

Master's thesis

**On improving efficiency of model checking
through systematically combining Nbac
and PMC/Uppaal**

Dong Ha Nguyen

June 2004



On improving efficiency of model checking through systematically combining Nbac and PMC/Uppaal

MASTER'S THESIS

to obtain the title of
Master of Science in Technical Informatics
at Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science
Software Engineering Group

by

Dong Ha Nguyen

June 2004

Supervisor:

ir. Hylke W. van Dijk

Thesis defense committee:

Prof. dr. Arie van Deursen

ir. Hylke W. van Dijk

ir. dr. D.H.J Epema

ir. Frans Ververs

Abstract

In this work, we aim at improving efficiency of model checking by employing abstraction technique in model checkers. We introduced a systematic approach to combine a tool that provides abstraction function and a model checker. `Nbac`, `PMC` and `Uppaal` are tools involved in our implementation of the approach. Using `Nbac` as an abstraction tool to generate abstract models, we hoped that we could obtain a smaller model for which verification by `PMC/Uppaal` is feasible. We first prepared all needed components for the evaluation of our approach such as specifying input models in `Nbac` input language, developing a transformation tool to serve as a bridge between `Nbac` and `PMC/Uppaal`, and selecting case studies for experiments. Next, we did several experiments to evaluate the practical benefits of our proposed approach. The experimental results were however unsatisfactory in the sense that the resulting abstract models were unverifiable due to state space explosion. Therefore, we tried to characterize the reason of the problem to find a reasonable solution to proceed with our work.

Acknowledgements

I wish to acknowledge and thank those people who helped me in my work.

First of all, I would like to thank my supervisor, Hylke van Dijk, for his guidance and assistance in my thesis work, for constructive remarks and valuable comments to improve the contents of my reports. My special thanks go to Bertrand Jeannet, who made the tool `Nbac`. He was always patient to answer my many questions to help me get an insight of the tool.

I thank Eun Young Kang for working with me on the same subject and care for me as her student. Kang was always enthusiastic to encourage me to proceed with my work. Hans Toetenel is not directly involved in the research of this thesis. Yet, he gave me an inspiration to start my research work on model checking. I wish to thank Frans Ververs, Arie van Deursen and D.H.J Epema for being a member of my thesis defense committee.

I would also like to thank Nuffic, the Netherlands organization for international cooperation in higher education, for the award of the University Fellowship, which has financially supported me during my two years to pursue a Master degree in this beautiful country.

And I want to thank my friends in the Master course: Rui, Orlando, Yohanes and Lei for their friendship. Especially, I thank Viet for all that he has done for me.

It has been very hard to stay away from home for such a long time. I am grateful to my family for their distant support and encouragement. My family are my source of motivation to finish this work as soon as possible.

Finally, it could be impossible for me to finish this work without my beloved. Minh Hoang, I would like to thank you for helping me get through the difficult times, and for all the emotional support, and care you provided.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Summary of research assignment report	3
1.3	Objectives	4
1.4	Outline	7
2	Specifying input systems	9
2.1	Nbac input model	9
2.1.1	Variable declaration	10
2.1.2	Dataflow equation (transition part)	11
2.1.3	Assertion	11
2.1.4	Initial and final condition	12
2.2	XTG model	12
2.3	Construct an input model for Nbac	13
2.4	Summary	19
3	Connecting Nbac and PMC/Uppaal	21
3.1	Input models	21
3.2	Output models	23
3.3	Transformation framework	25

3.4	Implementation	32
3.4.1	nbac2xtgxml	32
3.4.2	nbac2uppaal	34
3.5	Summary	35
4	Case studies	37
4.1	The tiny example	38
4.2	Asynchronous Reader/Writer Algorithm	39
4.3	Fischer’s protocol	40
4.4	Subway system	42
4.5	Summary	43
5	Experimental results and problem characterization	45
5.1	Experiment procedure	46
5.2	Experimental results	47
5.2.1	The tiny example	47
5.2.2	Asynchronous reader/writer algorithm (aka Burn’s algorithm)	48
5.2.3	Fischer’s protocol and Subway system	49
5.3	Problem characterization	49
5.4	Summary	55
6	Conclusions	57
A	Nbac input systems	66
A.1	Tiny example	66
A.2	Asynchronous reader/writer algorithm	66
A.3	Fischer’s protocol	70
A.4	Subway system	72

B Nbac output systems	74
B.1 Tiny example	74
B.2 Subway system	75
C Original models	81
C.1 Fischer’s protocol in PMC specification language	81
C.2 Subway system in Lustre specification language	82
D PMC XML model	86
D.1 XML representation of a simple PMC model	86

List of Tables

4.1	System characteristics	43
4.2	System physical size	44
5.1	Time and memory usage of the verification on different abstract models of the algorithm	49
5.2	Comparison between three models	51
5.3	Simulation data of the system in Figure 5.4	53

List of Figures

1.1	A systematic approach to combine <code>Nbac</code> and <code>PMC/Uppaal</code> . . .	5
1.2	Using <code>Nbac</code> to generate a set of abstract models for <code>PMC/Uppaal</code> to verify	6
2.1	An <code>Nbac</code> input example	10
2.2	Two parallel processes in Fischer’s protocol in an <code>XTG</code> model .	14
2.3	Two parallel processes in the <code>Nbac</code> updated Fischer’s protocol	15
2.4	Using dataflow equations to model transitions	17
3.1	An example of control structure in <code>Nbac</code> output model	22
3.2	An <code>Nbac</code> control structure	23
3.3	A simple <code>XTG</code> model	23
3.4	XML representation of the <code>Uppaal</code> model created from Figure 3.3.	24
3.5	A transformed automaton	26
3.6	A dataflow equation in <code>Nbac</code> model	26
3.7	A simple system	29
3.8	Modifying the system model to specify external environment	29
3.9	The final <i>input</i> graph	31
3.10	Input and output of <code>nbac2xtgxml</code>	32
3.11	Modeling assignments in <code>Uppaal</code>	33
3.12	Modeling assignments in <code>PMC</code>	33

3.13	Input and output of <code>nbac2uppaal</code>	34
4.1	The tiny example	38
4.2	Reader and Writer graph for the algorithm	40
4.3	Fischer’s protocol	41
4.4	The model for one subway	42
5.1	Adding a counter to restrict the number of iterations	50
5.2	Memory usage of the experiment for the subway system	51
5.3	A finite system	52
5.4	An infinite system	52
5.5	An illustration of blocking due to input values	54
6.1	A systematic approach to combine <code>Nbac</code> and <code>PMC/Uppaal</code>	58

Chapter 1

Introduction

1.1 Overview

Within the last twenty years, model checking has turned out to be a useful technique for the verification of systems. Among other formal verification methods, model checking has the advantage that the verification process is performed automatically by a computer program, a model checker. A number of model checkers have been developed and actually helped to discover errors in larger and larger systems. The following four model checkers, each of which aims at a particular application domain, are used nowadays for verification by a lot of teams around the world:

- **Spin** [4] is a tool developed at Bell Labs, Murray Hill, New Jersey, USA. It was designed for simulation and verification of distributed software systems. Spin can be used as a full LTL[19] model checking system, supporting all correctness requirements expressible in linear time temporal logic.
- **SMV** [3] has been developed by the model checking group at Carnegie-Mellon University, USA. It was originally developed for the automatic verification of synchronous hardware circuits and communication protocols. Model checking of CTL[19] properties is supported.
- **Uppaal** [5] is a tool suite for modeling, simulating and verifying real-time systems. It is developed jointly by the Basic Research in Computer Science laboratory at Aalborg University in Denmark and the Department of Computer Systems at Uppsala University in Sweden.

- **Hytech** [1] is an automatic tool for the analysis of embedded systems. HyTech computes the condition under which a linear hybrid system satisfies a temporal requirement. It was developed at Cornell University and improvements were added at University of California, Berkeley.

Although model checking is successfully applied in the area of hardware and protocol verification, its use in software engineering is limited. The most significant limitation of model checking in practice is the potential state space explosion when it is applied to moderate and large size systems. Section 1 of [16] provides with a simple example to exemplify the state space explosion problem. In order to tackle this problem, abstraction techniques have been sought to avoid exhaustive state space exploration. The general idea of abstraction is that the state space of the system is reduced to a smaller one, an abstract model, so that the verification is feasible. More precisely, the model checker will verify the abstract model rather than the original one, then the result obtained from verification on the abstract model is concluded for the original model.

A practical issue arising is that how we can employ abstraction technique into model checkers. A natural direction is to put abstraction technique into the model checking algorithm. This involves defining an abstraction function (to map concrete values to abstract values) suitable for the property to be verified, constructing the abstract model and relating the verification results to the behavior of the original (concrete) model[12]. This is not an easy task especially for real-time models. Integrating abstraction technique into an existing model checker is even more difficult.

Another possibility to employ abstraction in model checkers is to make use of existing technology, i.e to use a tool that provides abstraction function to generate abstract models and then feed those models into the model checker under considering.

This thesis reports the work of a master project, which was originally part of the larger project aimed at the theoretical and practical development of a verification algorithm for real-time embedded systems based on a combination between two tools **Nbac** and **PMC**:

- **Nbac** is a model checker developed in Verimag, France to prove safety properties of synchronous programs. The highlight of **Nbac** is that it employs an abstraction technique to reduce the state space and thus scale up the size of systems that can be verified.
- **PMC** is a model checker developed in TU Delft to verify fair-TCTL properties of real-time systems. Although the application area of **PMC**

is very wide, the state space explosion problem makes PMC infeasible for most practical systems.

The scope of this master project is, however, restricted to a study on the combination approach working on non-real-time systems.

1.2 Summary of research assignment report

The first work package of our project consists of a literature study on model checking theory and algorithms applied in **Nbac** and **PMC**. This task ends with an analysis on the differences between the two approaches and a proposed solution to combine two tools. Results of this work are reported in a separated document namely *research assignment report*[16]. In this section, we give a short summary of that document.

The first three sections of the report introduce background of model checking. We first review the essential part of fixed point theory which serves as the foundation for model checking technique. Then we introduce abstract interpretation theory. This theory is a framework for the approach to simplify the analysis of a concrete system by interpreting its operation in another universe of abstract objects[13]. Next, we give definitions of several basic concepts in model checking such as Kripke structure, timed automata, and real-time model checking. We also explain a technique to reduce the state space based on equivalence relation and the notion of bisimulation.

In the fourth and fifth section of the research report, we introduce **Nbac** and **PMC** including their characteristics and underlying algorithms. The algorithm implemented in **Nbac** is founded on the theory of abstract interpretation. The main advantage of **Nbac** algorithm is computational efficiency that leads to the capability of handling complex systems. **Nbac** algorithm operates on a so-called control structure that represents the abstract model induced from the original concrete model. **Nbac** makes use of approximations to reduce the complexity of the verification process and implements an automatic adjustment on the level of approximation applied.

While **Nbac** aims at verifying synchronous programs, **PMC** aims at verifying real-time systems. The algorithm developed for **PMC** is based on partition refinement approach. In contrast to **Nbac**, computation in **PMC** is exact, and thus accuracy is an advantage of **PMC**. The major disadvantage of **PMC** is that it can only handle simple or trivial systems due to state space explosion.

The next section of the report gives a comparison between **Nbac** and **PMC**. The purpose of this comparative study is to evaluate the possibility to

achieve a more efficient model checker by combining `Nbac` and `PMC`. On the one hand, `Nbac` and `PMC` share the same idea of partition refinement in the sense that in both algorithms the starting point is a very coarse partition of the state space then the partition is analyzed and refined until termination. On the other hand, they are very different from each other in computation method and typical systems they deal with.

The study on `Nbac` and `PMC` leads to a proposed approach to combine them in a systematic way to improve efficiency of model checking for a class of models. In the research report, we express our idea of a systematic combination and leave the implementation and evaluation of our approach for the second phase, the thesis project. The report also includes an experiment on Fischer’s protocol and some remarks on model checking in practice.

1.3 Objectives

The ultimate goal of our project is to improve efficiency of model checking by systematically combining tools in a reasonable way. We utilize `Nbac` as a tool to bring abstraction capability to `PMC/Uppaal`¹ through a systematic approach. In our approach, the original system model is put into `Nbac`. Then, `Nbac` generates an abstract model based on abstract interpretation framework. The output abstract model is fed into a transformation tool to be translated to an equivalent model that can be verified by `PMC/Uppaal`. Figure 1.1 depicts this approach. By preprocessing the system model in this way, we hope that we can obtain a smaller model for which verification by `PMC/Uppaal` is feasible.

In Figure 1.1, the dashed box contains the transformation step which is taken by a tool. This tool does not only translate the syntax but also assures the semantics equivalence between an `Nbac`-like² model and a `PMC/Uppaal`-like model. We characterize a `PMC/Uppaal`-like model as an `XTG`³ model for the sake of generality. An `XTG` model is a timed automata[7] extended with the notion of urgency and the synchronous communication mechanism.

