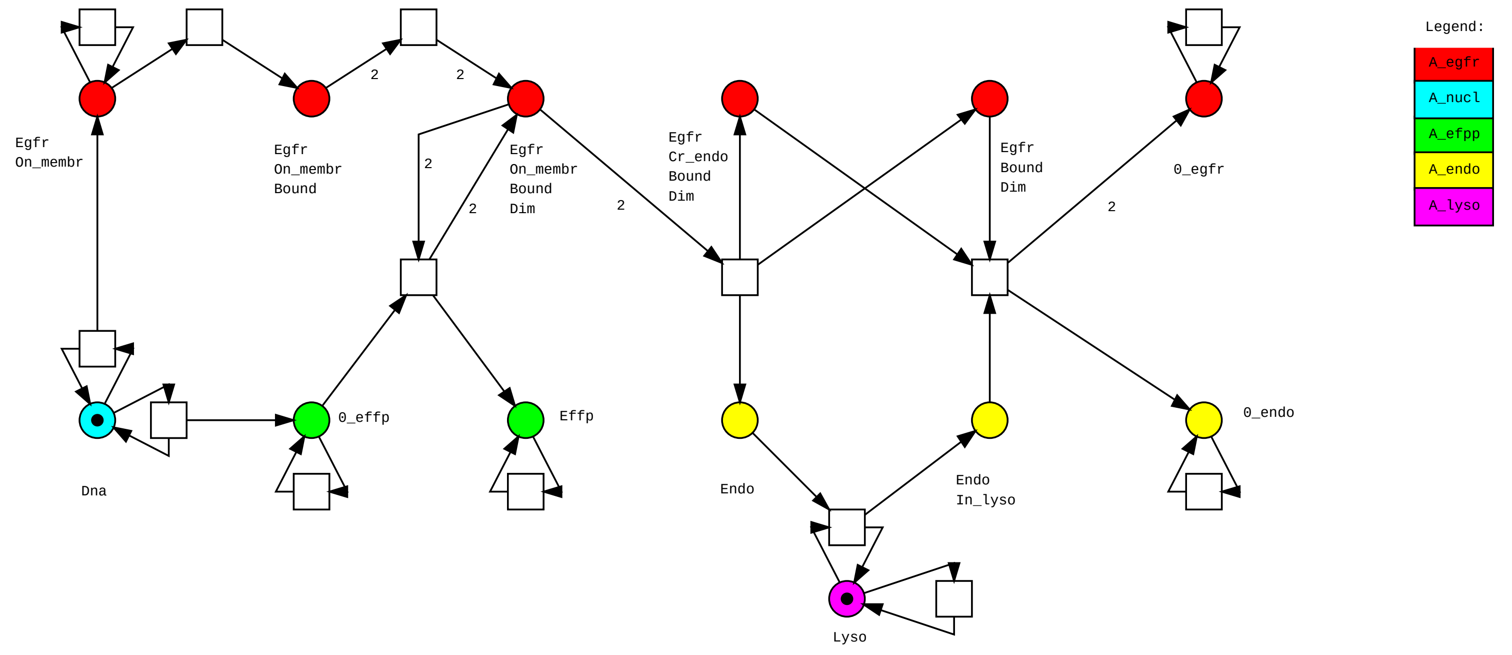


Figure 7.8: Petri net representation of the EGF Signalling Pathway

$$AF((\exists k'{:}\mathit{eff}.\mathit{Effp}(k')) \vee (\exists k''{:}\mathit{endo}.\mathit{true}_{\mathit{endo}}(k''))) \;] \tag{D3}$$

that can be proved to hold on the semantics of $P^{\mathit{egfr},\mathit{eff},\mathit{endo}}$.

The property *"If an endosome is created, it will be eventually destroyed"* cannot be expressed in DACTL$^-$ because the semantics does not possess enough information in order to permit tracing a particular instance over time. We approximate it by a weaker property (D4) stating *"If an endosome is created, an endosome will be eventually destroyed"*. This property is expressed by the formula

$$A[(\forall k{:}\mathit{endo}.\mathit{false}_{\mathit{endo}}(k)) \; U_w \; AF(\exists k{:}\mathit{endo}.\neg \mathit{Endo}(k))] \tag{D4}$$

that can be verified on the semantics of P^{endo}.

Chapter 8

Modular Verification of Pathways

The purpose of this chapter is to show how to verify a particular class of biological systems. It also provides a proof of concept for the scalability of the approach.

8.1 Pathways

In biochemistry, metabolic pathways are series of biochemical reactions occurring within a cell. The reactions are connected by their intermediates: the products of one reaction are the substrates for subsequent reactions.

In this section we give assumptions under which it is possible to decompose a pathway into components. The identified components will be modelled by using sync-automata in the next section.

Syntactic Assumptions Given an infinite set of species S, let us assume biochemical reactions constituting a pathway to have the following form:

$$r_1, \ldots, r_n \ \rightarrow \ p_1, \ldots, p_n \ \{ \, c_1, \ldots, c_m \, \}$$

where r_j, p_j and c_j, for suitable values of j, are all in S. We have that r_js are reactants, p_js are products and c_js are catalysts of the considered reaction. Consequently, given a reaction R we define $re(R) = \{r_1, \ldots, r_n\}$, $pro(R) = \{p_1, \ldots, p_n\}$ and $cat(R) = \{c_1, \ldots, c_m\}$. We denote the set of species involved in reaction R as $species(R) = re(R) \cup pro(R) \cup cat(R)$.

Note that in a reaction there are as many reactants as products. In addition, we assume a positional correspondence between reactants and products, namely we assume product p_j to be the result of the transformation of reactant r_j by the reaction.

In some sense, a species can be seen as a part of a "state" or "configuration" of a more general system component, and a reaction can be seen as a synchronised state change of a set of system components. It is usually possible to translate any reaction of a pathway into the assumed "normal form" with the same number of reactants and products. For the moment it is left to the modeller to perform such a preprocessing.

A pathway P is simply a set of reactions. Given a pathway P, we can infer the set of species involved in it as $\mathit{species}(P) = \bigcup_{R \in P} \mathit{species}(R)$.

Semantic Assumptions As regards the dynamics of a pathway we consider the following assumptions and abstractions.

In order to be able model a pathway by a sync-program, it is necessary to abstract from quantities or concentrations of species. Therefore species can be only either present or absent.

We need choose an interpretation of the qualitative dynamics of a reaction depending on whether it is catalysed. A reaction without catalysts creates the products but does not consume the reactants, as some quantity of reactants may remain in the quantitative view. On the other hand, a reaction favoured by catalysts tends to be performed as long as there are reactants (in a constant presence of catalysts). Therefore a reaction with catalysts creates the products and consumes the reactants. Consuming all reactants might be not realistic if concentrations of reactants differ significantly, but this is an abstraction we choose to make. It also implies that a reversible reaction in which both directions are catalysed, which frequently occurs in biological pathways, oscillates between two states. This could be handled differently (by considering some kind of dynamic equilibrium state), but we leave this problem as future work.

Lastly, we assume that all of the catalysts are required to be present in order for the reaction to take place. Alternative combinations of catalysts that may enable the reaction should be modelled as different reactions having the same reactants and products.

8.2 From Reactions to Sync-Programs

Let us fix a pathway P. The assumptions made in the previous section allow us to identify the set of species with the set of atomic propositions of sync program P^I representing pathway P. That is we have that $AP_I = \mathit{species}(P)$.

Sync-automata Identification The index set I containing indices of components is a priori not known, since the implicit components of the pathway are not given. Now we will present an algorithm that given a pathway P returns the index set I along with the partition of the set AP_I of atomic proposition to sets AP_i for each component i.

We illustrate the intuitive idea on an example. Each reaction can be seen as a synchronisation of components. For example reaction $r_1, r_2 \;\rightarrow\; p_1, p_2 \;\;\{\; c \;\}$ can be interpreted as a synchronisation of three constituents: one that changes its state from a state where r_1 holds into a state where p_1 is present and r_1 is not; another component that changes its state from a state where r_2 holds to a state where p_2 is present and r_2 is not; and a component which participates passively and stays in a state where c is present.

Since we suppose that only one reaction takes place at a time in the whole system, the states of all the components do not change other than those involved in the reaction in the way we described.

From the example we can see, that species r_1 and p_1 belong to the same component. Similarly r_2 belongs to the component that contains p_2, while c is from a separate one.

The algorithm follows. We start by assuming that each atomic proposition belongs to a different AP_i and we refine this assumption by iterating over the reactions constituting P. The result of the algorithm is a mapping *map* assigning each species to its sync-automaton index.

Algorithm 1 Algorithm to partition atomic proposition (species) into different sync-automata

```
Let $map : AP_I \mapsto I$ be an injective mapping
for all $R$ in $P$ do
    for all $r_j$ in $reactants(R)$ do
        $map := \begin{cases} p_j \mapsto map(r_j) & \\ s \mapsto map(s) & \forall s \in AP_I; s \neq p_j \end{cases}$
    end for
end for
return $map$
```

The algorithm updates the mapping by unifying the elements assigned to reactants and products in the same position in a reaction, and this is done for all reactions in the pathway.

The index set of sync-program P^I is the image of mapping *map*. The sets A_i for $i \in I$ of atomic propositions of each sync-automaton are obtained in the following way: $AP_i = \{p \in AP_I \mid map(p) = i\}$. States of sync-automaton P_i are as usual subsets of AP_i.

Specification of Moves Now, we can obtain moves of the sync-automata describing system components from reactions in which some of its atomic propositions are present as reactants and products.

We express the set of moves in terms of meta-moves, namely of moves in which the source and the target states are not completely specified. Actual moves will be obtained by instantiating the unspecified portion of the source state of each meta-move in every possible way, and the corresponding target state in the same manner. The set of meta-moves of sync-automaton P_i, denoted $mm(P_i)$, is the least set satisfying the following rules, where $re_i = re(R) \cap AP_i$, $pro_i = pro(R) \cap AP_i$ and $cat_i = cat(R) \cap AP_i$:

- For every $R \in P$ such that $re_i \neq \emptyset$ and $cat(R) = \emptyset$ we have that $mm(P_i)$ contains the following meta-move:

  $$(\bigwedge_{r \in re_i} r) \wedge \neg(\bigwedge_{p \in pro_i} p) \xrightarrow{r_{i_1}:p_{i_1} \wedge \ldots \wedge r_{i_k}:p_{i_k}} (\bigwedge_{r \in re_i} r) \wedge (\bigwedge_{p \in pro_i} p)$$

  where $re(R) \setminus re_i = \{r_{i_1}, \ldots, r_{i_k}\}$.