Due to the considerable size of an `Nbac`-like abstract model, the transformation process is very complicated and thus cannot be done manually. Therefore, the development of such a transformation tool is essential. Fur-

¹`Uppaal` [8] was not involved in our project in the beginning. After quite a long time working with `PMC`, we eventually found that `PMC` was not suitable in our approach for some reasons explained later on. Therefore, we have chosen `Uppaal` to replace `PMC` to proceed with our project.

²called “`Nbac` model” for short later on

³Extended Timed Graph

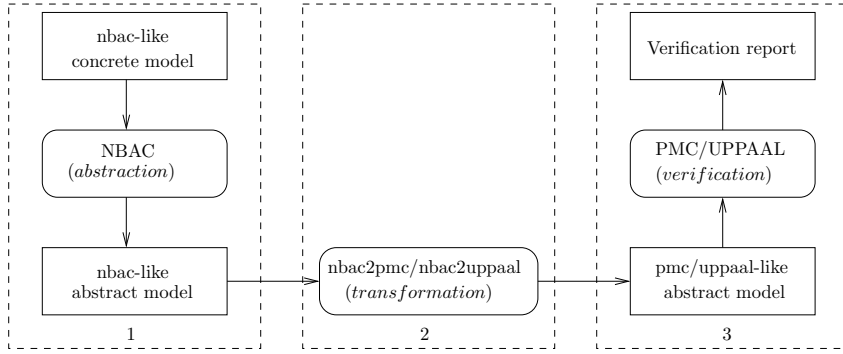


Figure 1.1: A systematic approach to combine Nbac and PMC/Uppaal

thermore, since we are not able to draw any conclusion about the outcome of the combination approach before we have the transformation technology, building a transformation tool is considered the heart of our work.

Since input systems now have to be specified in Nbac input language, another component of our work is a framework to construct an Nbac input model. This work component is meaningful because Nbac input specification language is a low-level format which is likely to confuse those who are familiar with modeling mechanism in PMC/Uppaal. Additionally, we also have to prepare a test suite in order to evaluate the outcome of our approach.

Although the procedure in our approach seems reasonable, the practical benefits of the combination are not clear. The abstract model might be small for Nbac but large for PMC/Uppaal due to differences in mechanism of those tools. Moreover, the abstraction technique is implemented particularly for Nbac but general purpose. Therefore, the main question to which we seek an answer is the following:

What are the practical benefits of combining Nbac and PMC/Uppaal using such systematic approach ?

Nbac is capable of generating several abstract models of different levels of abstraction thanks to options provided by the tool[16]. Taking into account this possibility, we hope that verification results on different abstract models can be used to reason out the relationship between efficiency and level of abstraction applied in the combination. Our idea is illustrated on Figure 1.2.

Hence, the second question to which we seek an answer is the following:

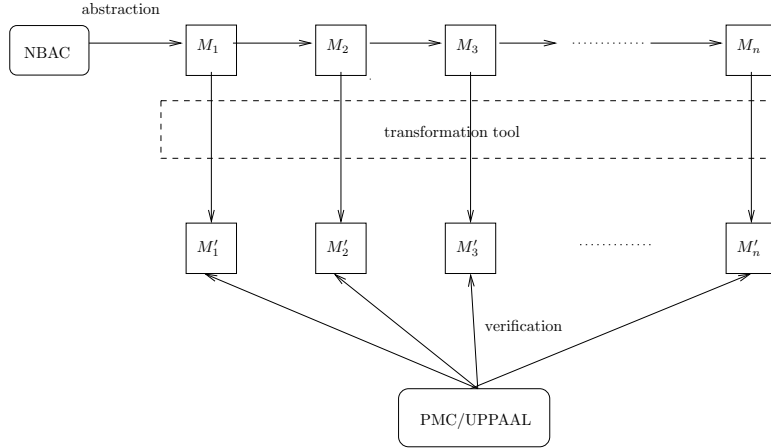


Figure 1.2: Using `Nbac` to generate a set of abstract models for `PMC/Uppaal` to verify

How can we drive `Nbac` to generate the right abstract model to achieve the best efficiency in terms of accuracy of verification results and the scalability of systems that can be verified ?

In conclusion, our project consists of three work components:

1. *Defining a framework to construct an input system in `Nbac` input language.* We focus on the construction of an input model from an XTG model. This work component is represented by the left dashed box on Figure 1.1.
2. *Implementing the transformation tool to build a connection between `Nbac` and `PMC/Uppaal`.* We study the `Nbac` output model and `PMC/Uppaal` input model to define a transformation framework that assures the equivalence between two models. Then we implement the tool based on the defined framework. This work component is represented by the middle dashed box on Figure 1.1.
3. *Doing experiments with case studies and analyzing the results.* We select four case studies to do the experiments. In reality, we could not obtain the expected results so we then tried to characterize the problems to find out how we can improve the situation. This work component is represented by the right dashed box on Figure 1.1.

1.4 Outline

This thesis reports our work to combine **Nbac** and **PMC/Uppaal** through a systematic approach in order to improve efficiency of model checking. The organization of the thesis is as follows:

- Chapter 2: We explain the procedure to specify an input system in **Nbac** input language.
- Chapter 3: We describe a framework to transform an **Nbac** model to an **XTG** model and then the implementation of the transformation tool based on that framework.
- Chapter 4: We describe four systems chosen as case studies to evaluate our approach.
- Chapter 5: We present the experimental results on the four case studies. Unfortunately, it turned out that our proposed approach did not work as we expected. We identify the problems and propose techniques to solve these problems.
- Chapter 6: The concluding chapter contains the summary and overall conclusions. We answer to the questions concerning our technique and we state recommendations for future work.

Chapter 2

Specifying input systems

As previously described in Section 1.3, the input system has to be specified in `Nbac` input language. In this chapter, we introduce three different ways to obtain an `Nbac` input model. Since people may need to try the new combination approach on systems that are initially modeled in `PMC/Uppaal`, we define a framework to transform an `PMC/Uppaal` model to an `Nbac` input model. Such a framework is necessary because `Nbac` input model is specified in a low-level format which mainly involves variables and dataflow equations on variables while `PMC/Uppaal` model is specified in a higher level format that involves locations and transitions of an automaton.

We first present components of an `Nbac` input model in Section 2.1. Then we give a short introduction of `XTG` structure in Section 2.2. Next, we list three methods to obtain an input model and describe in details the construction of an `Nbac` model from a `PMC/Uppaal` model in Section 2.3. The procedure is illustrated by the construction of the Fischer’s protocol model.

2.1 `Nbac` input model

`Nbac` input model is specified in a simple low-level format including dataflow equations on Boolean and Integer variables since `Nbac` was originally made to analyse programs written in `Lustre` [2] dataflow synchronous language.

Let M be an `Nbac` input model. Let Var_decl be the *variable declaration* part of M . Let $Trans$ the set of *dataflow equations* of M . Let $Assert$, $Init$, $Final$ be Boolean expressions indicating the *assertion*, *initial* condition and *final* condition of M , respectively. Using these definitions, M is described

as:

$$M = Var_decl; Trans; Assert; Init; Final$$

We will describe these parts in the next subsections and use the simple example in Figure 2.1 to exemplify our description. The reference manual of the input format can be found at [21].

```
state
  init,ok: bool;
  b0,b1:bool;
  x,y:int;

input
  p1,p2:bool;

transition
  init' = false;
  ok' = if init then true else ok and x>=y;
  b0' = if init then false else not b1;
  b1' = if init then false else b0;
  x' = if init then 0 else if (b0 = b1) then x+1 else x;
  y' = if init then 0 else if (b0 <> b1) then y+1 else y;

assertion
  p1 and not p2 or not p1 and p2;

initial init;

final not init and not ok;
```

Figure 2.1: An Nbac input example

2.1.1 Variable declaration

The variable declaration part *Var_decl* includes the declaration of Boolean state variables $(b_i)_{i=1\dots m}$ and Boolean input variables $(c_j)_{j=1\dots n}$ and the declaration of Integer state variables $(x_k)_{k=1\dots p}$ and Integer input variables $(y_l)_{l=1\dots q}$, where:

1. **State variable:** At each execution step, a *state* variable takes new value computed from the current value of the set of all *state* variables and the value of the set of all *input* variables.
2. **Input variable:** *Input* variables model the external environment. Their values are decided in a non-deterministic way. If their values are not constrained by an *assertion* of the program, then they are anything

in their domain ($\mathbb{B} = \{T, F\}$ for Booleans and \mathbb{N} for Integers). If an *assertion* constrains their values, they may take non-deterministically any values satisfying the constraint.

In the example, `init,ok,x,y,b0,b1` are *state* variables; `p1,p2` are *input* variables.

2.1.2 Dataflow equation (transition part)

The *transition* part *Trans* defines a set of parallel equations $b'_i = \phi_i(\vec{b}, \vec{c}, \vec{x}, \vec{y})$, $i = 1..m$ and $x'_k = \psi_k(\vec{b}, \vec{c}, \vec{x}, \vec{y})$, $k = 1..p$ giving the value of each *state* variable at the next execution step, as a function of the current values of state and input variables. In these equations, expressions ϕ_i and ψ_k are well-formed expressions of suitable types, possibly mixing Boolean expressions and linear numerical expressions. Atoms of Boolean expressions can be either Boolean variables or linear constraints. A conditional `if-then-else` statement can be used in a dataflow equation.

In the example, `init` is set to `true` once at the initial state of the system (described in the *initial* part). In the *transition* part, `init` is set to `false` for all subsequent executions because it specifies whether or not the system is in the initial state. The Boolean variable `ok` models the status in which the system still satisfies the condition $x \geq y$. Variables `b0,b1,x,y` model the behavior of the *M*.

2.1.3 Assertion

An *assertion* *a* allows to constrain the values of input variables depending on the current state. It is defined by a Boolean expression *Assert* on variables, which give for each state the possible values for input variables:

$$a = \text{Assert}(\vec{b}, \vec{c}, \vec{x}, \vec{y})$$

In the example, there is an *assertion* to constrain the values of two input variables `p1` and `p2`:

$$\text{p1 and not p2 or not p1 and p2;}$$

This assertion means that `p1,p2` can be either `true` or `false` but they never take the same value.

2.1.4 Initial and final condition

The control structure underlying an Nbac model consists of three kinds of states: *initial* states, *good* states and *final* states (bad states). The *initial* and *final* condition are two Boolean functions $Init(\vec{b}, \vec{x})$ and $Final(\vec{b}, \vec{x})$ respectively defining the set of initial states and final states (i.e. the condition in which the safety property is violated) of M .

In the example, the *initial* condition is “`init`”, meaning that the system starts with `init = true`. Actually, the Boolean variable `init` is added to the example to assure that there is only one initial location in the program. The final condition is “`not init and not ok`”. If we look into the *transition* part, the assignment of variable `ok` is:

$$ok' = \text{if } \text{init} \text{ then } \text{true} \text{ else } ok \text{ and } x >= y,$$

which corresponds to the safety property of this system of $x \geq y$.

2.2 XTG model

Input models for PMC and Uppaal are characterized as XTG structure. In this section, we introduce this structure. An XTG structure is used to represent models of real-time parallel systems. It is a finite state machine augmented with clocks and data. Clocks are non-negative real-valued variables that increase at the same fixed rate. The increasing of clocks models the progress of time. An important feature of an XTG is that it may consist of several parallel processes.

The basic representation for an XTG is in the form of a timed automaton. A timed automaton is formally defined as a tuple $(L, L^0, \Sigma, X, I, E)$, where

- L is a finite set of locations,
- $l^0 \in L$ is the initial location,
- Σ is a finite set of labels,
- X is a finite set of clocks,
- I is a mapping that labels each location $l \in L$ with some clock constraints in $\phi(X)$ and
- $E \subseteq L \times \Sigma \times \phi(X) \times 2^X \times L$ is a set of edges.

Since `Nbac` does not support real-time notion, features of an `XTG` such as clocks and clock constraints will be treated the same as Integers. More details of a timed automaton including definition and example can be found in the research report [16] or [6].

2.3 Construct an input model for `Nbac`

Generally, there are three ways to obtain an `Nbac` input model:

1. **Specify the model directly in `Nbac` input language:** Using this method, we have to imagine how variables are changed when the system works. The system is specified in a low level format, meaning that we work on variables and dataflow equations only. The advantage of this option is that the semantics of the resulting model is straightforward for the modeling person. She/He knows for sure about why a variable is used, what it means, and why a dataflow equation is set in such way. However, it is extremely difficult to create a model for a non-trivial system this way.
2. **Transform from a Lustre program:** We create a model in the synchronous language Lustre[2]. Then we transform the model to `Nbac` using a transformation tool `lus2nbac`, which is provided by the author of `Nbac`. The disadvantage of this option is that it is difficult to comprehend the meaning of the transformed `Nbac` model since variables are generated by a tool. See the original model and the `Nbac` input model of the Subway system in Appendix C.2 and A.4 respectively for a reference.
3. **Transform from a “high-level” model such as `XTG` models:** The term “high-level” is used to emphasize that the model consists of components of an automaton such as locations and transitions in addition to primitive elements like variables and assignments. The semantics of an `XTG` model is described in Section 3.2. We first create an `XTG` model by specifying locations and invariants, transitions and guards of the automaton representing behaviors of the system being verified. Next, we transform that `XTG` model into `Nbac` input language. Although principles to transform a model from `XTG` syntax to `Nbac` syntax are well-established, the manual transformation procedure is very tedious and error-prone.

In this section, we will explain in details the third method since it is likely that verification engineers are familiar with the general modeling mechanism

like in PMC/Uppaal. Furthermore, several systems may be already modeled in PMC/Uppaal or similar input languages. In this case, we only have to do the transformation step.

We choose the Fischer’s protocol as a “case study” to explain the construction procedure. Figure 2.2 presents the graphical XTG model of the Fischer’s protocol. The protocol is explained in details in Chapter 4. Here we describe how to specify such a system in Nbac input language.

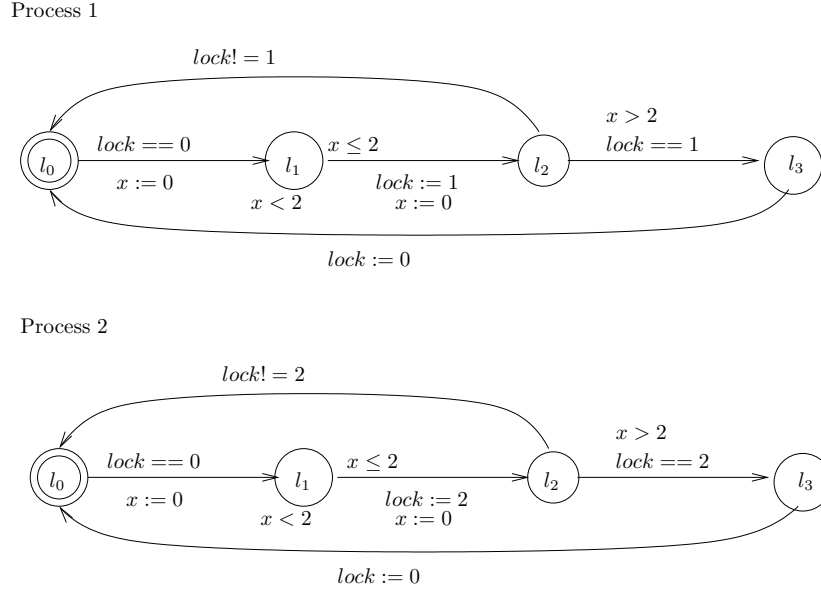


Figure 2.2: Two parallel processes in Fischer’s protocol in an XTG model

The general recipe to construct an Nbac input model from an XTG model consists of the following items:

- **Variable declaration:** specifying how variables are declared in an Nbac model.
- **Parallel processes:** creating a global graph from all parallel processes.
- **Time:** modeling real-time by physical clock tick in Nbac.
- **Location and Transition:** transforming the automaton indicating the behavior of the model.
- **Property specification:** specifying the *final* condition from the property specified in an XTG model.

Variable declaration. All variables, global and local variables, of an XTG model are declared in `Nbac` as global *state variables*. Since `Nbac` has a single namespace, it is possible that there are coincidences on local variable names in an XTG model. In these cases, names of variables must be changed.

Integer and Boolean variables in an XTG model can be declared the same way in `Nbac` model. Clock variables, however, are not supported in `Nbac`. We declare them as Integers. Our method to model their real-time notion is described hereafter.

In Figure 2.2, `lock` is a shared integer variable between two processes; `x` is a local clock variable to represent the clock in each process. In the `Nbac` model, `lock` is modeled by two Boolean variables `lock1` and `lock2`. The reason for such adaption is that Boolean variables is handled exactly while Integer variables are approximate in `Nbac`. Integer variables are approximate in `Nbac` because they are abstracted by convex polyhedra. Moreover, in Fischer’s protocol, `lock` is used to control the status of the critical section ($lock \in \{0, 1\}$). Thus, it is natural to model `lock` like that. Two integer variables `c1`, `c2` are introduced to model the local clock variable `x`.

Figure 2.3 shows an updated version of the Fischer’s protocol when it is specified in `Nbac`.

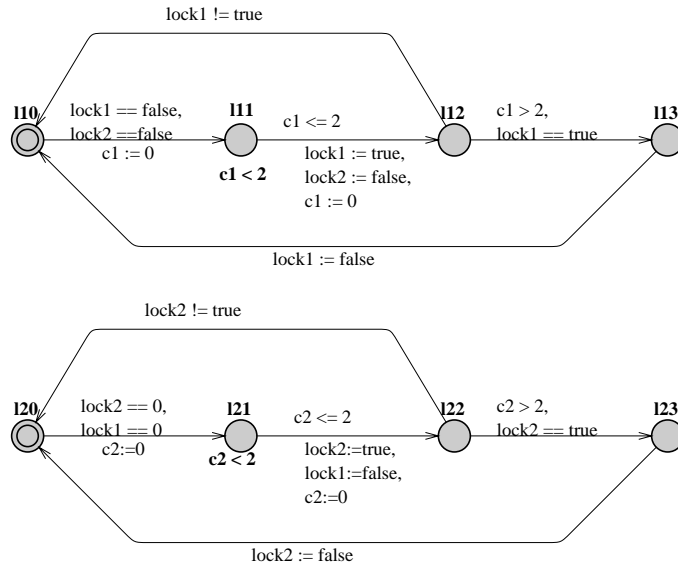


Figure 2.3: Two parallel processes in the `Nbac` updated Fischer’s protocol

Parallel processes. In XTG, we can specify several processes in a system independently and then simply declare them as parallel processes. In

Nbac, we have to create the global graph from all parallel processes. That means we will have to specify explicitly how those processes interact. Fortunately, the execution order of processes in the system can be left unspecified. The sequence of running these parallel processes can be modeled in a non-deterministic way in which *input* variables and *assertion* are used to coordinate the parallel processes.

There are two parallel processes in Fischer's protocol shown in Figure 2.2. We introduce two Boolean input variables **p1**, **p2**. At every execution step, the system will consult the *input* variables to decide which process should take action. If **p1** is **true**, it is the turn of the first process to execute. Similarly, if **p2** is **true**, it is the turn of the second process to execute. Obviously, **p1** and **p2** cannot be **true** or **false** at the same time. Therefore, the assertion to constrain these two variables will be:

$$\text{p1 and not p2 or not p1 and p2}$$

Time. Since **Nbac** does not support real-time, we model a variable of clock type x_i of an **XTG** model in **Nbac** using a Boolean *input* variable namely **tick** to model *time action* and an Integer *state* variable c_i to represent clock variable value:

$$\text{Map} : x_i \longrightarrow (\text{tick}, c_i)$$

where $c_i = x_i$.

In the Fischer's protocol example, **tick** is an input variable to model the time action. Hence, the external environment setting involves three Boolean *input* variables **p1**, **p2**, **tick**. The assertion is modified as follows:

```
(if (l11 and not c1<2) or (l21 and not c2<2) then
not tick else true) and not (p1 and p2)
and (if tick then not p1 and not p2 else
if p1 then not tick and not p2 else
if p2 then not tick and not p1 else true);
```

The first part of the above assertion specifies that when the system is in location **l1** of a process p_i and the corresponding clock variable $c_i \geq 2$, a time action (**tick**) must not be taken, i.e. the system cannot stay in the current location any more. The rest part of the assertion specifies that only one variable can be true at any time.

Location and transition. Since **Nbac** input language does not support specification of locations and transitions we have to model those elements using standard variables and dataflow equations supported by **Nbac**.

Note that although it is possible to explicitly specify a control structure (automaton) in an **Nbac** input model, we cannot map the automaton in an **XTG** model to a control structure in an **Nbac** model directly one-to-one. The reason is that a dataflow equation $b'_i = \phi_i(\vec{b}, \vec{c}, \vec{x}, \vec{y})$ or $x'_k = \psi_k(\vec{b}, \vec{c}, \vec{x}, \vec{y})$ must be specified for every state variable in such a way that the equation expression remains the same for any edge (transition) in the resulting **Nbac** model, while it is not the case in an **XTG** model.

We use Boolean variables to specify *locations* in **Nbac**. Every location loc_i will be represented by a Boolean *state* variable l_i . As such, if the system is in loc_j , $l_j = true \wedge l_{i,i \neq j} = false$. Boolean variables are used to model locations for the reason that Booleans are handled exactly while Integers are abstracted in **Nbac**.

In our example, $lij \in \{110, 111, 112, 113, 120, 121, 122, 123\}$ is a Boolean variable representing a location loc_j of process p_i . This naming rule is just our convention.

Transitions are modeled implicitly through dataflow equations in **Nbac**. The specification of transitions is best explained through a few cases in the Fischer's protocol presented in Figure 2.4.

```

110' = if p1
      then if 112 then not lock1
           else 113
      else 110;
lock1' = if p1 then
         if 111 then true
         else if 113 then false
         else lock1
      else if p2 then
         if 121 or 123 then false
         else lock1
      else lock1;
c1' = if tick then c1+1
     else if p1 and (110 or 111) then 0
     else c1;

```

Figure 2.4: Using dataflow equations to model transitions

Consider the equation on 110. The purpose of the equation is to give the value for 110 at the next execution step. According to the external environment setting specified by the assertion on input variables **p1**, **p2**, **tick**, there are three possibilities: process 1 takes action (**p1**), process 2 takes action (**p2**), or time action (**tick**). In the two later cases, the value of 110 does

not change because in those cases process 1 will stay in the same location. In case process 1 takes a transition, then we will consider four sub-cases according to the location where process 1 currently stays: `l10,l11,l12,l13`. Among these cases, the two cases in which process 1 may go to location `l10` and thus `l10` is set to `true` are: process 1 is currently in location `l12`, `lock1` is `false`, and process 1 is currently in location `3`. For other sub-cases, `l10` does not change.

Next, consider `lock1`. Again, the value of `lock1` is set according to the external environment setting variables. There are three cases where `lock1` is reset: process 1 takes the transition from location `l13` to location `l10`, process 1 takes the transition from location `l11` to location `l12` and process 2 resets the lock.

Consider the last case, Integer variable `c1`. If the time action is taken (`tick`), `c1` is incremented by 1. Otherwise, `c1` is reset if there is a transition of process 1 from location `l10` to location `l11` or from location `l11` to location `l12`.

Property specification. Nbac only supports the verification of safety property in terms of reachability. In an XTG model, a safety property ϕ is specified as one of the following two forms:

$\phi = \text{AG not bad-condition}$

or

$\phi = \text{EF bad-condition}$

The semantics of `AG`, `EF` can be found in [16]. The first one is interpreted as “The system is never in `bad-condition`”. The second one is interpreted as “It is possible for the system to be in `bad-condition`”. In Nbac ϕ is specified using the **final** condition as follows:

`final bad-condition`

In our example, ϕ is specified in the XTG model as `AG not (P1.l3 and P2.l3)`, which is specified in the Nbac model as:

`final l13 and l23`

The complete Nbac input model of the Fischer’s protocol can be found in Appendix A.3.

Since we have to specify locations and transitions explicitly at low level and create a global execution of the system from all parallel processes, this

transformation procedure indeed takes over one part of the verification that should be done automatically.

The risk of missing cases increases when the system grows in the sense that it contains more locations, transitions, and especially parallel processes. A tool to take over this complicated procedure would help to avoid the error-prone manual transformation. However, we did not pursue the development of such a tool.

2.4 Summary

In this chapter we have introduced the specification language to model an input system for `Nbac`. `Nbac` input format is a low-level formal that specifies dataflow equations on Boolean and integer variables.

There are three ways to obtain an `Nbac` input model: create the model directly in `Nbac` input language, transform manually from an `XTG` model, and transform automatically from a Lustre program. We explained the framework to construct an `Nbac` input model from an `XTG` model using the Fischer's protocol example as an illustration. Although the model of Fischer's protocol is simple, the manual construction procedure still requires a large amount of work.

We found that the aforementioned manual procedure to construct an `Nbac` input model is an error-prone procedure. For example, in a system with a significant number of parallel processes, it is very likely that we will overlook a few cases. A tool to take over this part which has not yet been developed at the moment is indispensable.

In the next chapter, we move to an important part of our project: develop a tool serving as a bridge to connect `Nbac` and `PMC/Uppaal`.

Chapter 3

Connecting Nbac and PMC/Uppaal

The main goal of this thesis work is to combine `Nbac` and `PMC/Uppaal` in such a way so that `PMC/Uppaal` can “use” `Nbac` abstraction function. In our proposed approach, we use `Nbac` to generate abstract models and then use `PMC/Uppaal` to verify those abstract models. An abstract model generated by `Nbac` cannot be directly verified by `PMC/Uppaal` due to differences between an `Nbac` model and `XTG` model. Therefore, we have to provide a transformation utility to automatically construct an `XTG` model from an `Nbac` output model. That kind of transformation tool is a must for the reason that an abstract model generated by `Nbac` is usually very complicated.