- For every $R \in P$ such that $re_i \neq \emptyset$ and $cat(R) \neq \emptyset$ we have that $mm(P_i)$ contains the following meta-move:

  $$(\bigwedge_{r \in re_i} r) \wedge \neg(\bigwedge_{p \in pro_i} p) \wedge (\bigwedge_{c \in cat_i} c) \xrightarrow{r_{i_1}:p_{i_1} \wedge \ldots \wedge r_{i_k}:p_{i_k} \wedge c_1:c_1 \wedge \ldots \wedge c_l:c_l} (\bigwedge_{r \in re_i} \neg r) \wedge (\bigwedge_{p \in pro_i} p) \wedge (\bigwedge_{c \in cat_i} c)$$

where $re(R) \setminus re_i = \{r_{i_1}, \ldots, r_{i_k}\}$ and $cat(R) = \{c_1, \ldots, r_l\}$.

These items define moves corresponding to reactions in the case of absence of catalysts (i.e. reactants are not consumed) and presence of catalysts (i.e. reactants are consumed).

Note that moves test for the absence of some products to avoid occurrence of a reaction that does not change the state of the system.

As only fair paths are considered in our approach to represent a correct behaviour of the system, we need to ensure that each behaviour that we intuitively consider correct is present as a fair path in the semantics of the system. In other words we have to extend unfair paths representing correct behaviour so that they become fair.

It should be noted that this is not possible in general, since the considered form of fairness (unconditional fairness) is not *feasible* [3], or *machine-closed* [1]. However, in the present case of components of pathways it is possible to perform such an extension empirically in an automatic way by exploiting the possibility of NuSMV as seen in the following section.

8.3 From Reactions to NuSMV

In order to implement the sync-program P^I obtained from the pathway P we could use the translation *transl* defined in Section 5.1. In this chapter we choose a different, ad-hoc method that gives rise to a more succinct model leading to more efficient verification.

The reason we can apply a different translation method is the special way in which the sync-automata created from a pathway are synchronised. Each synchronisation of sync-automata is induced by an occurrence of some reaction from the pathway. In the translation from Section 5.1 this would correspond to a series of moves of the involved sync-skeletons following the protocol. On the other hand, since the reactants, products and catalysts of the reactions provide extra information, the state change corresponding to the reaction taking place can be performed in one step.

Furthermore note that one reaction corresponds to more combinations of moves in the related sync-automata. Since the present ad-hoc translation introduces one transition for each reaction, it provides a simpler NuSMV description of the program.

It should be noted that in the whole translation process of pathway P to a NuSMV model the sync-automata are not necessary. The partitioning of atomic propositions into sync-automata is however crucial for the projections used in the modular verification.

Species involved in a pathway are translated into boolean variables (where value `TRUE` means that the species is currently present in the system) whereas reactions are translated into transitions. Given a pathway P such that $species(P) = \{s_1, \ldots, s_N\}$, the translation produces a NuSMV model that starts as follows:

```
MODULE main

VAR
  s1 : boolean;
  ....
  sN : boolean;
```

In addition, let $\{a_1, \ldots a_X\}$ be the indices obtained by applying to P the partitioning of atomic propositions defined in the previous section. We also add the following boolean variables that will allow us to track moves of sync-automata, later to be used in fairness constraints.

```
VAR
  a1_moves : boolean;
  ....
  aX_moves : boolean;
```

Subsequently, for each reaction $R_i \in P$ we define macros `sourcestate-Ri`, `targetstate-Ri` and `nochange-Ri`. The first two represent conditions to be satisfied in the source state and in the target state, respectively, of a transition representing the reaction. The third states that variables not involved in the reaction must not change. In the NuSMV model for each reaction $R_i = r_1, \ldots, r_n \;\rightarrow\; p_1, \ldots, p_n$, where $species(P) \setminus species(R_i) = \{nc_1, \ldots, nc_{N'}\}$, we have:

```
DEFINE
  sourcestate-Ri := r1 & ... & rn & !(p1 & .... & pn);
  targetstate-Ri := r1 & ... & rn & p1 & .... & pn
                    &  ai1_moves & .... &  aiY_moves
                    & !aj1_moves & .... & !ajZ_moves;
  nochange-Ri := nc1 = next(nc1) & .... & ncN' = next(ncN');
```

where $\{a_{i1}, \ldots, a_{iY}\}$ are the sync-automata involved in the reaction, $\{a_{j1}, \ldots, a_{jZ}\} = \{a_1, \ldots, a_X\} \setminus \{a_{i1}, \ldots, a_{iY}\}$.

In addition, for each reaction $R_i = r_1, \ldots, r_n \;\rightarrow\; p_1, \ldots, p_n \;\{\, c_1, \ldots, c_m \,\}$, with $m > 0$, where $species(P) \setminus species(R_i) = \{nc_1, \ldots, nc_{N'}\}$, we have:

```
DEFINE
  sourcestate-Ri := r1 & ... & rn & !(p1 & .... & pn) & c1 & ... & cm;
  targetstate-Ri := !r1 & ... & !rn & p1 & .... & pn & c1 & ... & cm
                    & ai1_moves & .... & aiY_moves
                    & !aj1_moves & .... & !ajZ_moves;
  nochange-Ri := nc1 = next(nc1) & .... & ncN' = next(ncN');
```

where, as before, $\{a_{i1}, \ldots, a_{iY}\}$ are the sync-automata involved in the reaction and $\{a_{j1}, \ldots, a_{jZ}\} = \{a_1, \ldots, a_X\} \setminus \{a_{i1}, \ldots, a_{iY}\}$.

These definitions allow us to define transitions of the model in a simple way. For each reaction $R_i \in P$ we write:

```
DEFINE
  react-Ri := (sourcestate-Ri & next(target-state-Ri) & nochange-Ri);
```

Now, add self-looping transitions in some states to transform unfair paths into fair ones. The idea is that we should add a self-loop in every state where a sync-automaton is blocked, namely none of reactions it is involved in is enabled.

For each species s_i we define a macro meaning that some reaction having s_i as a reactant can take place.

```
DEFINE
  canmove-Si := sourcestate-Rj1 | ... | sourcestate-RjM';
```

where $\{r_{j1}, \ldots, r_{iM'}\}$ are reactions having s_i as a reactant.

Next we specify a macro for each automaton meaning that the automaton cannot participate in any of its reactions. Note that a specification of this macro is easy in NuSMV since we can use negation on state conditions.

Then macro follows saying that automaton with index a_i is the only one that moves.

```
DEFINE
  blocked-ai := !(canmove-Sj1 | ... | canmove-SjM');
  movesonly-ai := ai_moves & !ak1_moves & akY_moves;
```

where $\{s_{j1}, \ldots, s_{jM'}\}$ are the species of automaton with index a_i and $a_{k_1}, \ldots, a_{k_Y}$ are indices of all the other automata in the system. The last macro states that no species is changed after the transition.

```
DEFINE
  nochange-all := s1 = next(s1) & .... & sN = next(sN);
```

where $species(P) = \{s_1, \ldots, s_N\}$.

Now we can define the self-loop of automaton with index a_i.

```
DEFINE
  selfloop-ai := (blocked-ai & movesonly-ai & nochange-all);
```

Finally, the transition relation is specified as follows

```
TRANS
  react-R1 | ... | react-RM |
  selfloop-a1 | ... | selfloop-Au
```

where $\{r_1, \ldots, r_M\}$ are all reactions in P and $a_1, \ldots, a_Y$ are indices of all automata in the system.

The initial state of the system needs to be set. However, choosing the initial assignment to variables `s1`, . . . ,`sN` is dependent on the concrete pathway under study. Thus we postpone the choice to the next section.

To conclude, we write the fairness constraint stating that each automaton a_i has to move infinitely many times in a fair path.

```
FAIRNESS
  ai-moves
```

Note that this variable is set to true only if a reaction of automaton a_i takes place or automaton performs a self-loop when blocked.

A syntactic projection onto subset J of I is rather simple – leave out automata from $I \setminus J$, their species, reactions and fairness constraints.

8.4 Case Study: MAP Kinase Cascade Activated by Surface and Internalised EGF receptors

We apply our modular verification approach to a computational model of the MAP kinase cascade activated by surface and internalised EGF receptors, proposed by

Schoeberl et al. in 8.1. The modelled system is essentially the same as the one seen in the case study we considered in 7.6, but in this case the model, depicted in figure 8.1 from [82], is much more detailed. In particular, it includes:

(i) a detailed description of all of the reactions involved in the pathway,

(ii) a detailed description of the signalling activity performed by the internalised receptors, and

(iii) a model of the the MAP kinase cascade that is influenced by the EGF signalling pathway.

As regards (i), the model includes reactions that involve active EGF receptors and several effectors named GAP, ShC, SOS, Grb2, RasGDP/GTP and Raf. As regards (ii), reactions analogous to those involving EGF receptors placed on the external cell membrane are considered for internalised receptors (denoted EGFRi). Finally, as regards (iii), reactions involving MEK and ERK proteins (and their phosphorilated variants MEK-P, MEK-PP, ERK-P and ERK-PP) are included.

Model. Let us denote by P the pathway described above. A manual preprocessing is performed consisting only of considering Phosphatase1, Phosphatase2, Phosphatase3, Raf* and MEK-PP as catalysts rather than reactants and products in the reactions of the MAP kinase cascade. The model satisfies the assumptions made in Section 8.1. It is made up of 143 species and 80 reactions.

In an automatic way we transform P according to the translation described in Section 8.3. After performing the automata identification procedure described in Section 8.2, 14 automata are identified. It remains to specify the initial state of the model.

8.4.1 Finding a Suitable Initial State

We adopt a semi-automatic procedure to find an initial state for the pathway model. The idea is the following: for each species s in $species(P)$, if there is no reaction creating it (i.e. if $s \notin \bigcup_{R \in P} prod(R)$) then in the initial state we set the corresponding boolean variable to true. This means that species that cannot be produced are assumed to be present in the initial state. Otherwise their presence in the model would not be meaningful. Subsequently, we resort again to the partitioning of species into sync-automata to find other variables to be set to true. In particular, we find those sync-automata containing no atomic proposition set to true in the previous phase. These sync-automata must contain loops, hence we choose manually some of their boolean variables to be set to true. All of the other variables are set to false, in order to ensure that the initial state of each component corresponds to one species being present.

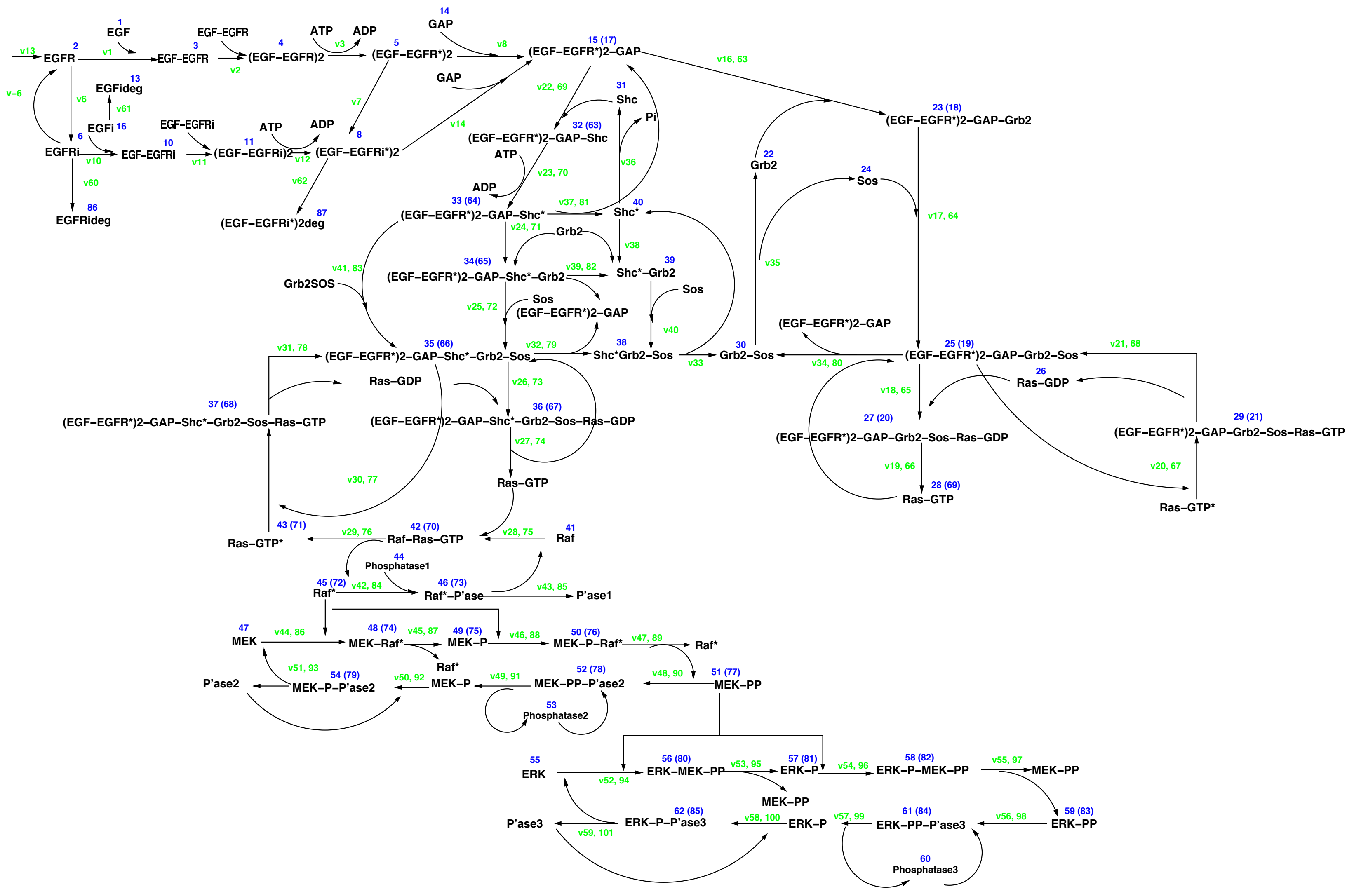


Figure 8.1: Scheme of the EGF receptor-induced MAP kinase cascade [82]

8.4.2 Interaction Graph

In modular verification the choice of the submodel to be used for verification of a property is up to the person performing the verification. The property preservation theorem guarantees satisfaction of properties that are true in a submodel, but does not say anything about those that do not hold. To reject the property as false one would need to establish falsehood in the complete model. However when the truth is presumed, it might be desirable to try the verification on another (perhaps larger) submodel. A naive approach is to start by considering the system components that are mentioned in the property and try incrementally to add components one by one.

A somewhat more sophisticated method can be based on analysing the interaction graph of the model that can be automatically inferred. An interaction graph is a directed graph in which vertices are system components (elements of I) and edges connect components that are involved together in a synchronisation.

It is easy to see that the subgraph corresponding to a reasonable submodel needs to be connected for other than trivial properties.

In the particular case of pathways, we connect components that are involved together in a reaction. If two components are both involved as reactants (and consequently products), the edge connecting them will not be oriented (we have both directions). If one of the two is involved as reactant and the other either as catalyst or as inhibitor, then the edge will start from the vertex representing the latter to the vertex representing the former. There is no edge between vertices representing components involved in the same reactions only as catalysts and inhibitors.

On the Figure 8.2 we can see the interaction graph of pathway P. Each node of the graph contains the index that has been assigned to the automaton by the algorithm from Section 8.2 along with its intuitive name. We can identify enzymes like Phosphatase1, Phosphatase2 and Phosphatase3. We can see the first part of the pathway corresponding to the EGF receptor and its interaction with effectors, and its connection to the MAP kinase cascade through the component RasGDP.

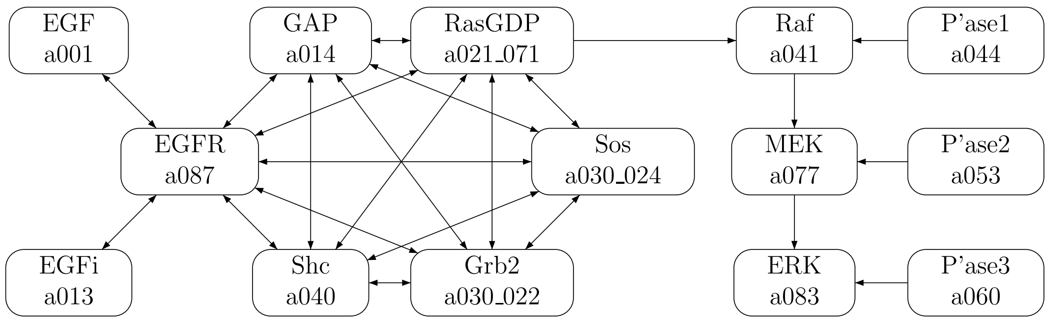


Figure 8.2: Interaction graph

8.4.3 Experiments

The final product of the MAP kinase cascade activated by surface and internalised EGF receptors is species ERK-PP. As can be seen in the interaction graph and in the diagram in Figure 8.1, some components are involved in complex interactions. This is true in particular for components EGFR, GAP, RasGDP, Sos, Shc and Grb2 which form a clique in the interaction graph. We are interested in understanding whether all of these components are really necessary in order to obtain the final product of the pathway.

The idea is to test whether the final species is produced when the components of interest are assumed one by one as absent. The property to be tested should be

$$AF(ERK\!-\!PP|ERK\!-\!PPi) \tag{E1}$$

where ERK-PPi is the ERK-PP produced by the internalised branch of the pathway.

We could try to verify this property by using the modular verification on a suitably chosen subset of the components, but the verification would fail since the property does not hold in the whole model. The reason why the property does not hold are our assumptions on the behaviour of catalysed reactions. In particular, the reactants of catalyzed reactions are always completely consumed, and this introduces looping behaviours of the MAP kinase part of the pathway, which prevent creation of ERK-PP.

Note that the components we are interested in are not directly involved in the MAP kinase cascade. Therefore, in order to establish whether they are necessary it might be enough to check if they are necessary to produce Raf*, the activator of the MAP kinase cascade. In fact, we can prove the property *"Raf* is a necessary predecessor of MEK-PP"*. The same holds for their internalised couterparts.

The property is an instance of the pattern (S3) from the category "sequence" in Section 2.8, in particular *"A state S_2 is reachable and is necessarily preceded at some time by a state S_1"*. Unfortunately, the associated formula $EF(S_2) \wedge \neg E(\neg S_1 \; U \; S_2)$ is not from ACTL$^-$ since the first conjunct is not.

However, if we weaken the desired property and sacrifice reachability, that is we consider the property *"A state S_2 is necessarily preceded at some time by a state S_1"*, we get the formula $\neg E(\neg S_1 \; U \; S_2)$ which is equivalent to the ACTL$^-$ formula $A[\neg S_2 \; U_w \; (S_1 \wedge \neg S_2)]$ (from the equivalence $A[f \; U_w \; g] \leftrightarrow \neg E[\neg g \; U \; \neg(f \vee g)]$). Hence, we successfully apply the modular verification to the following formula (we do not use the latter equivalent one since in NuSMV the U_w operator is missing)

$$\neg E(\neg Raf^* \; U \; ERK\!-\!PP) \tag{E2}$$

The fact that Raf* is a precursor of ERK-PP allows us to reason about the indispensability of EGFR, GAP, RasGDP, Sos, Shc and Grb2 on property

$$AF(Raf^*|Raf^*i) \tag{E3}$$

rather than on (E1).

In order to assess whether each component is necessary to produce Raf*, we check for each component the property (E3) on a suitably chosen projection.

The property is checked by considering two initial states, the first in which the involved component is assumed to be present and the second in which the component is assumed to be absent. The absence of the component is modelled by setting to false all of its atomic propositions in the initial state of the system.

Property (E3) can be successfully verified in the case that all the components are assumed to be present. This means that all components contribute (actually, are not obstacles) to the production of Raf*.