In this chapter, we describe our design and implementation of such a transformation tool. It can be considered the bridge to connect `Nbac` and `PMC/Uppaal` in our approach. We first introduce the characteristics of input models (`Nbac` output models) and output models (`XTG` input models) for the transformation tool in Sections 3.1 and 3.2. Next, we present the transformation framework underlying the tool in Section 3.3. It is not our goal to describe in details the implementation work. Some general information about the implementation of the tool is given in Section 3.4.

3.1 Input models

The input of the transformation tool is an output model generated by `Nbac`. This model is specified in the same format with an `Nbac` input model. An `Nbac` output model can also be specified in `DOT` format¹. However, we did

¹see www.graphviz.com for details

not use that format for the reason that it is used by `Nbac` for visualization purpose only, meaning that model details are necessarily reduced to serve that purpose.

An `Nbac` output model contains all the components that have been described in Section 2.1 including variable declaration, dataflow equation, assertion, initial and final condition. Therefore, we will not repeat those parts here.

The major difference between an `Nbac` input model and output model is the presence of an explicit *automaton* or *control structure* in the output model. This control structure is the foundation for our transformation between models. It is defined by a set of locations and edges:

- Locations: A location is associated with an *invariant*, which is a Boolean expression that gives the set of states represented by the location.
- Edges: An edge is associated with with a *guard*, which is a Boolean expression on *state* and *input* variables that indicates under which condition the edge can be taken.

The union of all invariants of the locations indicates the considered state-space. The union of all guards of the edges allows to enforce the behavior of the global system.

Figure 3.1 presents an example of the control structure specified in an `Nbac` output model.

```

automaton
  location bad_0 : (not ok and not init0) and (y+x-2>=0 and -y+x+2>=0
                 and y-x>=0);
  location init_0 : (init0) and true;
  location good_0 : (ok and not init0) and (y>=0 and x>=0 and -y+x+1>=0);
  edge (init_0,good_0) : true and true and ((not p2 and p1)
                                           or (p2 and not p1));
  edge (good_0,bad_0) : true and (y-x-1>=0) and ((not p2 and p1)
                                                  or (p2 and not p1));
  edge (good_0,good_0) : true and (not y-x-1>=0) and ((not p2 and p1)
                                                       or (p2 and not p1));

```

Figure 3.1: An example of control structure in `Nbac` output model

The control structure in Figure 3.1 contains three locations and three edges. We can see that there is no assignment in this automaton besides locations, edges, invariants and guards. Assignments are put in the *transition*

part (dataflow equations), meaning that for every transition to be taken, variables will be updated according to those dataflow equations.

Figure 3.2 depicts a shape of control structures in `Nbac` output models. The semantics of the `Nbac` automaton in Figure 3.2 is somewhat different from automata in `XTG` models. An `Nbac` control structure represents the status (good or bad) of the system rather than explicitly specify the behaviors of the systems as in `XTG`. As illustrated in Figure 3.2, the system is supposed to loop around the `good` location. If something bad happens the system will move to the `bad` location. To prove a safety property, `Nbac` will try to prove that there is no link between the good and bad locations. In an `Nbac` output model, the safety condition is mixed with the system model, i.e. it is specified in the guard of the transition (`good, bad`).

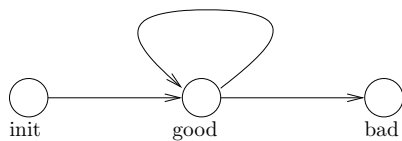


Figure 3.2: An `Nbac` control structure

3.2 Output models

The output of the transformation tool is an input model `XTG` for `PMC/Uppaal`. In general, an `XTG` is a type of Kripke structure, which is defined in the research report[16]. It is used to specify the behavior of the systems. It allows the specification of locations, invariants, edges, guards and assignments associated with edges.

Figure 3.3 shows a graphical representation of an `XTG` input model. In

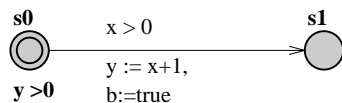


Figure 3.3: A simple `XTG` model

Figure 3.3, `s0,s1` are two locations; `y>0` is the invariant of location `s0`; `x>0` is the guard of the transition from `s0` to `s1`; `y:=x+1,b:=true` are two assignments associated with that transition.

The output of the transformation tool is specified in `XML` format. `Uppaal` models are saved directly in `XML` format while `PMC` models are saved in its

own specification language. However, an XML schema for PMC model was already defined and a tool to translate the model in XML format to PMC input language is available.

Figure 3.4 presents the Uppaal XML representation of the model given in Figure 3.3.

```

<nta>
  <declaration>int x,y;bool b;</declaration>
  <template>
    <name>P</name>
    <location id="id0">
      <name>s0</name>
      <label kind="invariant">y &gt;0</label>
    </location>
    <location id="id1"><name>s1</name></location>
    <init ref="id0"/>
    <transition>
      <source ref="id0"/><target ref="id1"/>
      <label kind="guard">x &gt; 0</label>
      <label kind="assignment">y := x+1,b:=true</label>
    </transition>
  </template>
  <system>system P;</system>
</nta>

```

Figure 3.4: XML representation of the Uppaal model created from Figure 3.3.

Consider Figure 3.4. Variable declarations are specified in tag `declaration`. Each automaton in Uppaal is called a template and thus saved in tag `template`. Locations and transitions are specified together with invariants, guards, assignments in appropriate tags. Note that invariants, guards and assignments are stored as strings.

The XML representation of PMC model is similar to Uppaal. Components such as variable declarations, Boolean expressions and assignments are specified in a different way. For instance, in Uppaal, to model the assignment $y:=x+1$, we just put the whole string in the label associated with the transition. On the contrary, in PMC, that assignment is split into atomic elements as follows:

```

<ValueAssignment>
  <VariableName>y</VariableName>
  <Value>
    <BinaryPlus>
      <LeftValue>
        <Value type="Name">x</Value>

```

```

        </LeftValue>
        <RightValue>
            <Value type="Integer">1</Value>
        </RightValue>
    </BinaryPlus>
</Value>
</ValueAssignment>

```

As a consequence, the PMC XML model is much bigger than an Uppaal model provided that they model the same system. Moreover, more work is needed when we implement the transformation tool to generate a PMC XML model. Since the PMC XML document of the model in Figure 3.3 is too large, we put it in Appendix D.1.

Another minor difference between PMC and Uppaal model is that the property in Uppaal is saved in a separate file while in PMC, it is specified in the same file of the system graph.

3.3 Transformation framework

In previous sections we have described the input and output models for the transformation tool. In this section we define a framework for the transformation from an input model to an output model.

Locations and transitions. Basically, the transformation is performed based on two automaton structures one in Nbac and one in PMC/Uppaal. Locations and transitions (edges) as well as their associated invariants and guards in an Nbac control structure will be locations and transitions in an XTG model.

Figure 3.5 presents the automaton of the XTG model transformed from Nbac model specified in Figure 3.1. The safety property is “not init0 and not ok”.

Assignments in transitions. In Nbac assignments are put in the *transition* part. That means for every execution step, the new values of variables are set by those assignments. Assignments remain the same for all transitions, only the values of variables change. In an XTG model, assignments are attached to transitions so we have to explicitly put all assignments defined in the *transition* part of an Nbac model to all transitions in the XTG model.

To bring assignments in an Nbac model to an XTG model, we have to solve the following two issues: the existence of **if-then-else** statement and parallel assignments in Nbac dataflow equations.

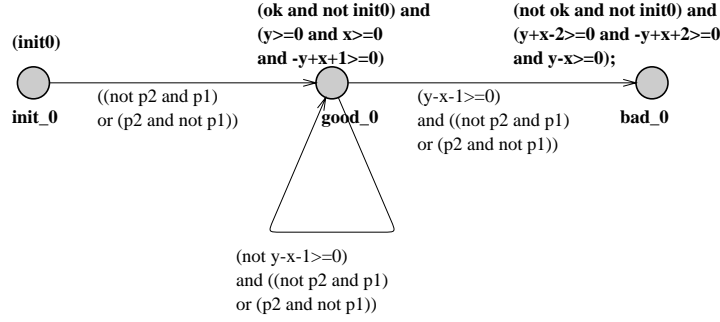


Figure 3.5: A transformed automaton

1. **The existence of if-then-else statement in Nbac dataflow equations.** A dataflow equation (assignment) in an Nbac model can be as follows:

```
x' = if (init0) then 0
      else if (not init0) and ((not b1 and not b0) or (b1 and b0))
            then x+1
      else if (not init0) and true
            and ((b1 and not b0) or (not b1 and b0))
            then x
      else 0;
```

Figure 3.6: A dataflow equation in Nbac model

where x : Integer; $init0, b0, b1$: Boolean.

Since PMC/Uppaal does not support if-then-else statement in assignments, we have to model such statements in a reasonable way in an XTG model. We remove the if-then-else statement from the assignment by transforming it to a value expression containing only numerical operators. Consider the following assignment:

```
x:= if a then x1
     else if b then x2
     else x3;
```

where $x, x1, x2, x3$: Boolean or Integer; a, b : Boolean.

Although there is else-if syntax, the Boolean conditions associated with else-if component are already exclusive (this only holds for the Nbac output model)². The last else part is a fake just to fulfill the

²claimed by the author of Nbac

`if-then-else` syntax. The above assignment is modeled in an XTG model as

$$x = a * x1 + b * x2 + x3$$

where `*`, `+` are multiplication and plus operators on Integers. It is possible to model the `if-then-else` statement this way because `Uppaal` implicitly implements a conversion from Boolean to Integer value.

The resulting assignment of the example in Figure 3.6 is:

$$\begin{aligned} x' = & (\text{init0}) * 0 + \\ & ((\text{not init0}) \text{ and } ((\text{not b1 and not b0}) \text{ or } (\text{b1 and b0})) * (x+1) \\ & + (\text{not init0}) \text{ and } ((\text{b1 and not b0}) \text{ or } (\text{not b1 and b0})) * x \end{aligned}$$

2. **Parallel assignments in Nbac dataflow equations.** Assignments in Nbac are parallel. Those assignments set values for variables in the next execution using the current values of variables. In an XTG model, assignments are executed sequentially. Consider the following example:

$$\begin{aligned} \text{b0}' & := \text{b1}; \\ \text{b1}' & := \text{not b0}; \end{aligned}$$

where `b0`, `b1` are both state variables. They will be updated in the next execution step. In Nbac, setting the new value for `b0` does not affect the new value of `b1` since `b1` is computed based on the *current* value of `b0`. However, if we write `b0:=b1;b1:=not b0` in XTG, the value of `b0` in the second assignment is the new assigned value of `b0` in the first assignment. Consequently, additional variables are needed to sequentialize those parallel assignments. We save the current values of variables in temporary variables (`prime`). Then we replace all occurrences of the original variables on the right hand side of the assignments by temporary variables.

Let $(b_i)_{i=1\dots m}$ be the set of Boolean state variables, $(x_k)_{k=1\dots p}$ be the set of Integer state variables, $(c_j)_{j=1\dots n}$ be the set of Boolean input variables, and $(y_l)_{l=1\dots q}$ be the set of Integer input variables. All dataflow equations in the transition part

$$b'_i = \phi_i(\vec{b}, \vec{c}, \vec{x}, \vec{y}), i = 1\dots m$$

$$x'_k = \psi_k(\vec{b}, \vec{c}, \vec{x}, \vec{y}), k = 1\dots p$$

are transformed as follows:

$$\begin{aligned}
b_{i_prime} &:= b_i, i = 1 \dots m \\
x_{k_prime} &:= x_k, k = 1 \dots p \\
b_i &:= \phi_i(b_{prime}, \vec{c}, x_{prime}, \vec{y}) \\
x_i &:= \psi_k(b_{prime}, \vec{c}, x_{prime}, \vec{y})
\end{aligned}$$

As such, the two assignments above are specified in XTG as follows:

```

b0_prime := b0;
b1_prime := b1;
b0 := b1_prime;
b1 := not b0_prime;

```

External environment. As described in Chapter 2, Nbac uses *assertion* and *input* variables to model the external environment setting. At every execution step (when a transition is taken), the *assertion* is consulted to obtain new values for input variables. These values are set in a non-deterministic way provided that they satisfy the constraint stated by the *assertion*.

Hence, the external environment is actually specified implicitly in Nbac in the sense that we just have to declare the variables and conditions. Nbac takes care of other parts including generating relevant values in a non-deterministic way and getting new values at the beginning of every execution step. As a consequence, we have to model those processes in XTG explicitly.

Our solution to model the external environment involves two parts:

1. We introduce an additional graph namely *input graph* to generate values for *input* variables. Values are generated in a non-deterministic way and constrained by the *assertion*.
2. We set up a synchronous communication between the *system graph* and the *input graph* in order to give *input* values to the system at every execution step. This requires additional transitions and locations in the *system graph*.

Modeling the communication between the *system graph* and the *input graph* as a synchronous communication is a natural way due to the semantics of the communication:

1. At every execution step, input values are consulted by the system graph to compute new values for state variables and evaluate Boolean expressions on locations and edges.

2. The input graph computes input values from current values of state variables from the system graph, i.e. the latest values of variables from the system graph must be taken into account.

In other words, the system graph and the input graph must interactively execute. An asynchronous communication would not be relevant to model such communication.