In the cases where the components are selectively removed from the system, we have that the modular verification allows us to prove the property (E3) only in the case of Shc. This implies that Shc is not a necessary component, since Raf* is always produced even if Shc is absent from the system. The modular verification in the cases of the other components does not succeed, therefore we cannot infer anything about their indispensability yet. However, rather than proving (E3), we can try with the following property

$$AG\neg(Raf^* \vee Raf^* i) \tag{E4}$$

which implies that (E3) does not hold. This property holds in all the previously considered projections in which the verification of (E3) failed. Indeed, this proves that all the involved components are necessary to activate the MAP kinase cascade.

In Table 8.1 we summarise the property verification results and compare the modular verification times with those that would have been obtained by verifying the properties on the whole model. The projection $proj_1$ consists of components RasGDP, Raf, EGFR, GAP, Grb2, Shc and Sos. We did not perform verification of (E2) on the whole model as the scope of the analysis is (E3) and (E4).

8.5 Discussion

It is worth pointing out that the fairness we consider might be too weak since it permits some behaviours that intuitively should not be considered fair. We recall that the fairness requires that every component is executed infinitely many times in a fair execution. In our case, components synchronise by performing chemical reactions. We illustrate the issue on the following part of the pathway P.

$$\text{Raf, Ras-GTP} \rightarrow \text{Raf-Ras-GTP, Raf-Ras-GTP'} \tag{R12}$$

$$\text{Raf, Ras-GTPi} \rightarrow \text{Raf-Ras-GTPi, Raf-Ras-GTPi'} \tag{R60}$$

The reaction R12 represents binding of Raf and Ras-GTP producing Raf-Ras-GTP. The reaction R60 represents binding Raf and of the internalised Ras-GTP, i.e. Ras-GTPi, producing Raf-Ras-GTPi.

Prop.	**Absent**	**Modular verification**			**Monolithic ver.**	
	automata	**Projection**	**Result**	**Time**	**Result**	**Time**
(E2)	-	Raf, MEK, ERK	true	0.1s	true	-
(E3)	-	$proj_1$	true	492.7s	true	1129.0s
(E3)	EGFR	$proj_1$	false	45.6s	false	84.7s
(E4)	EGFR	$proj_1$	true	51.5s	true	81.5s
(E3)	GAP	$proj_1$	false	48.2s	false	85.6s
(E4)	GAP	$proj_1$	true	43.4s	true	82.7s
(E3)	Grb2	$proj_1$	false	50.6s	false	86.4s
(E4)	Grb2	$proj_1$	true	51.7s	true	82.9s
(E3)	RasGDP	$proj_1$	false	51.7s	false	176.6s
(E4)	RasGDP	$proj_1$	true	51.8s	true	161.2s
(E3)	Shc	$proj_1$	true	51.9s	true	164.9s
(E3)	Sos	$proj_1$	false	51.6s	false	90.1s
(E4)	Sos	$proj_1$	true	49.4s	true	81.7s

Table 8.1: Comparison of verification times

Species Raf, Raf-Ras-GTP and Raf-Ras-GTPi belong to the same sync-automaton. All of the other species are part of another sync-automaton.

Let us consider a state in which both of Ras-GTP and and its internalised version Ras-GTPi are available. Then the species Raf can "choose" with which of those two species to react. If it does not behave impartially, and prefers one of those two in every choice, then the other species will never be created. Subsequently, the branch of the pathway following after this species is left for starvation. Note that both Ras-GTP and Ras-GTPi are from the same component, which performs moves (reactions) and thus satisfies the fairness constraint.

Hence, not having a fairness on the level of reactions, some behaviours that should be excluded (starvation of a branch of a pathway) are instead allowed. Note that this is the cause of proving in the property (E3) reachability of disjunction of Raf* and Raf*i. Intuitively, each of them should be always reachable separately, that is both $AF Raf^*$ and $AF Raf^*i$ should hold but it is not the case.

Even though the desired stronger fairness is not too complicated to define, in order to be able to use the modular verification the whole theory would have to be revised and the property preservation theorem proved for this new case.

We remark that for the verification in NuSMV of the complete model the use of dynamic BDD reordering is essential. The order of variables is critical to control the memory and the time required by operations over BDDs. Reordering methods to determine better variable orders can be applied in order to reduce the size of the existing BDDs. The reordering is activated by the command line parameter `-dynamic`. We have used the default reordering method *sift* in all our experiments leading to the computation times and BDD sizes in Table 8.1. The verification without the reordering always crashed after hours of computation.

It should be noted that in the modelling approach in this chapter, pathways consist of reactions without inhibitors. This is motivated by the fact that for the case study we analysed, MAP kinase cascade activated by surface and internalised EGF receptors, it was not necessary as the pathway does not include inhibitors.

Also, in order to model inhibitors in a natural way, the formalism of sync-programs would need to be extended. The reason is that inhibitors need to be implemented by checking the absence of a state of a component which is not possible directly in sync-programs. The extension could be an approach combining the state checking interaction by enabling conditions of synchronisation skeletons and of the synchronisation on moves of sync-automata. This is, however, left as future work.

As dynamic sync-programs can be translated into Petri nets and sync-programs are a special case of dynamic sync-programs, one might ask what is the class of Petri nets corresponding to pathways. It is easy to see that Petri nets corresponding to pathways subject to assumptions of Section 8.1 may contain at most one token per place and for each transition T the number of incoming arcs is equal to the number of outgoing ones.

Chapter 9

Stochastic Dynamic Sync-Programs and an Application to Epidemiology

In order to be able to describe quantitative aspects of biological systems, in this chapter we sketch a possible stochastic extension of dynamic sync-programs. We do not study their modular verification, the objective is only to motivate and suggest possible future developments.

Even though dynamic sync-programs have been developed for the description of biological systems such as signalling pathways, they can well serve as an agent description language, where each individual is modelled by a finite-state automaton. To represent interactions, synchronisation is utilised. We choose to apply the extended formalism on systems of interest in epidemiology.

Mathematical modelling of the progress of infectious diseases gives the means to discover the likely outcomes of epidemics or helps manage them by vaccination. In case of large populations a deterministic approach using differential equations can be employed. More recently, individual-based methodology has been applied to study the epidemic dynamics. Although computationally quite expensive, it has an ambition to account for stochastic effects characterising such dynamics in small populations. Individual-based models, thanks to their similarity to systems of interacting agents, allow benefiting from analysis methods originally developed in computer science.

We attempt to apply such a technique, probabilistic model checking [66], to study compartmental population models. These are utilised for many common childhood diseases that confer long-lasting immunity.

We present the framework on two models. The first is a compartmental model *SIR* from the literature and where only hosts are modelled – each individual by one automaton. This model describes well the progress of infectious diseases with droplet contact route of transmission such as measles, mumps and rubella [74]. The other model, *VectSIR*, demonstrates the dynamic aspect of the description language, namely of creation of new automata in the runtime. This approach serves for investigating epidemics of diseases with vector-borne transmission. Vectors are organisms that do not cause disease themselves but that transmit infection by conveying pathogens from one host to another. Even though not supported by exact data from field studies, we believe that these models can faithfully be employed for studying tick-borne encephalitis, Chikungunya (vector mosquitoes), Pappataci fever (vector sandfly) and diseases caused by Rickettsia bacteria like rickettsialpox, Boutonneuse fever and various spotted fevers (transmitted by ticks, fleas and lice).

The analysis is done via probabilistic model checking, an extension of model checking systems that exhibit stochastic behaviour. It allows for verification of properties specified in probabilistic logic.

We are able to check properties regarding the behaviour of each population over time, as for instance to identify conditions for the outbreak of the infection or to demonstrate the retreat of the epidemic. Note that in contrast to simulation approaches with a limited number of traces we obtain exact results based on inspection of all possible behaviours of the system, giving strong formal guarantees.

9.1 Stochastic Dynamic Sync-Programs

Now we introduce a stochastic extension of dynamic sync-programs.

The standard way of extending a formalism to model quantitative aspects [53] of systems is by incorporating a collision-based stochastic framework on the lines of the one presented by Gillespie [51]. The idea is that a rate constant is associated with each considered reaction. Following the law of mass action, such a constant is obtained by multiplying the kinetic constant of the reaction by the number of possible combinations of reactants that may occur in the system. The resulting rate is then used as the parameter of an exponential distribution modelling the time spent between two occurrences of the considered reaction.

The use of exponential distributions to represent the (stochastic) time spent between two occurrences of chemical reactions allows describing the system as a Continuous Time Markov Chain (CTMC), and consequently allows verifying properties of the described system by probabilistic model checking.

In the case of stochastic sync-programs, the transition in the semantics represents a reaction and automata moves represent reactants. Hence stochastic rates need to be associated with every move. A move $s_i \xrightarrow[cc]{sc} t_i$ with rate r is denoted as $s_i \xrightarrow[cc]{sc[r]} t_i$.

In the semantics, the exact rate of a transition is equal to the product of the rates of all participating moves multiplied by the number of possible combinations of automata moves. More precisely, rate of a transition tr is $r_{tr} = \Pi_{m \in tr}(r_m * \#_m)$ where for each move m participating on the transition r_m is the rate of m and $\#_m$ is the number of automata able and ready to perform this move.

In order to make sure that the rate of a synchronised transition is meaningful, a common technique is to make one move active, which actually defines the rate for the synchronised transition, and the other moves passive, with rates 1.

9.2 Compartmental Models in Epidemiology

In order to represent the development of an epidemic a model needs to consider just the characteristic aspects that are relevant to the infection in consideration. In case of SIR model (Figure 9.1), the population is divided into three compartments: those who are susceptible (S) to the disease, those who are infected (I), and those who have recovered and are immune (R). In the diseases under consideration a single epidemic outbreak is far more rapid than the vital dynamic and we might neglect the birth/death processes.

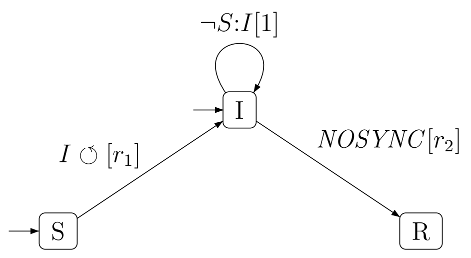
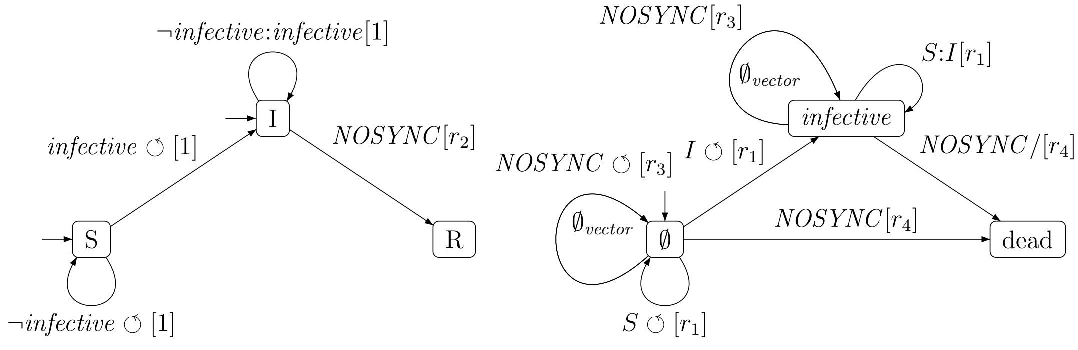


Figure 9.1: SIR model – P_{SIR}.

The typical progress of each host is S to I to R. We model this with a sync-automaton representing each individual. Atomic propositions used are three S, I and R. Each automaton has three states: $\{S, \neg I, \neg R\}$, $\{\neg S, I, \neg R\}$ and $\{\neg S, \neg I, R\}$. We display only the atomic propositions that are true in a state.

The move from S to I occurs by getting an infection from an individual that is infected. This is modelled by a synchronisation, where this move can only be synchronously executed with a move of another sync-automaton that goes from state satisfying I to state satisfying I (denoted as $I \circlearrowleft$). The rate of this move r_1 represents the probability of getting the disease in a contact between a susceptible and an infectious subject. The synchronising partner loop move on state I is passive and thus has rate 1. The recovery from the disease occurs autonomously for each individual, hence the *NOSYNC* move. Its rate is in general dependent on the recovery time D, in particular $r_2 = 1/D$. The key role in determining the dynamic of our model plays the ratio of r_1 to r_2.

In the second model we consider disease with vector-borne transmission. The hosts are modelled by using the SIR approach (Figure 9.2a), the change is that the infection occurs by a vector (Figure 9.2b). By feeding on blood of an infected host the vector gets infected (move to state *infective* with rate r_1) and transmits the infection to all successive hosts. We consider a fixed population of hosts. On the other hand, since the reproduction cycle of most vectors (mainly insects) tends to be considerably shorter, we model it by creating new individuals (*NOSYNC* loops in states $\emptyset$ and *infective* at rate r_3). Moreover, vectors, independently of their infectivity, die at rate r_4.



(a) P_{host} (b) P_{vector}

Figure 9.2: VectSIR model

9.3 Analysis via Probabilistic Model Checking

While in traditional model checking the structure to check the satisfaction of a temporal logic formula is a Kripke structure, in the probabilistic setting different models exist. Their appropriateness is mainly determined by the application, which may need assuming continuous time, nondeterminism, and so forth.

In setting of the present work, as our stochastic semantics is a Continuous Time Markov Chain (CTMC), we are interested in model checking of these models. The logic used is Probabilistic Computation Tree Logic (PCTL) [54]. PCTL is a quantitative variant of CTL where the path quantifiers A and E are replaced by a probabilistic operator P that allows querying the probability of a path formula. Another logic, Continuous Stochastic Logic (CSL) [8] extends PCTL's path operators with time bounds.

Efficient probabilistic model checking tools exist and have been applied by a large number of users from different areas. We concentrate on the model checker PRISM [59]. PRISM supports model checking of CTMCs and Markov decision processes for the logics PCTL and CSL. Other probabilistic model checkers include MRMC, LiQuor and YMER.

We performed probabilistic model checking of the two models, varying rates related to the infection process.

In order to be able to perform the analysis described in the following subsection, we perform an ad hoc translation of the stochastic sync-program to the PRISM input language. The translation preserves the CTMC semantics of stochastic sync-programs.

Note that for obtaining a model that is amenable to probabilistic model checking, that is a finite-state model, we need to restrict the number of instances of automata of each type. Thus, we set a limit of number of individuals to 10, both for hosts and vectors.

The first model, SIR, is represented by program

$$SIR = I||S||S||S||S||S||S||S||S||S$$

with one automaton in state I and nine in state S. In this model we expected different behaviour depending on the ratio $R_0 = r_1/r_2$. From the deterministic model analysis, if $R_0 > 1/S(0)$ then there is an outbreak with an increase of infectious population; if $R_0 < 1$ then no epidemic outbreak occurs, independently of the initial population in S. This conjecture is checked by fixing rate r_2 and varying r_1. We check the PCTL formula

$$P =?[((R > S))]$$

representing that the population of recovered individuals is bigger than the population of susceptibles. Since before a susceptible becomes recovered, necessarily spends some time as infected, the property means that at least half of the population was infected. In Figure 9.3, result of plotting the evaluation of the above formula with different values of r_1 it can be seen, that if r_1 is small, the probability of reaching a point, where at least half of the population was infected, is low. When increasing r_1, probability of such event increases towards 1. In figure Figure 9.4 this progress is plotted against time as the evaluation of CSL formula

$$P =?[F <= t((R > S))]$$

again varying r_1.

Probability of the retreat of the epidemic, or reaching state $(I = 0)$ is 1. In Figure 9.5 it is shown how the retreat is likely to happen in time t.

That the epidemic with small r_1 does not occur is clear from Figure 9.6a in which the probability of being in a state $I = X$ at time Y is expressed by colour intensity. With higher values of r_1 number of infected individuals is likely to increase and then due to constant population size decrease to 0. The higher r_1 is, the more rapidly the epidemic occurs (Figure 9.6b and 9.6c).

In the second model VectSIR vector-borne transmission is considered. The program *VectSIR* consists of nine susceptible hosts, one infected and five non infected vectors. The decisive rate for the speed of epidemic outbreak is the creation rate of new vectors. This is witnessed in Figure 9.7 and Figure 9.8 where r_3 is varying with r_1, r_2 and r_4 fixed.

Similarly as in the previous model, the retreat of the infection is unavoidable (probability of reaching $(I = 0)$ is 1) and Figure 9.9 details is progress over time.

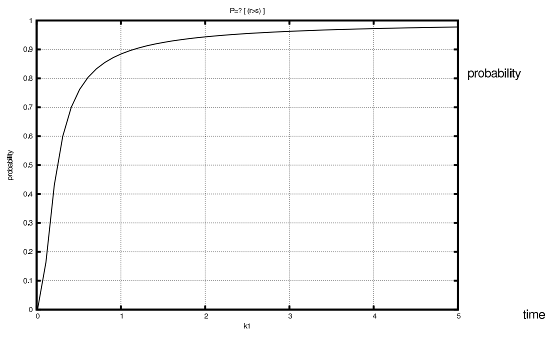


Figure 9.3: SIR: reach. ($R > S$).

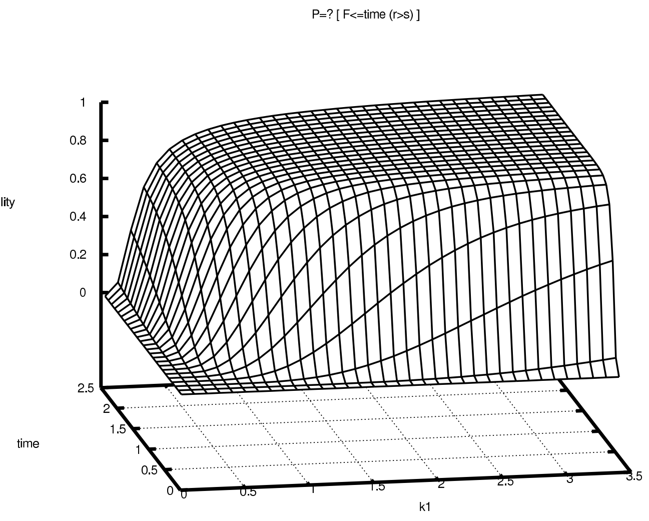


Figure 9.4: SIR: reach. ($R > S$) vs. time.

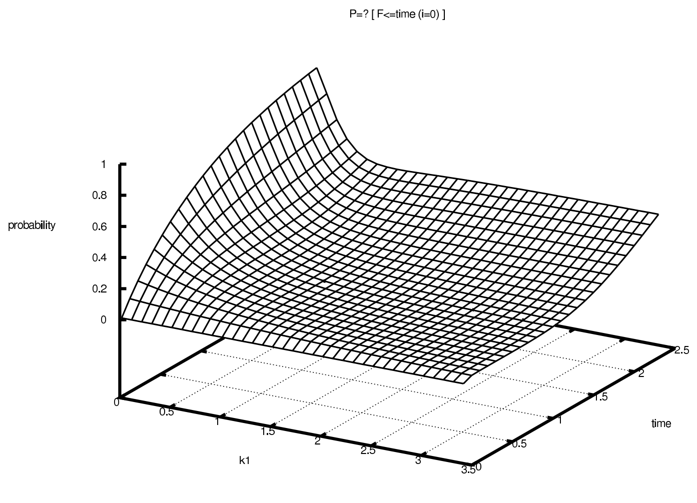
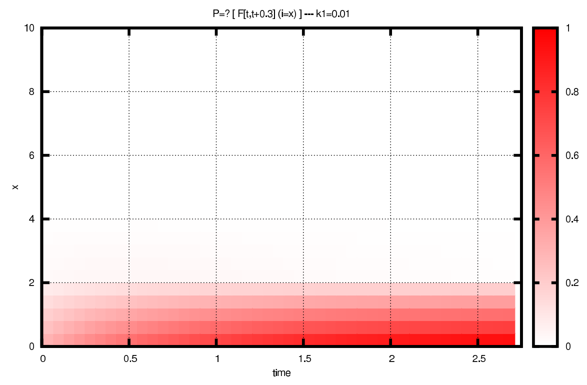
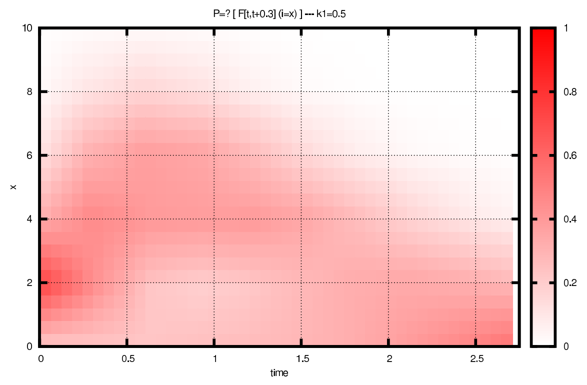


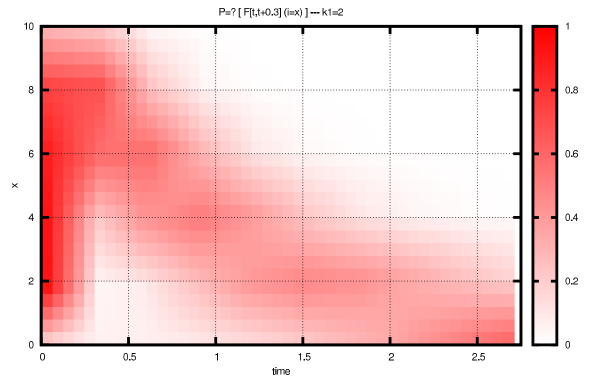
Figure 9.5: SIR: reach. ($I = 0$) vs. time.