Consider the example in Figure 3.7. $\mathbf{b1}, \mathbf{b2}$ are two *input* variables; \mathbf{c} is a *state* variable whose value is set by an expression on $\mathbf{b1}, \mathbf{b2}$.

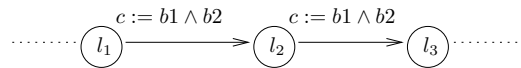


Figure 3.7: A simple system

It is required that *input* variables $\mathbf{b1}, \mathbf{b2}$ get their new values after the current transition is taken and before coming to a new location. Therefore, an intermediate location is added to model that requirement. We adapt our model as in Figure 3.8.

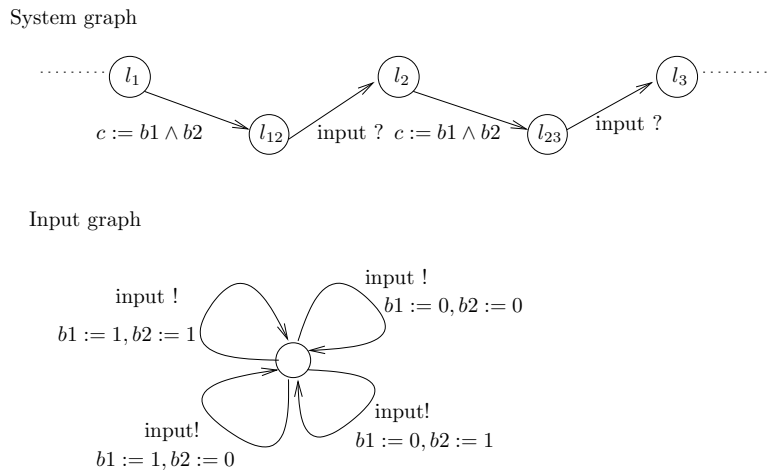


Figure 3.8: Modifying the system model to specify external environment

In Figure 3.8, there are a *system graph* and an *input graph*. There is a synchronization between two graphs through channel **input**. The model in Figure 3.8 is interpreted as follows: Before really coming to l_2 , the system will be in the intermediate location between l_1, l_2 to set the value for input variable. The *input* graph can take non-deterministically one transition and set the new value for $\mathbf{b1}$ and $\mathbf{b2}$. Thus, when the system is in the location l_2 , $\mathbf{b1}, \mathbf{b2}$ have been set to the right values.

The example in Figure 3.8 is simple because there is no constraint (*assertion*) on **b1**,**b2** and thus there is no guard on transitions in the *input graph*. However, it is often the case that the *assertion* constrains the values of input variables. For example:

```

assertion
    (a1 and b1 and b2) or
    (not a2 and b1) or
    (a1 and a2 and b2)

```

In this example, **a1**, **a2** are *state* variables and **b1**, **b2** are *input* variables. Now we have to specify how **b1**, **b2** are set using the above *assertion*.

First, we explicitly state the condition for combinations between **b1**, **b2**, meaning that the Boolean expression in the above assertion is transformed to the following equivalent expression:

```

(a1 and b1 and b2) or

(not a2 and b1 and b2) or
(not a2 and b1 and not b2) or

(a1 and a2 and b1 and b2) or
(a1 and a2 and not b1 and b2)

```

Then we group conditions according to combinations of **b1**, **b2**. We get the following expression:

```

((a1 or (not a2) or (a1 and a2)) and (b1 and b2)) or
((not a2) and (b1 and not b2)) or
((a1 and a2) and (not b1 and b2)) or
(not b1 and not b2)

```

Note that the last case (**not b1 and not b2**) is not stated anywhere in the *assertion*. That means **b1** and **b2** can be both **false** without any constraint. The final *input graph* of the example is presented in Figure 3.9.

In addition to those aforementioned adaptations, we introduce an additional location to play as the *fake* initial location and create a transition from this location to the *truly* initial location so that input variables can be set properly at the start of execution.

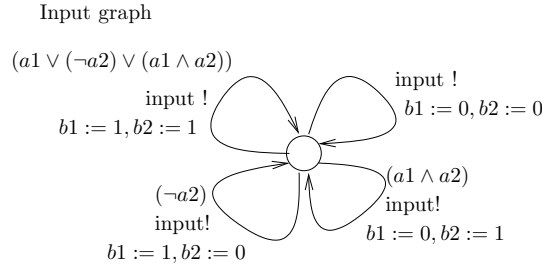


Figure 3.9: The final *input* graph

Initial condition. An *initial* string in Nbac is a Boolean expression on Boolean variables. For example:

```
initial init and ok;
```

The meaning of this *initial* condition is that the system starts with `init` and `ok` are set to `true`. In XTG if we declare a variable without an initialization, it is set to zero. Therefore, all variables declared as `true` by the *initial* condition must be initialized. Thus, instead of :

```
int x,y;  
bool init,ok,b0,b1;
```

We will have to declare:

```
int x,y;bool b0,b1;  
bool init:=true;  
bool ok:=true;
```

Final condition. In an Nbac model, the *final* condition defines the condition under which the system violates the safety property. Therefore, the *final* condition is used to specify the safety property of the model. For example, if the *final* condition is “not `init` and not `ok`”, the property specification will be: “`AG not init and not ok`” in a PMC model or “`A[] not init and not ok`” in an Uppaal model. It means that the system always satisfies the condition “not `init` and not `ok`”.

3.4 Implementation

In this section, we present the implementation of the transformation tool based on the abovementioned transformation framework. This tool takes an `Nbac` output model as the input and outputs an `XTG` model.

Taking into account the syntax of `Nbac` output models, `XTG` input models, and the transformation framework, we choose the flexible scripting language `Perl` as our implementation language. We use `Perl` standard functions to process the input models and `XML::LibXML` package to build the `XML` tree of the output model.

In two subsections below, we present more details about two versions of our transformation tool:

- `nbac2xtgxml`: the tool to transform an input `Nbac` model to an `XML` representation of `PMC` model
- `nbac2uppaal`: the tool to transform an input `Nbac` model to an `XML` representation of `Uppaal` model

3.4.1 `nbac2xtgxml`

We implemented the transformation tool `nbac2xtgxml` according to the framework described above. Figure 3.10 depicts the input and output of the tool.

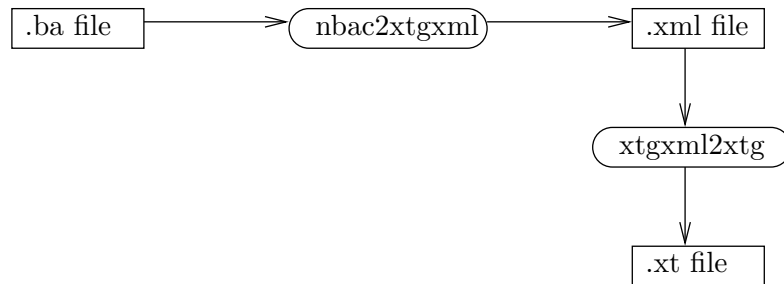


Figure 3.10: Input and output of `nbac2xtgxml`

In Figure 3.10, the `.ba` file is generated by `Nbac` using the `-obdd` option. The transformation tool exports an `.xml` file defined for `PMC` model. After we have the `.xml` file, we use the tool `xtgxml2xtg`³ to transform it to an `.xt` file that can be an input for `PMC`.

³developed by another group

When applying the abovementioned procedure to implement `nbac2xtgxml`, we faced the following problems:

1. As explained in Section 3.2, expressions have to be split into atomic elements in PMC models. In the implementation, we had to write procedures to parse expressions obeying the precedent rule on operators.
2. PMC does not support Boolean operations on Boolean type variables. Therefore, `Nbac` Boolean variables are saved as integer variables in PMC model. The Boolean expression “`b and not c`”, in which `b, c` are Boolean variables, is specified in PMC model as “`b==1 and c==0`”.
3. Since PMC neither supports Boolean type variables nor returns Boolean values for comparison expressions, `if-then-else` statement in assignments cannot be modeled as described in Section 3.3. A number of extra locations and transitions have to be added to model assignments. Consider a set of assignments for a transition in the following example:

```
x:= if a then x1 else if b then x2 else 0;
y:= if c then y1 else if d then y2 else 0;
z:= if e then z1 else if f then z2 else 0;
```

Figure 3.11 shows how the transition is modeled in Uppaal.

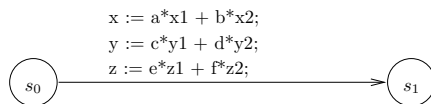


Figure 3.11: Modeling assignments in Uppaal

Figure 3.12 shows how the transition is modeled in PMC.

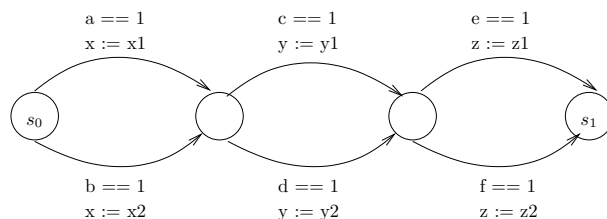


Figure 3.12: Modeling assignments in PMC

Modeling assignments this way adds a large amount of transitions and locations to the PMC model. However, we had no other choice to solve this problem.

4. PMC does not support logical OR operator on invariants. We cannot also use AND and NOT to model OR on invariants. Since Nbac employs BDD structures to store Boolean expressions, text representations of those Boolean expressions are formed by logical AND, OR, NOT operators. The lack of this support in PMC makes it impossible to obtain an equivalent PMC model from an Nbac output model.

Since the output Nbac model is already complicated in the sense that it contains a lot of locations, transitions and Boolean expressions, adding a number of locations and transitions to the model makes it extremely big in size. The practical result shows that PMC could not even initialize the data structure for the Fischer’s protocol consisting of two processes. The memory usage to initialize that model exceeds 2.2GB. Furthermore, the lack of support OR logical on invariant prevents us from verifying any systems. For those reasons, we had to turn our focus to Uppaal, which is more mature than PMC in terms of functionality.

3.4.2 nbac2uppaal

We also follow strictly the framework described in Section 3.3 to implement this tool. Figure 3.13 presents the input and output of the tool. The tool

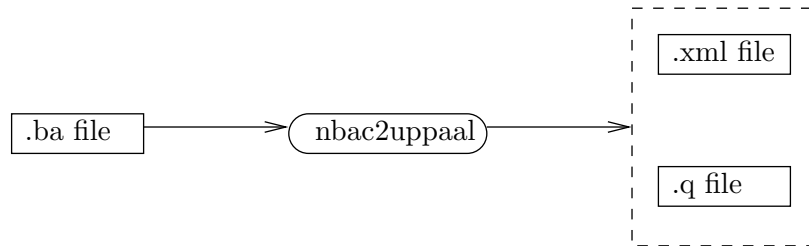


Figure 3.13: Input and output of nbac2uppaal

nbac2uppaal generates two file: .xml file and .q file. The .xml file contains the system model and the .q file contains the property to be verified. We test the functionality of this tool with a number of models and it works properly for all cases we have tried it on.

Note that in the resulting Uppaal model, there is no graphical information of the model because there is no such information in Nbac output model. Therefore, the model will be poorly placed and routed when we view it using the GUI version of Uppaal especially when the model contains too many items.

3.5 Summary

In this chapter we have presented the development of the tool to transform an `Nbac` output model to an `XTG` model. Since the size of an `Nbac` output model is too large to be transformed manually, the transformation tool plays an important role in making our approach feasible.

After the tool `nbac2xtgxml` was built, we found that `PMC` could not be suitable for our approach and then we decided to switch to `Uppaal`. Although the `PMC approach` based on partition refinement might be impressive according to comparative experimental results presented in [23], the lack of a few features make it impossible to be employed in our approach.

We have presented so far the construction of an input model and the transformation technology to connect `Nbac` and `PMC/Uppaal`. We will describe a few case studies in the next chapter before doing the experiments.

Chapter 4

Case studies

This chapter contains four systems which are of different complexity and application domains chosen as case studies for our approach. We roughly classify them according to physical size and characteristic of the state space. We define classification standards as follows:

1. **Physical size:** In our context, physical size is represented by the number of locations, transitions, Boolean operators, and variables. Since input models are merely text files, the physical size may have significant impact on the performance of tools applied. Our goal is to see whether or not the physical size of a system affects the efficiency of verification. We define two categories: **small** and **large**. A system is categorized as small in physical size if the total number of locations, transitions, Boolean operators, and variables is less than 10000 and vice versa.
2. **Characteristic of the state space:** Two categories are defined including Finite and Infinite. We found two reasons for which a state space becomes infinite:
 - (a) The system involves unbounded numerical variables.
 - (b) The existence of real-time in the system.

We characterize the state space of models in the perspective of `Uppaal` model checking approach. In an infinite system due to unbounded variables, the iterative resolution of a fixed point equation *defined by Uppaal* involves infinite iterations. Note that the fixed point equation defined by `Nbac` may have a solution. It depends on the type of values on which the fixed point equation is defined. For example, values $x = (1..n)$ may be abstracted as a unique value $pos = x > 0$ in `Nbac`. In this case, if x always increases, the

fixed point can be reached without further analysis on every values of $x > 1$. On the contrary, `Uppaal` does not employ such abstraction. Integer values are handled exactly as they are and thus the fixed point equation can be infinite. A detailed explanation of fixed point theory and its application in model checking is presented in the research report[16].

The original models of the four systems described in this chapter are not the same source; They were initially specified in `Nbac`, `PMC/Uppaal`, and `Lustre` [2] input language. All of them are however eventually transformed into `Nbac` input language. Therefore, it is possible that the original model is simple but the `Nbac` output model is complicated due to transformation and abstraction steps. We categorize these systems based on output models from `Nbac` which are inputs for verification done by `PMC` or `Uppaal`.

Practical models of the case studies in text format can be found in the Appendix section.

4.1 The tiny example

The tiny example is taken from `Nbac` tutorial[21]. This simple system has two counters x, y which are alternatively incremented. It is represented by the automaton shown in Figure 4.1.

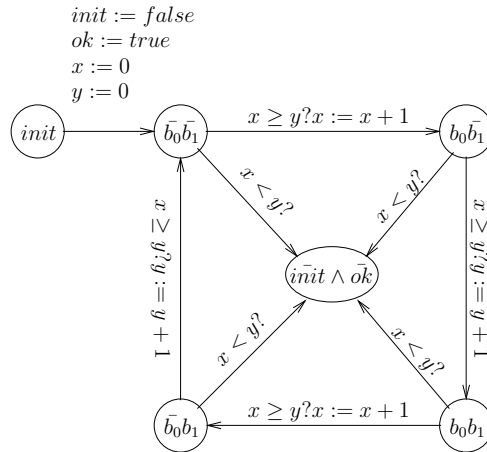


Figure 4.1: The tiny example

In Figure 4.1, the system starts from location `init`. Since x and y are initialized by zero, the system will take the transition to location `b0b1` and increase x by 1. Then y and x are increased alternatively.

The safety property to prove is $x \geq y$. It is readily seen that the system meets the safety property.

We choose this example as one of our case study in order to get an insight on its models due to the simplicity of the system. Thus, we can trace the output model manually to check whether or not the transformation tool works properly.

The tiny example is originally specified in Nbac input language. Therefore, the meaning of the model is very straightforward. See Appendix A.1 Two input variables $p1, p2$ was added just to see how assertion is generated and the impact of input variable in the model.

This example is categorized as infinite system caused by unbounded Integer variables. If we let the system run for a long time, x, y will exceed the bounded Integer value.

4.2 Asynchronous Reader/Writer Algorithm

In [11] J. Chen and A. Burns describe an algorithm that implements a fully asynchronous reader/writer algorithm which addresses the problems of blocking and priority inversion within multi processor real-time systems. The approach is conceived from the concept of process consensus in which the writer and the reader come to an agreement on accessing the shared data before proceeding to carry out their respective data operations. The PMC/Uppaal model of this algorithm for one reader and the writer is presented in Figure 4.2.

In Figure 4.2, there are two graphs: one for the Reader process and one for the Writer process. The algorithm uses shared variables $L1, L2, Prefer1, Prefer2, Precon$ between Reader and Writer to control these processes.

The safety property states that whenever the writer and the reader are actually in the state of writing and reading simultaneously the writer and the reader index must be unequal. This property is specified in Uppaal as “ $A[] \text{ not } (Reader.reading \text{ and } Writer.writing \text{ and } Reader.idx == Writer.idx)$ ”.

This algorithm is originally modeled in PMC. It is then transformed manually to make it an Nbac input model. This step increases the physical size of the model significantly. Then Nbac generates an output model from its input and thus increases the size of the model even more.

This case study is categorized as finite system because all variables are

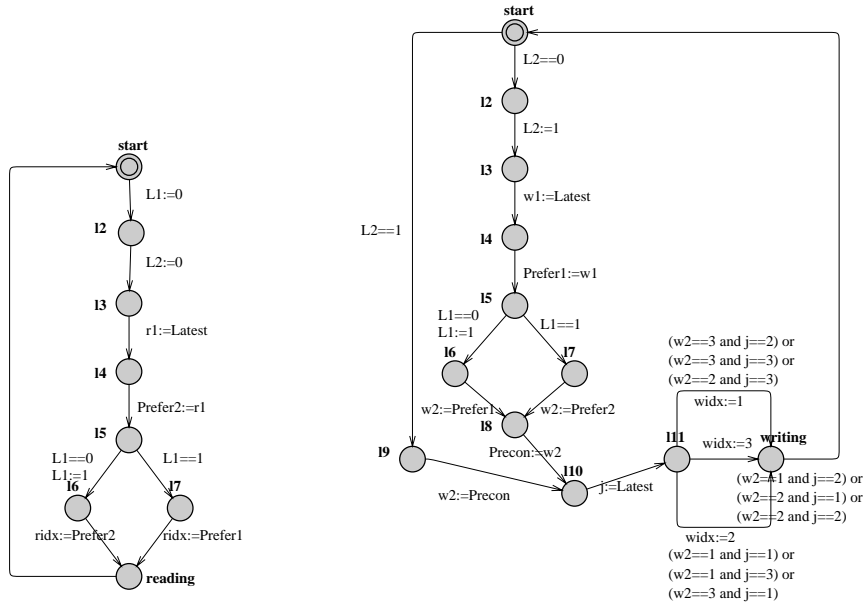


Figure 4.2: Reader and Writer graph for the algorithm

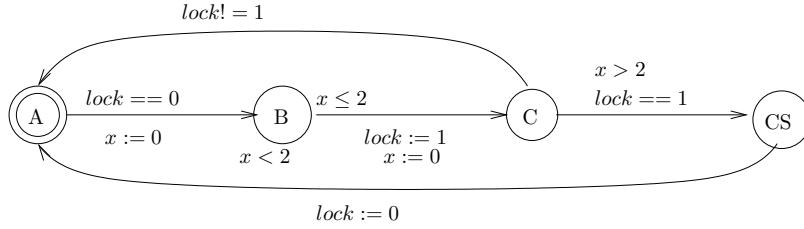
bounded. Moreover, there is no real-time in this model. Since this model contains many locations and transitions, the Nbac model is large in terms of physical size.

We choose it as a case study because PMC proves that the algorithm does not satisfy the safety property. We are interested in checking whether we get the same result using our approach. However, Nbac and Uppaal can prove the safety property separately. Despite of that fact, this system still remains a good case study for its characteristic which is a finite system and large in physical size.

4.3 Fischer's protocol

Fischer's protocol is a mutual exclusion algorithm in which n processes uses timing constraints and a shared variable to ensure mutual exclusion. Figure 4.3 shows two processes from Fischer's protocol. Each process has a unique identifier, a clock x and 4 locations A, B, C and CS . They share a common variable namely $lock$. From location A each process can make a transition to location B if $lock$ has value 0. It stays in B for k seconds before assigning its id to $lock$. After setting value for $lock$, it goes to location C . Then the process may enter the critical section (location CS) after it spends at

Process 1



Process 2

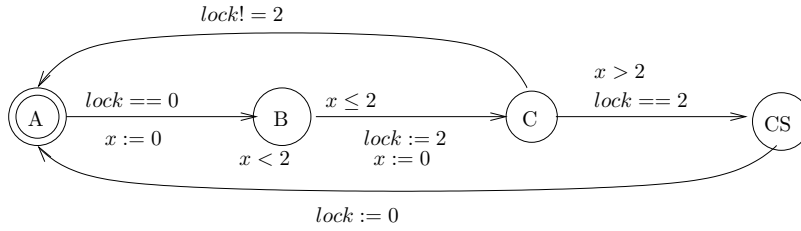


Figure 4.3: Fischer's protocol

least k seconds in C and the value of $lock$ is still equal to its identifier. This constraint is to make sure that other processes which stay in B already make the transition to C . Upon leaving its critical section, it sets $lock$ to 0.

The safety property states that it is impossible for more than one process to be in the critical section at the same time. The model satisfies this safety property. This property is specified in PMC as “ $AG \text{ not } P1@CS \text{ and } P2@CS$ ”. This protocol is widely used as a case study for real-time model checkers benchmark because it can be easily scaled up by including more processes.

This system is classified as small in physical size. However, the state space is infinite due to real-time notion. The original model is an XTG model. An Nbac model has been manually constructed from the original model. In the Nbac model, time is modeled as discrete clock tick. In Section 3.3, we have explained the construction of this system from an XTG model.

In this system, the time is physically modeled as clock tick. An Integer variable x is used to represent the elapsed time for a clock variable c in an XTG model. A time action increases the value of x by 1. The state space of this system is infinite due to the fact that x is unbounded.

4.4 Subway system

This case study is a subway regulation system taken from [22]. It concerns a (simplified version of a) speed regulation system avoiding collision. Each train detects beacons that are placed along the track, and receives the “clock tick” from a central clock. Ideally, a train encounters one beacon each clock tick. The space between beacons rules the speed of the train. Now, a train adjusts its speed as follows: let b and s be respectively the number of encountered beacons and the number of received clock ticks.

- When $b \geq s + 10$, the train notices it is early, and puts on the brake as long as $b > s$. Continously braking makes the train stop before encountering 10 beacons (using a counter c).
- When $b \leq s - 10$, the train is late, and will be considered late as long as $b < s$. A late train signals it to the central clock, which does not emit the “second” as long as at least one train is late.

Figure 4.4 shows the model for one subway.

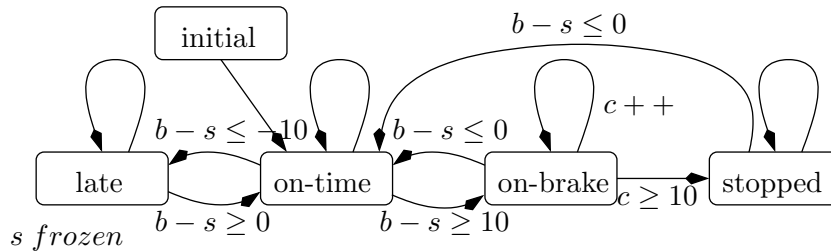


Figure 4.4: The model for one subway

The property to check is:

1. A train cannot move in one step from the state **late** to the state **on-brake**, and conversely.
2. The number $b - s$ remains in the interval $[-10, 20]$.

The original model is written in LUSTRE[2], a synchronous programming language. Then this model is transformed to an Nbac model by a tool made by the author of Nbac. For that reason, the Nbac version of this system is too difficult to make any sense to human. The Nbac output model and transformed PMC/Uppaal model are even more complicated. Therefore,

this case study is a very good example to test the capability of our transformation tool.

This case study is classified as small in physical size. The state space is infinite because of the increment of Integer variables.

4.5 Summary

This chapter introduced four case-studies that are used to evaluate our approach for improving efficiency of model checking through systematically combining Nbac and PMC/Uppaal.

We choose case studies so that we can evaluate the technique at different typical models. Although the ultimate goal is the verification of huge and infinite models, simpler models will help to characterize the effect of factors such as size, state space, etc. on verification.

The characteristic of four case studies are summerized in Table 4.1 and 4.2. Table 4.1 gives a general view about the characteristic of those case-studies. Table 4.2 presents the physical size of those systems in terms of the number of locations, transitions, Boolean operations and variables.

Table 4.1: System characteristics

System	Physical size	State space
Tiny example	small	infinite-variable
Asynchronous reader/writer alg.	large	finite
Fischer's protocol	small	infinite-realtime
Subway system	small	infinite-variable

Nbac is able to generate several different abstract models of a system. Table 4.2 shows a few abstract models distinguished by the number of locations for every system. **Fischer-2** means that the system is of the Fischer's protocol that consists of two processes. **Subway-2-p1** means that the system is of a subway system that consists of two subways and the property to verify is property 1. **Subway-2-p2** is the model for which Uppaal could not even initialize the data structure since memory was exhausted.

An observation on Table 4.2 shows that there is no environment factor in the Tiny example and the Asynchronous reader/writer algorithm. In addition, the number of Boolean operators applied in the Asynchronous reader/writer algorithm is very large (more than 20000) in compared with other models. We expect that these features could affect the verification performance.

Table 4.2: System physical size

System	Shared vars	Input vars	Locations	Edges	Bool opt
Tiny	6	0	3	3	48
Fischer-2	12	3	3	3	2117
Fischer-2	12	3	4	7	1826
Fischer-2	12	3	6	14	1722
Fischer-2	12	3	7	23	1649
Fischer-2	12	3	13	36	1672
Subway-2-p1	13	3	3	3	336
Subway-2-p1	13	3	4	7	1050
Subway-2-p1	13	3	6	13	2555
Subway-2-p1	13	3	9	27	3117
Subway-2-p1	13	3	14	49	3542
Subway-2-p1	13	3	23	84	3233
Subway-2-p2	13	3	562	3533	20528
Asyn. r/w alg.	32	0	3	3	59484
Asyn. r/w alg.	32	0	4	6	21070
Asyn. r/w alg.	32	0	39	48	24456

In the next chapter, we will present our experiments on verifying these case studies. Although the performance of verification is widely known as system-dependent, we believe that experiments on such diverse case studies will help to evaluate the approach, characterize problems to improve or adapt it.