(a) $r_3 = 0,01$



(b) $r_3 = 0,5$



(c) $r_3 = 2$

Figure 9.6: SIR: probability distribution of I vs. time

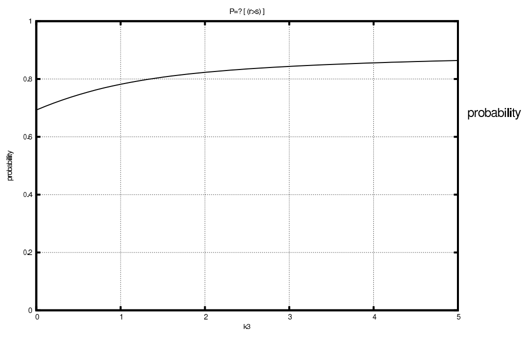


Figure 9.7: VectSIR: reach. ($R > S$).

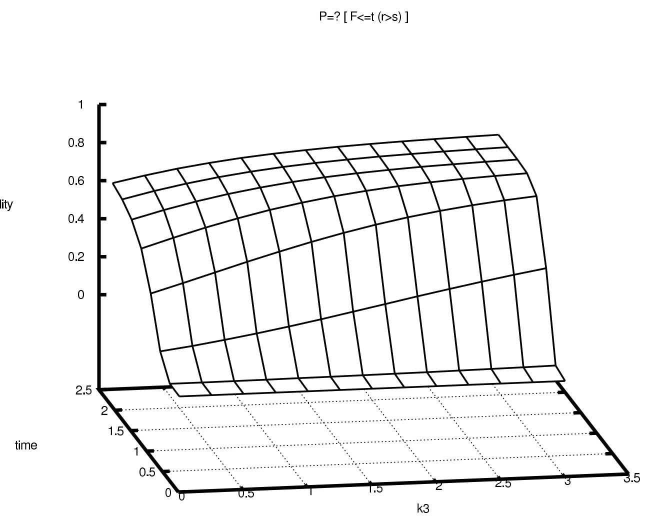


Figure 9.8: VectSIR: reach. ($R > S$) vs. time.

The probability distributions of values of I in time for r_3 equal to 0.01, 1 and 2 are shown in Figure 9.10a, 9.10b and 9.10c, respectively.

9.4 Discussion

We have used an automata based formalism to model individuals, where each agent is represented by an finite-state automaton. The formalism seems to be a suitable means for description of these systems, also because of the powerful possibility of specifying interactions of individuals as synchronisation. Another necessary aspect that allows for dynamicity is the runtime automata creation. Since arbitrarily complex behaviour of one agent is expressible by a finite-state automaton, large scale of epidemics can be modelled.

As for the analysis method, probabilistic model checking provides useful insight into the dynamics of the modelled system. Complex queries can be evaluated over the models considering probabilities of values of variables in question and, in turn, plot the results in graphs. All the plots were done by using the tool gnuplot [43] (see [44] for a tutorial).

A serious drawback, however, are the computational costs of the procedure. In particular, in order to evaluate the queries in reasonable time (in the order of hours), we needed to limit the analysis to the order of tens of individuals. A possible resolution of this impediment is to use approximate model checking that admits errors as long as they can be bound [64].

As regards related work, probabilistic model checking has recently been applied to study epidemiological models by Huang [62] who focuses the analysis to preventative and controlling measures in order to limit effects of diseases. Ciocchetta and Hilston [26] apply the formalism and toolkit Bio-PEPA to modelling and analysis of avian influenza. More often, stochastic models in epidemiology are used in connection with analysis by simulation [72].

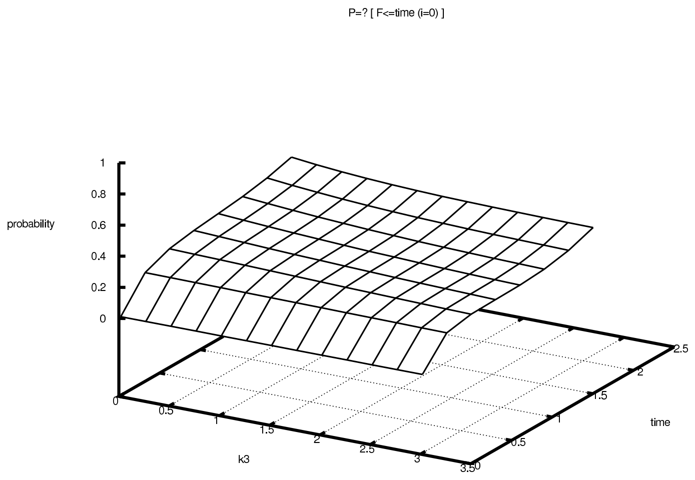
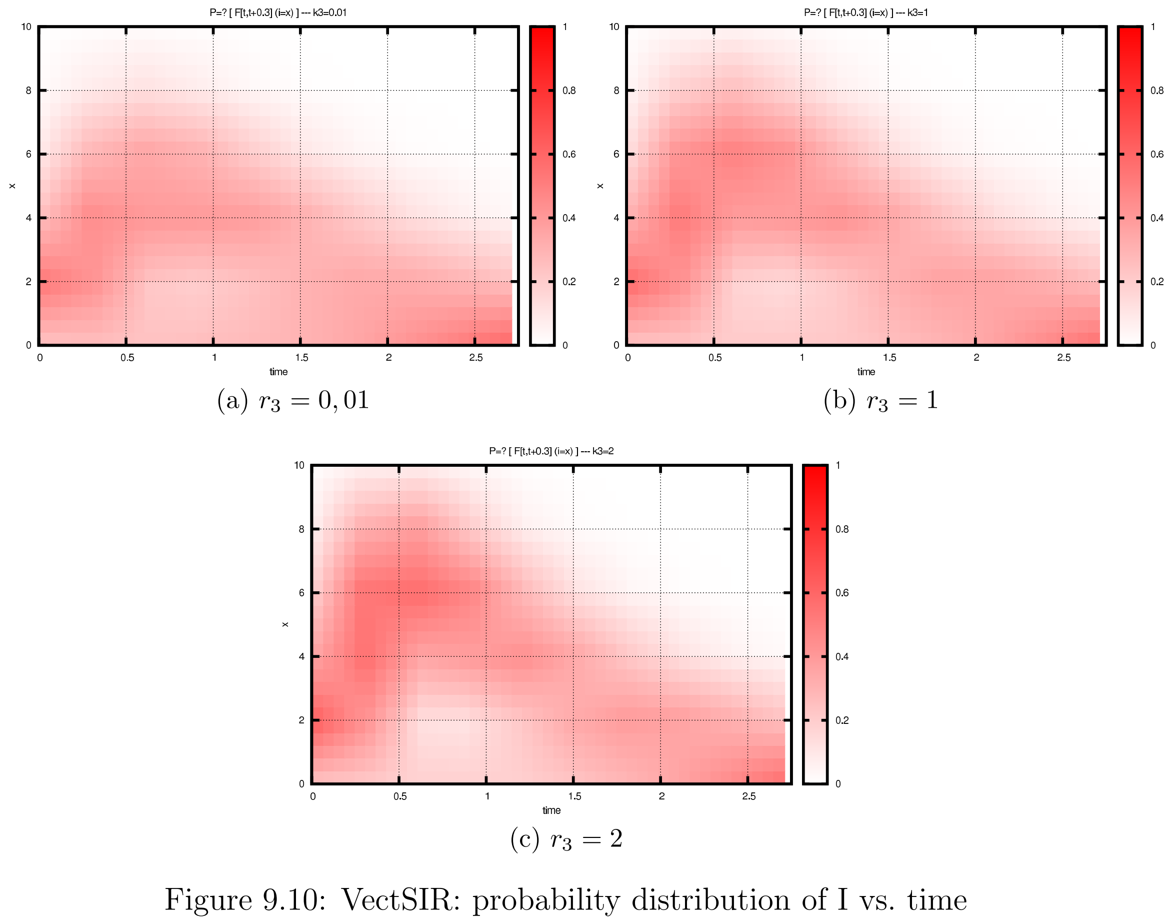


Figure 9.9: VectSIR: reach. $(I = 0)$ vs. time.



(a) $r_3 = 0,01$

(b) $r_3 = 1$

(c) $r_3 = 2$

Figure 9.10: VectSIR: probability distribution of I vs. time

Chapter 10

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. This formalism is suitable for modelling qualitatively a large class of biological systems, such as metabolic pathways, signalling pathways and gene regulatory networks. We have provided an example in the modelling of the well-known biological process of *lac* operon gene regulation. Our formalism, sync-programs, supports modular construction in that a sync-program is composed of sync-automata where each automaton models a component. A sync-program can be studied in a modular way, when some of the components are abstracted away and references to them are rendered void through an operation of projection

For this formalism we have developed a modular verification technique that allows properties expressed in the universal fragment of temporal logic CTL to be verified on suitably chosen fragments of models. In particular, these properties can express behavioural patterns that hold universally. The approach allows to preserve satisfaction of a formulae from fragments to the whole model. In practise, if one successfully verifies a property on a portion of the model obtained via projection of the original model, then the property is guaranteed to hold in the whole model. We have shown that the class of the properties that can be preserved includes several useful and biologically relevant properties, as we have illustrated on our case study.