Chapter 5

Experimental results and problem characterization

In the previous chapters, we have presented our work to prepare all required parts to do the verification in our approach:

1. Specifying the input model of the system being verified.
2. Developing the transformation tool to provide with a connection between `Nbac` and `PMC/Uppaal`.
3. Selecting case studies for the experiments.

We are now ready to observe outcome of the combination approach. Basically, we expected that the verification based on our combination approach would be more efficient than verification taken by `PMC/Uppaal` only. Then, we would find out how we should coordinate the tools, which level of abstraction we should use to obtain the most efficient model checking or a set of parameters to control the efficiency of model checking under that approach. However, the practical results are not as we expected. Therefore, we characterize the problem and try a few techniques to improve the situation.

This chapter describes the practical results we got so far and the problem characterization. All experiments are conducted using `Uppaal`. We could not use `PMC` in our approach for the reason that it was really a prototype and thus immature for being involved in our approach.

Section 5.1 presents all needed steps to do an experiment. Section 5.2 shows the practical results obtained from the verification of case studies described in Chapter 4. Then we explain our reasoning on models to figure

out the problem characterization and techniques applied to deal with the problem in Section 5.3.

5.1 Experiment procedure

Since our approach involves several tools and steps, we set up a uniform procedure to do an experiment which consists of the following steps:

1. *Construct a model in Nbac input language as we discussed in Section 2.3.* The input model can be directly specified in Nbac input language, or manually transformed from an XTG model, or automatically transformed from a Lustre program.
2. *Input the abovementioned model of the system to Nbac.* Use the following command to get the abstract model:

```
>nbac -analysis 2 foo.ba --partition "-maxloc N" -obdd  
-o bar.ba
```

The meaning of options in the above command is as follows:

- `foo.ba`: file containing an input model
 - `bar.ba`: file containing an output abstract model
 - `-analysis 2`: type of analysis is a combination of forward and backward analysis
 - `--partition "-maxloc N"`: this option controls the level of abstraction by limiting the number of locations in the control structure. The larger N, the less abstract the model.
 - `-obdd`: this option is used to specify the format of the Boolean expression
 - `-o`: output the abstract model to a file
3. *Chose some models to do experiments.* Nbac generates several abstract models. The level of abstraction is represented informally by the number of locations. We prefer the model that contain 3-4 locations, meaning that N in the above command is set to 3 or 4, for our convenience in the simulation and problem analysis since it turned out that we had to trace the execution by hand for a few cases.
 4. *Transform Nbac abstract model to Uppaal model.* We use the tool `nbac2uppaal` we have developed to do this step. The command is as follows:

```
>nbac2uppaal.pl foo.ba foo.xml foo.q
```

The first argument is the output abstract model. The second argument is the system graph and the last one contains the property to check.

The tool also provides information about the physical size of the resulting XTG model including the number of locations, edges, Boolean operators and variables. For example, after transforming the Fischer's protocol, the tool reports the following information:

```
Successfully transformed fischer.ba to fischer.xml
Number of shared variables:12
Number of input variables:3
Number of locations:3
Number of edges:3
Total number of boolean operations: 2353
```

5. *Verify the abstract models using Uppaal.* After obtaining the abstract model, we start to verify the model using Uppaal. We prefer to use the command line version instead of a GUI-supported version of Uppaal so that we can measure¹ the time and memory usage of the verification. The command to verify is as follows:

```
>verifyta foo.xml foo.q
```

where `foo.xml` is the file containing system graph and `foo.q` is the file containing the property specification.

We use a threshold time of 4 hours to classify the model as practically verifiable or not.

We have presented above the procedure we define to do experiments. Next, we report experimental results for each case study.

5.2 Experimental results

5.2.1 The tiny example

The original model of the tiny example is specified in Nbac input file `tiny.ba`. Nbac generates only one abstract model which contains 3 locations along the run.

¹using `memtime` utility created by the developers of Uppaal

Since this case study is very small, the performance of verification was not our concern. Our goal was to see how the verification result would be. `Uppaal` could prove the safety property for the only abstract model generated by `Nbac`.

Although `Uppaal` could prove the safety property, it notified that “*State discarded: Assignment to variable x is out of range*”, meaning that `Uppaal` only proved the safety property for x was in the range of integer values. This warning may not cause too bad impact to the satisfaction of verification people because it is likely that the system is supposed to work for integer values only. Nevertheless, this fact could shed some light on the way `Uppaal` verified the system.

5.2.2 Asynchronous reader/writer algorithm (aka Burn’s algorithm)

The original model of this algorithm is specified in PMC input file `burn_simple_2.xt`. We constructed an `Nbac` input model for this algorithm from the PMC input model using the framework specified in Section 2.3. The resulting model is saved in file `burn.ba`. `Nbac` can generate 23 abstract models for this algorithm.

`Uppaal` gives a conclusive result for the safety of this system on the verification of the abstract model that contains 3 locations.

Although we expected the system would not satisfy the property as proved by PMC, `Uppaal` proved that the safety property was indeed satisfied using the `Nbac` abstract model. We were skeptical that it was the abstraction technique that affected the verification results. Therefore, we made an equivalent `Uppaal` directly from PMC model since it was probably that the transformation from PMC model to `Nbac` model had some unfilled gaps. This `Uppaal` model is saved in file `burn.xml`. Again, `Uppaal` did prove the property for the same model with PMC. However, it is not our goal to see what was wrong with the model checkers so we did not go further into this issue.

Since `Uppaal` could give a conclusive result for the algorithm, we had a chance to test the efficiency of verification on several abstract models of this algorithm. However, we found that the difference of time and memory usage between different abstract models was insignificant. The difference is mainly due to the difference in the physical size of the model. We believe that this system is too simple for the verification results on different abstract models to be distinguishable. Table 5.1 presents some measurement data for this test.

Table 5.1: Time and memory usage of the verification on different abstract models of the algorithm

Number of locations	Number of edges	Time	Memory usage
3	3	0.55s	13208KB
15	18	0.22s	3324KB
39	48	0.44s	10724KB
71	95	0.98s	15116KB

5.2.3 Fischer’s protocol and Subway system

The Fischer’s protocol input model is constructed from an *XTG* model. The manual transformation process was a big effort even for the input model consisting of only two parallel processes. We had to revise it several times before getting a confidence on the correctness of the model. The final model, which is saved in file `fischer-2.ba`, can be found in Appendix A.3.

The original model of the subway system is specified in Lustre input file `metros.lus`. The system being verified involves 2 subways. We derived three different models according to the property of interest: property 1, 2a and 2b. The properties 2a, 2b are two different parts that form property 2 (see Chapter 4). Those models are saved in file `metros-prop1.ba`, `metros-prop2a.ba`, and `metros-prop2b.ba`.

In this thesis, we only present the result of model checking property 1 for the subway system since there is no difference in the result of model checking between either of three properties. This property states that “A train cannot move in one step from the state *late* to the state *on-brake*, and conversely.”. `Nbac` could generate 6 different abstract models for the model to verify property this property.

The verification result is not very encouraging since we have no conclusive answer about the truth of the property for either two case studies. `Uppaal` could not finish the verification on any model before exhausting all available memory resource.

5.3 Problem characterization

As presented in Section 5.2.3, the verification experiments of the Fischer’s protocol and Subway system did not give any conclusive answer due to the exhaustion of memory resource during verification. This result implies that the state space of those systems are too large for `Uppaal` to finish the exhaustive search within available memory resource. An estimation of

the size of the state space based on the amount of memory usage is shown hereafter.

State space explosion has been a well-known problem of model checking in practice. Therefore, it is not a surprise that we could not successfully verify a model. However, experimental results from verifying models of the Fischer’s protocol and Subway system are still disappointing since `Nbac` could prove the same models in a short time. We then figure out the reason for such large state space to answer the second question posed in Section 1.3 of this thesis.

We tried to get a view on the size and growth of the state space using a practical method. We used the simulation utility of `Uppaal` to see how the system worked. Then we found the execution trace looped around the *good* location. First, we measured the memory usage with respect to the number of iterations around the *good* location. This measurement was to estimate the size of state space of the model and to see how it grew after every iteration. To measure the memory usage for that purpose, we modified the resulting `Uppaal` model manually by adding a counter so that if the number of iterations around the good location exceeded the constant specifying the maximum number of iterations allowed, the system would stop and thus the verification could finish. The implementation of the abovementioned mechanism is illustrated in Figure 5.1.

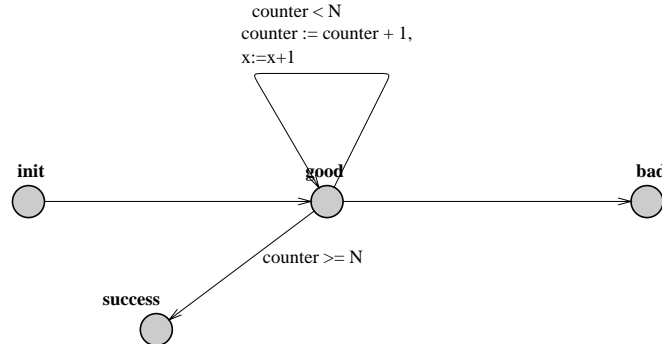


Figure 5.1: Adding a counter to restrict the number of iterations

In Figure 5.1, N is a constant specifying the maximum number of iterations allowed. From the measurements on the verification of the modified model, we found that the memory usage grew quickly as we increased constant N . Figure 5.2 presents the memory usage of the experiment for the subway model. The maximum N for which the verification could be finished is 19.

The measurement showed that the state space created from the abstract

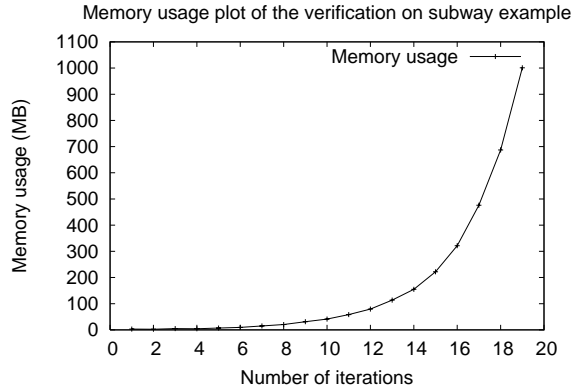


Figure 5.2: Memory usage of the experiment for the subway system

model was significant for **Uppaal** to verify. We tried to identify the specific reason for such a large state space.

The first reason might be the physical size (defined in Chapter 4) of the model. However, a comparison between abstract models of Fischer’s protocol, Asynchronous reader/writer algorithm and Subway system showed that physical size was not the real reason. Table 5.2 presents that comparison. Although the model of Asynchronous reader/writer algorithm contains a large number of variables and Boolean operators, we can still get conclusive verification result for that algorithm.

System	Shared vars	Input vars	Locations	Edges	Bool opt
Fischer-2	12	3	3	3	2117
Subway-2-p1	13	3	3	3	336
Asyn. w/r alg.	32	0	3	3	59484

Table 5.2: Comparison between three models

From the above comparison, we noticed that the difference in the number of input variables between those systems could be a reason for large state space especially when input variable involved non-deterministic factor. Remind that in an **XTG** model, the values generated using the input graph control the behavior of the system. However, taking a closer look at the input graph and doing the math, we see that although values of input variables are assigned non-deterministically, the total number of cases to assign input values are finite. Figure 5.3 depicts this argument.

The right graph in Figure 5.3 is the input graph. It generates two values 0 and 1 in a non-deterministic manner for the system graph on the left. The system graph specifies that if **b** is larger than 1, the system will go to end.

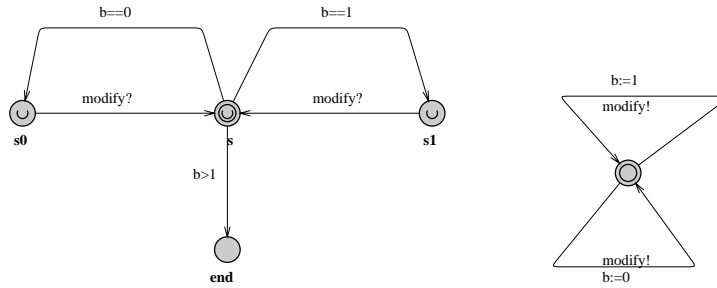


Figure 5.3: A finite system

Uppaal could prove the simple system in Figure 5.3 very quickly. Hence, the existence of input variables in the model is not the major reason for state space explosion.

The warning we got when verifying the tiny example suggested us to switch our concentration to the way values of variables changed along the simulation. It turned out that values of variables in abstract models of the Fischer’s protocol and Subway system increased forever, and thus the state space is infinite. Figure 5.4 shows a modified version of the above simple model that illustrates this phenomenon in verification performed by Uppaal.