In order to verify properties of sync-programs practically, instead of developing an instrument tailored to the formalism, we rely on the established and efficient symbolic model checker NuSMV [23]. Since NuSMV accepts as input a formalism called synchronisation skeletons that is built upon a shared-memory concurrent model, we encode sync-programs into synchronisation skeletons. We prove the correctness of the verification on the encoding, that is we show formally that whenever the tool is used for checking a property on the encoding, its result applies to the original program as well. The support of the model checker allows us to evaluate the performance of the modular verification in our case study, and we conclude that the verification of the properties on a suitable portion of the model takes much less time than verifying the same properties on the whole model.

All our experiments were performed by using NuSMV 2.5.3 on a machine running Ubuntu Linux 11.10, with Intel Core i5 processor clocked at 2.8 GHz with 8 GB of RAM. The models of the *lac* operon regulation and the MAP kinase cascade activated by surface and internalised EGF receptors, along with the tool used to translate the pathway into NuSMV can be found on the webpage of the Research Group on Modelling, Simulation and Verification of Biological Systems [36].

In sync-programs we use a general multiway type of synchronisation. In this synchronisation a move of an automaton explicitly indicates all the moves required to synchronise with it. We have investigated other modes of synchronisation in connection to the modular verification. In particular, driven by the practical motivation of obtaining more succinct models, we tried to generalise the definition of the synchronisation. However, we have discovered that in order for the modular verification technique to work, a specific type of synchronisation is required for which we have identified a necessary condition. Interestingly, this condition implies that a synchronisation that makes the modular verification correct coincides with the notion used in the original definition of sync-programs. We have also briefly investigated the possibility of application of the modular verification technique to other formalisms and we conclude that for some of the most well-known formalisms used for descriptions of concurrent systems the modular approach is applicable. Some confrontations of the synchronisation style by using preconditions and postconditions on the synchronously performed moves of other automata and the more widely used shared-channel approach is done and the equivalence proof is sketched. This suggest relations of our formalism to some other formalisms.

Since in biological systems it is often the case that the components are created and destroyed dynamically, we have extended the formalism by allowing sync-automata (the components of a sync-program) to be created dynamically by other already running sync-automata. Moreover, we allow several instances of the same sync-automata to be enabled at the same time, without any bound of the number of concurrent instances of the same sync-automaton. 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 required the modular verification approach to be adapted. We have specified a dynamic variant of the ACTL logic suitable for describing systems with multisets of agents. We have proved that the verification of DACTL properties can be limited to a portion of the model, which renders the model checking or other verification techniques more efficient. As an application, we consider the modelling and the verification the EGF signalling pathway.

In Chapter 8, have shown how to verify metabolic pathways. We showed how to automatically identify components of a pathway and implement them as sync-automata. We also give an ad-hoc translation to NuSMV that shows to be more efficient for this class of systems than the one given in Chapter 5. The results of analysis of the case study of the MAP kinase cascade activated by surface and internalised EGF receptors suggest applicability and scalability of the approach.

10.1 Discussion and Related work

In this section we provide comments and discuss the related literature of several aspects investigated in the thesis.

Logics. For the verification we use a fragment of the logic CTL, in particular universally quantified properties. Even though this class of properties can be considered as quite limited, properties expressible by ACTL^- formulae represent a significant class of properties investigated in systems biology literature as identified in [73]. Properties concerning exclusion, necessary consequence and necessary persistence and oscillatory behaviour can be considered (see Section 2.8). Also in our case studies we have shown a number of biologically relevant properties for which the presented modular verification approach can be exploited.

Most of these properties regard aspects of correct functioning of the model. It is not evident, whether emergent properties of the system can be expressed as well. Nevertheless, the possibility of verifying the correctness enables the validation of properties expressing the hypotheses one has created during the model construction. This is necessary in order to understand how the system works.

Formalism. We have defined sync-automata essentially by extending synchronisation skeletons [27] with synchronisation. Since the purpose was to develop a formalism for modelling biological skeletons such that it would allow modular verification by using a simple projection operation, we chose to stay as close to synchronisation skeletons as possible. As regards biological systems, the formalism is general enough to describe a wide class of biological systems interesting for systems biology.

The fact that the formalism is based upon finite automata, brings several important advantages: clear mathematical semantics, systems visualisation, and decidability of many crucial behavioural properties [67].

The formalism, being a form of interacting finite-state automata, bears similarities with other formalisms, such as I/O automata [68], Team automata [65], interacting automata [21] and communities of interacting automata [67].

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 relationship with synchronisation skeletons, the shared-memory model used by Attie and Emerson [7].

Finally, we have chosen the synchronisation among an arbitrary number of components, rather than a two-way synchronisation, in order to allow a more natural modelling of biological phenomena. Multi-way synchronisation is used by several other formalisms, such as [4, 65, 87].

Modular verification. 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 [52] and Dams [31] in different contexts. Other approaches to modular verification infer properties of a system from some properties of its components, e.g. [65], [75] and [14].

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 [10, 17]. 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$^-$ formulae amounts to full CTL preservation and can be obtained only under bisimilarity [31]. For application in systems biology see [77].

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 $E[g\ U\ f]$ (as done in [31] 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 [31].

Synchronisation and modular verification We have investigated the way the definition of synchronisation influences the characteristics that a subsystem obtained via projection always contains sound behaviour with respect to the whole system. A somehow related study was done in [85] for Team automata. Team automata is a formalism based on finite-state automata synchronising on shared actions. The distinctive feature of this formalism is that an automaton does not necessarily participate in every synchronisation of an action it shares (different from most other models of concurrent systems). Therefore the transition relation of a team automaton is not uniquely determined by its constituting component automata. There is freedom to choose for an action a synchronisation strategy: how automata synchronise on it. Some of the possibilities are these three:

- free – actions are never executed simultaneously by more than one automaton

- action-indispensable – actions are executed as synchronisations in which all automata having them in the alphabet must participate
- state-indispensable – actions are executed as synchronisation in which only automata which are ready participate

In [85] the computations and behaviour of team automata in relation to those of their constituting component automata has been studied in detail. Several types of team automata that satisfy compositionality could be identified, their behaviour can be described in terms of its constituting component automata. In particular compositionality is satisfied in an important case of team automata where all actions are action-indispensable, i.e. guarantee the participation of all components that share the action in sync.

Dynamicity. Dynamic sync-automata, enabling the creation of new sync-automata required the modular verification approach to be adapted with respect to the original sync-automata. Indeed, in the dynamic case the verification of a property cannot be performed only on the components of the system that are directly involved in the property, but also on those that may cause (even indirectly) the creation of some components involved in the property. Hence, we have introduced a new syntactic projection operation that takes these aspects into account. For simplicity, when abstracting away some of the component not involved in the property, we keep in the projection all the automata that may cause the creation of automata of interest. But since the internal behaviour of these extra sync-automata is irrelevant for the property, it could be abstracted from. Thus the syntactic projection could be improved (i) by removing unnecessary $NOSYNC$ moves in the components not directly involved in the property, and (ii) by removing synchronisations between components directly and non-directly involved in the property. The efficiency of these operations is to be studied.

We remark that the existence of the Petri net representation of dynamic sync-automata has several important implications.

Petri nets provide a balance between modelling power and analysability. Many important problems that are not decidable in general such as reachability, boundedness and liveness are decidable for Petri nets [47]. Model checking of branching time logics such as CTL and ACTL is, however, undecidable in general [47].

A dynamic sync-program might correspond to an unbounded Petri net, since there is no upper bound for the total number of instances, corresponding to the number of tokens.

A projection operation gives a smaller but still possibly unbounded model. The property preservation theorem is an implication, that states that from a successful verification of a DACTL$^-$ property in the projected model satisfaction of the same property in the whole model can be inferred.

The solution is to resort to standard approaches to verification of unbounded systems such as [2, 70, 46].

However, the analysis of unbounded models in the context of Systems Biology is one of major challenges for Petri net analysis techniques [56].

There are several subclasses of Petri nets that can still model interesting classes of concurrent systems, where this problem is decidable. In particular, for bounded Petri nets there are a number of tools that allow efficient CTL model checking, such as Marcie [84] for general bounded Petri nets where no previous knowledge about the boundedness degree is required and Model-checking Kit for 1-safe Petri nets [83].

As might be expected, our approach works well when an artificial bound on the number of instances is imposed. Then the structure becomes finite and is amenable to standard ACTL$^-$ model checking technique. We remark that such an assumption is quite realistic in the context of applications, since usually the instances represent molecules or other entities and it is reasonable to limit their quantity in practice.

Furthermore, other analysis techniques and tools (for a list of tools see [42]) used in the research field of Petri nets such as analyses of structural properties, invariant based analysis and various types of reachability graph based analyses might be applicable to dynamic sync-programs through the translation as well.

Fairness. 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 (see Section 2.1). The fairness constraint consists of requiring that each component of the system contributes to the overall behaviour with infinitely many transitions (Definition 3.2).

The purpose of the fairness constraints is that they guarantee that a projection of a correct behaviour of the complete system is a correct behaviour of the projected system, a property that without fairness does not hold.

We remark that 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.

This definition of fairness is sound since we chose to give the semantics of sync-programs as a labelled transition system, storing the information about the sync-automata performing the transition in the transition label. This approach is followed in sync-programs and since the semantics of synchronisation skeletons is given through LTSs as well, the two notions of fairness correspond.

In the case of implementation of the synchronisation skeletons in NuSMV (Section 5.4), unlabelled transition systems are constructed from the models, without the possibility of accessing further information. The solution is to delegate the fairness on the scheduler process selector, that schedules the asynchronous execution of individual modules. Once a module is scheduled, it must perform some move, and thus it is enough to guarantee that each module is scheduled infinitely many times in a fair path. This is ensured by using as a fairness constraint, that is a formula that has to be true along the path infinitely often, the selection of each of the modules.

We remark that the concept of fairness in connection with the reasoning about instances is a little different from the concept of fairness for sync-programs in the static case. In the dynamic setting (Section 7.5), it says that for each type i of dynamic sync-automata it is infinitely often the case that some instance of type i performs the move. Note that there is no guarantee of an infinite behaviour of all instances, which might not entirely correspond to our intuitive interpretation of the fairness. However, for specifying the fairness as a requirement of an infinite behaviour of each instance, there is not enough information in the semantics of the dynamic sync-program, since the semantics is built over multisets and no distinction of individual instances is made. Nevertheless, the present notion of fairness is sufficient for the scope of the modular verification.

As mentioned in Section 8.2, we need to ensure that each behaviour that we intuitively consider correct is present as a fair path in the semantics of the system. That is unfair paths representing intuitively correct behaviour of the system have to be extended so that they become fair. Even though this is not possible in general [3, 1] in some cases it is possible empirically add self-loops to certain states as it is done for the case of pathways automatically in Section 8.3.

The concept of unconditional fairness used in this thesis works fairly well in the applicative field of systems biology. However, in certain situations a stronger version would be desirable, as in the case of pathways. For more details see a discussion in Section 8.5.

Applicability. As regards scalability of our approach, Chapter 8 includes a model of MAP kinase cascade activated by surface and internalised EGF receptors which is made up of 143 species and 80 reactions. Checking properties on the monolithic model and in a modular fashion shows a significant difference.

The problem of decomposition or modularisation has not been treated in this thesis in a systematic way. The assumption of the approach was that it is the responsibility of the modeller to identify and model components. We are not aware of a general method, but in certain cases it is possible to do it automatically. As an example, in Section 8.2 there is an automatic procedure that identifies components in the case of pathways.

Lastly, in modular verification the choice of the submodel to be used for verification of a property must be done manually. A naive approach is to start by considering the system components that are mentioned in the property and try incrementally to add components one by one. In Section 8.4 we give an intuition about how to do it in a little more sophisticated way by using the automatically derived interaction graph of the system.

10.2 Future Work Directions

A first improvement could be extending the property preservation theorem to preserve all ACTL* properties in the way used in [52]. There should be no major technical obstacles. We have chosen to follow the ACTL$^-$ preservation to follow the approach of Attie and Emerson [7] and also because of the computational complexity advantage of CTL model checking problem with respect to the one of CTL*.

Moreover, it could be interesting to consider a weaker notion of fairness in order to be able to avoid the impossibility of extension of unfair paths to fair ones in general. An idea is to follow the line of [7].

Furthermore, investigation of a stronger fairness would be desirable for applications such as modelling pathways as suggested in Section 8.5. There is a need of a fairness on a level finer than the level of automata. In particular the fairness should avoid the problem of starvation on the level of reactions.

In addition, the experience with modelling and analysis of pathways suggests that a tool for the could be developed that allows biologists to study indispensability of components, effects of the removal (or mutation) of some components and causality relationships among components (or species). The tool would use model checking internally without exhibiting it to the user. Furthermore, the modular verification would consent to obtain results of analyses in the order of seconds or minutes even for complex pathways.

The dynamic sync-programs modular verification could be made more efficient by considering a more sophisticated projection operator as discussed in the previous section.

From the practical point of view it is still to be assessed whether a new verification tool should be developed for the formalism, or there is some existing model checker for Petri nets (e.g. [57, 83, 84]) or generally for infinite state systems that could be efficiently exploited.

From another perspective, it is interesting to understand how the proposed modular verification technique applies to Petri nets and compare the performance with verification tools for this formalism. Also, it remains an open problem whether the class of Petri nets induced by the translation from dynamic sync-programs given in Section 7.3 is proper.

As regards the quantitative approach, we have sketched a possible stochastic extension that could be useful for describing quantitative aspects of biological systems. We have not investigated the modular verification in the quantitative setting. As future work it certainly would be desirable given that the extent of state explosion is massive, as can be seen in our application to epidemiology. There is an active research field exploring this direction.

There have been first attempts to investigate the modular approach in connection with quantitative descriptions of biological systems. In [24] the authors consider a modular approach to quantitative model checking in a biological context.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. SAT-Solving the coverability problem for petri nets. *Formal Methods in System Design*, 24:25–43, January 2004.

[3] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988. 10.1007/BF01872848.

[4] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40:109–124, November 1999.

[5] Paul C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 130–145, London, UK, 1999. Springer-Verlag.

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

[7] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, 1998.

[8] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. *Computer Aided Verification*, pages 269–276, 1996.

[9] Paolo Ballarini, Radu Mardare, and Ivan Mura. Analysing biochemical oscillation through probabilistic model checking. *Electronic Notes in Theoretical Computer Science*, 229(1):3–19, 2009.

[10] 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.

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

[12] Grégory Batt. *Validation de modèles qualitatifs de réseaux de régulation génique: une méthode basée sur des techniques de vérification formelle Validation de modèles qualitatifs de réseaux de régulation génique: une méthode basée sur des techniques de vérification formelle Validation de modèles qualitatifs de réseaux de régulation génique: une méthode basée sur des techniques de vérification formelle*. PhD thesis, Université Joseph-Fourier - Grenoble, 2006.

[13] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983-09-01.

[14] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 64–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[16] Mieczyslaw Borowiecki, Izak Broere, Marietjie Frick, Peter Mihok, and Semanišin Peter. A survey of hereditary properties of graphs. *Discuss. Math. Graph*, 17:5–50, 1997.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] Nathalie Chabrier and François Fages. Symbolic model checking of biochemical networks. *Computational Methods in Systems Biology*, pages 149–162, 2003.

[23] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando 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.

[24] 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.

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

[26] Federica Ciocchetta and Jane Hillston. Bio-PEPA for epidemiological models. *Electronic Notes in Theoretical Computer Science*, 261:43–69, 2010.

[27] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, pages 52–71, 1982.

[28] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

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

[30] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

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

[32] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[33] David L. Dill. The Murphi verification system. In *In Computer Aided Verification. 8th International Conference*, pages 390–393. Springer-Verlag, 1996.

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

[35] Property pattern mappings for CTL. `http://patterns.projects.cis.ksu.edu/documentation/patterns/ctl.shtml`.

[36] Research Group on Modelling, Simulation and Verification of Biological Systems, Department of Computer Science, University of Pisa. `http://www.di.unipi.it/msvbio/`.

[37] 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.

[38] 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, 2010.

[39] Peter Drábik, Andrea Maggiolo-Schettini, and Paolo Milazzo. Modular verification of interactive systems with an application to biology. *Scientific Annals of Computer Science*, 21:39–72, 2011.

[40] Peter Drábik, Andrea Maggiolo-Schettini, and Paolo Milazzo. On conditions for modular verification in systems of synchronising components. In Marcin Szczuka, Ludwik Czaja, Andrzej Skowron, and Magdalena Kacprzak, editors, *Proceedings of Concurrency, Specification and Programming - XXth International Workshop, CS&P 2011*, Pultusk, Poland, 2011. Białystok University of Technology.

[41] 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.

[42] Petri nets tools database. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html`.

[43] Gnuplot graphing utility. `http://www.gnuplot.info`.

[44] Gnuplot tutorial. `http://gnuplot.sourceforge.net/demo_4.5/pm3d.html`.

[45] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8:275–306, June 1987.

[46] E. Allen Emerson and Kedar. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, LICS '98, pages 70–81, Washington, DC, USA, 1998. IEEE Computer Society.

[47] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets. *Petri nets newsletter*, 94:5–23, 1994.

[48] 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.

[49] Uriel Feige and Shimon Kogan. The hardness of approximating hereditary properties. Available on: `http://research.microsoft.com/research/theory/feige/homepagefiles/hereditary.pdf`, 2005.

[50] Colin Fidge. A comparative introduction to CSP, CCS and LOTOS. *Software Verification Research Centre, University of Queensland, Tech. Rep*, pages 93–24, 1994.

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

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

[53] John Haigh. Stochastic modelling for systems biology by d. j. wilkinson. *Journal Of The Royal Statistical Society Series A*, 170(1):261–261, 2007.

[54] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

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

[56] Monika Heiner and Ina Koch. Petri net based model validation in systems biology. In *In 25th International Conference on Application and Theory of Petri Nets*, pages 216–237. Springer, 2004.

[57] Monika Heiner, Martin Schwarick, and Alexej Tovchigrechko. DSSZ-MC – a tool for symbolic analysis of extended petri nets. In *Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, PETRI NETS '09, pages 323–332, Berlin, Heidelberg, 2009. Springer-Verlag.

[58] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[59] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, David Parker, Holger Hermanns, and Jens Palsberg. PRISM: A tool for automatic verification of

probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[60] C. A. R. Hoare. *Communicating Sequential Processes*, volume 9. Prentice-Hall International, London, 1985.

[61] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, September 2003.

[62] Samuel Huang. Probabilistic model checking of disease spread and prevention. Technical report, Computer Science Department, University of Maryland, 2010.

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

[64] Joost-Pieter Katoen. Abstraction of probabilistic systems. In *FORMATS*, pages 1–3, 2007.

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

[66] Marta Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 449–458. ACM Press, September 2007.

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

[68] 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.

[69] Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[70] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 250–264, London, UK, UK, 2002. Springer-Verlag.

[71] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[72] Charles J. Mode and Candace K. Sleeman. *Stochastic Processes in Epidemiology: HIV/AIDS, Other Infectious Diseases and Computers*. World Scientific Publishing Company, 2000.

[73] 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, 2008.

[74] Charles Nunn and Sonia Altizer. *Infectious Diseases in Primates: Behavior, Ecology and Evolution*. Oxford University Press, Oxford, 2006.

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

[76] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[77] 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.

[78] Amir Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45 – 60, 1981.

[79] 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.

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

[81] Christian Rohr, Wolfgang Marwan, and Monika Heiner. Snoopy—a unifying petri net framework to investigate biomolecular networks. *Bioinformatics*, 26(7):974–975, 2010.

[82] Birgit Schoeberl, Claudia Eichler-Jonsson, Ernst Dieter Gilles, and Gertraud Muller. Computational modeling of the dynamics of the map kinase cascade activated by surface and internalized egf receptors. *Nat Biotech*, 20(4):370–375, 2002.

[83] Claus Schröter, Stefan Schwoon, and Javier Esparza. The model-checking kit. *Applications and Theory of Petri Nets 2003*, pages 463–472, 2003.

[84] Martin Schwarick and Alexej Tovchigrechko. IDD-based model validation of biochemical networks. *Theoretical Computer Science*, 412(26):2884–2908, June 2011.

[85] Maurice ter Beek and Jetty Kleijn. Team Automata satisfying compositionality. *FME 2003: Formal Methods*, pages 381–400, 2003.

[86] Hantao Zhang. Sato: An efficient propositional prover. In William McCune, editor, *Automated Deduction—CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin / Heidelberg, 1997.

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