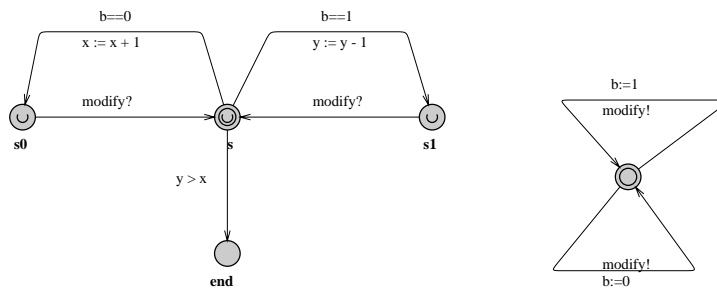


Figure 5.4: An infinite system

The system starts with x and y are both zero. Although we can easily see that there is no way for the system to go to location “end”, UPPAAL could not prove it. In our observation, verification done by Uppaal in this case is very much similar to a simulation on all possibilities of variable values rather than a linear relation analysis on variables in a model. We believe that this is the main reason for the state space explosion in model checking

the Fischer’s protocol and Subway system.

The aforementioned reason for state space explosion can be explained by the model checking approach implemented in `Uppaal` [8]. The model checking engine of `Uppaal` performs an exploration of the reachable state space, the set of states reachable from the initial state, on-the-fly. For each unexplored state, its successors are computed and compared to already explored states. If a successor has already been seen in the past, it is discarded. On the other hand, if a successor has not yet been seen, it is added to the list of states waiting to be further explored.

`Uppaal` approach to verify safety properties is founded on the fixed point theory. The reachable state set can be characterized as the least fixed point of the following function:

$$F(X) = S_0 \cup Next(X)$$

where S_0 is the set of initial states and $Next(X)$ denotes the direct successors of X . Let x be an Integer variable, $Next(x) : x \rightarrow x + 1$. Then this fixed point equation involves infinite iterations. Although the computation may stop when x exceeds the bounded Integer value, the fixed point cannot be reached.

In Table 5.3, we show a few first states that result from one execution of the model presented in Figure 5.4 provided that the input sequence for the Boolean variable \mathbf{b} is $(0, 0, 1, 1, 1, 0)$. The model checking engine in `Uppaal` has to traverse through all paths of the execution before giving the answer about the truth of the property which is stated that the system never goes to the *end* location. We can see that x will be increased and y will be decreased infinitely. An infinite sequence of input values (\mathbf{b} in this case) contributes to the explosion because it makes `Uppaal` search the state space infinitely.

State #	Location	x	y
1	s0	1	0
2	s0	2	0
3	s1	2	-1
4	s1	2	-2
5	s1	2	-3
6	s0	3	-3

Table 5.3: Simulation data of the system in Figure 5.4

Hence, in Figure 5.1, every new execution of the cycle around the *good* location gives rise to new states, i.e. new value for x . As described above, the model checking procedure only stops when no new state is added, i.e.

the reachable state space remains unchanged. When `Uppaal` deals with such infinite system, it will search on all possibilities for Integer variable x in the default range of Integer numbers, i.e. $[-32768, 32767]$. This method leads to the state space explosion in the verification of the class of infinite systems that involve unbounded Integers.

Since we could not touch the underlying algorithm of any model checkers, we had to find some ways to solve the above problem at the model level. We tried to put a restriction on the number of iterations around the good location a system can take. This mechanism should be done automatically. There are two ways we can restrict the number of iterations in a model:

1. Restrict the number of iterations directly using a counter and limitation as we have done to measure the memory usage after every iteration. We implemented that restriction for the model containing one good location by modifying the transformation tool so that it can generate a model as presented in Figure 5.1.
2. Restrict the length of sequence of input values generated by the input graph. It would be more natural if we constrain the number of times a system goes through a good location this way. However, we did not implement this option for the following reason:

Sometimes the system can be blocked on the good location(s) by input values from the external environment. Consider Figure 5.5.

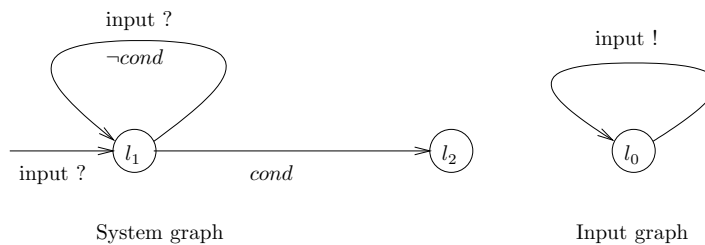


Figure 5.5: An illustration of blocking due to input values

In Figure 5.5, it is possible that after getting input values for the first time in location l_1 , the guard $cond$ is not enabled yet and thus the transition (l_1, l_2) cannot be taken. We modify the model so that the system will try to get input values for several times until the guard on transition (l_1, l_2) is enabled. In the case where there is no such input value that (l_1, l_2) can be taken, there must be something wrong in the model.

Remember that the execution of two graphs are alternative, meaning that the schedule for two graphs is $(system, input, system, input, \dots)$.

Hence, if we limit the length of the sequence of input variables, there could be several cases in which only the transition to get input values again is taken instead of a real transition action in the system. Therefore, modeling the restriction in this way does not help to show the actual workload of `Uppaal` in the verification.

Although the aforementioned solution does help to make `Uppaal` finish the verification, it does not solve the problem at its root since doing that way means we only verify a small part of the real state space. We are looking for a solution to remodel the `Nbac` output model in such a way that the resulting model becomes finite for `Uppaal`. Solving the verification problem for infinite systems is a research problem that is still on progress in many research groups. For such systems, theorem proving has been relied on [20, 18]. Abstraction techniques have also been used to solve that problem [9, 14, 10]. However, we ran out of time for any further investigation.

5.4 Summary

We have presented experimental results that we obtained on the verification of abstract models of case studies. `Uppaal` could finish verification on the tiny example and the asynchronous reader/writer algorithm in a short time. However, it could not verify the Fischer's protocol and Subway system.

We measured the memory usage in the verification for a range of number of iterations around the good location that the system can take. It turned out that the memory usage grew very quickly and thus state space explosion was unavoidable. We investigated the problem through several trials to figure out the reason for the state space explosion. The reason is that those systems are infinite due to the unlimited increment of Integer variables and thus the fixed point can not be reached. `Uppaal` will only terminate verification when it reaches the bounded integer value. However, the memory resource was exhausted before `Uppaal` could reach that limit when model checking the Fischer's protocol and Subway system.

We tried to solve the problem by restricting the number of iterations so that `Uppaal` can finish the verification. Among two different ways to do it, we chose to implement the counter and limit the value of the counter in the system graph.

Chapter 6

Conclusions

In this master thesis we introduced a systematic approach to combine a tool that provides abstraction technique and a model checker to improve efficiency of model checking. In short, our approach uses an existing tool to generate abstract models for a model checker to verify. We hoped that the abstraction tool would help to reduce the model of the system being verified and thus model checking on that model would be more efficient. If this approach works, it will help to employ abstraction in model checkers without directly implementing the abstraction function in those model checkers.

`Nbac`, `PMC` and `Uppaal` are tools involved in the implementation of our approach. `Nbac` is a model checker that has a built-in abstraction utility which is based on the theoretical framework of abstract interpretation. `PMC` and `Uppaal` are two model checkers aimed at real-time systems. We started the implementation of our approach by combining `Nbac` and `PMC`. Due to the fact that `PMC` lacks in some important features such as Boolean variables and operators, logical operator `OR`, we had to switch our implementation to combining `Nbac` and `Uppaal`. Our approach is depicted by Figure 6.1.

Of course, to do the model checking, an input model of the system is needed. Since `Nbac` is responsible for generating abstract models, the input model must be specified in `Nbac` input language. `Nbac` input language is a low level language mainly consists of variable declarations and dataflow equations on variables. As a result, specifying an input system in `Nbac` specification language is a complicated job for verification engineers. Among three different ways to specify an input model, transforming a model modeled in `PMC/Uppaal` to an `Nbac` model is likely to be the most popular one. In Chapter 2, we have established a framework to do such transformation. It turned out that even when a transformation framework is available, the manual transformation is an error-prone process. A tool to take over that

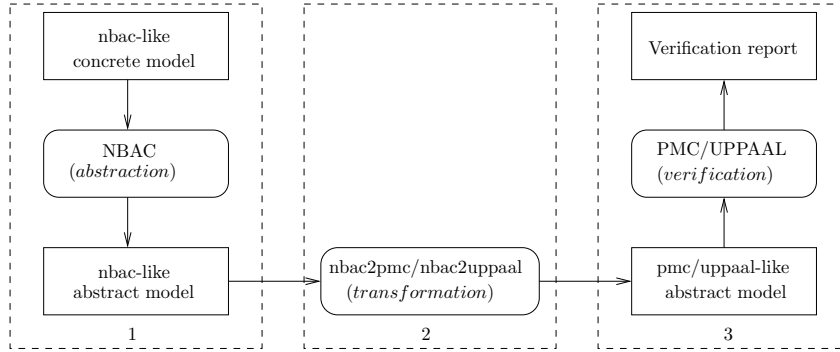


Figure 6.1: A systematic approach to combine Nbac and PMC/Uppaal

part is necessary if we want to go further with complex systems. Anyhow, such a tool has not been developed yet at this moment.

To enable `PMC/Uppaal` to verify abstract models generated by `Nbac`, a transformation step is necessary. This step is responsible for transforming an `Nbac` abstract model to an `XTG` model. Since the size of an `Nbac` abstract model is usually extremely large, this step must be done automatically by a computer program. We have developed such a tool for that purpose. Two versions of that tool are available: one for `Nbac` and `PMC` (`nbac2xtgxml`) and the other for `Nbac` and `Uppaal` (`nbac2uppaal`). We presented its design and implementation in Chapter 3.

A few modifications have to be made when an `Nbac` abstract model is transformed to an `XTG` model. We firstly established a framework for the transformation and then implemented the tool following that transformation framework. The lack of some features in `PMC` made a resulting model too large for the reason that a number of elements have been added to the original model just to model missing features in `PMC`. However, it turned out that we still could not use `PMC` in our approach even after so much efforts had been made. Consequently, we decided to switch to `Uppaal` and thus the version `nbac2xtgxml` is obsolete now.

After all components in our approach have been available, we are ready to evaluate the implementation of our approach. Four systems were picked up to use as case studies for our experiments including the tiny example, the asynchronous reader/writer algorithm (aka Burn’s algorithm), the Fischer’s protocol, and the subway system. As we described in details in Chapter 4, these systems are very different from each others in original specified language, physical size, the characteristic of the state space and application area.

The experimental results have shown that the tiny example and the

asynchronous reader/writer algorithm can be verified using our approach. Unfortunately, `Uppaal` could not finish the verification on the Fischer’s protocol and Subway system before memory resource was exhausted. These results are unsatisfactory since we aim at the verification on such complex systems.

We now try to answer the first research question of this thesis, which asked: “*What are the practical benefits of combining `Nbac` and `PMC/Uppaal` using such systematic approach ?*”.

Frankly, the practical benefits of using `Nbac` abstract models in verification done by `Uppaal`¹ are not clear, since those abstract models are unverifiable. Although we could successfully verify the tiny example and the asynchronous reader/writer algorithm, no conclusions can be drawn about efficiency of the verification due to the fact that these two systems are quite small and thus the differences in performance between verification using the combination approach and using only `Uppaal` are insignificant.

The unsatisfactory experimental results bring us to the second research question of this thesis, which asked “*How can we drive `Nbac` to generate the right abstract model to achieve the best efficiency in terms of accuracy of verification results and the scalability of systems that can be verified ?*”.

To answer the second question, efforts have been made to determine the reasons for large state space created from `Nbac` abstract models. Remind that an `Nbac` abstract model contains an automaton specifying an execution loop around the *good* location(s) if the system does not violate the safety property. Taking into account that fact, we measured the memory usage to estimate the size of state space and the way the consumption of memory resource grew after every iterations. It turned out that the memory usage increased very quickly when we incremented the number of iterations.

After analyzing a model and its simulation, we believe that the reasons for the state space explosion of model checking using our combination approach are the following:

1. The infinite characteristic of the model. This is caused by the presence of unbounded Integer variables in the model.
2. The way `Uppaal` treats such infinite model in verification. The infinite model makes `Uppaal` search until it reaches the bounded Integer value. In complicated systems like the Subway system, memory resource is exhausted before `Uppaal` can reach that boundary and give a conclusive result.

¹`PMC` was not involved in our experiments

Our method to get out of the problem is to restrict the number of iterations that a system can loop around the good location. However, this method is not a complete solution since it means that only a part of the state space is verified.

In conclusion, experiment results on a few case studies are negative at the time when this thesis is written. It remains the problem of state space explosion and thus we have not reached our goal of efficiently obtaining a reasonable combination of two tools. Anyhow, our work sets up a foundation for the evaluation of the possibility of combining `Nbac` and `PMC/Uppaal` and for further improvements to solve the problem of combining these tools. This work also helps us to gain more insight on approaches implemented in these tools and the general problem of model checking. We found that abstract models generated by `Nbac` are not very well-fitted to be verified by `Uppaal`. An `Nbac` abstract model helps `Nbac` to accelerate the verification but not `Uppaal` for the reason that the `Nbac` abstract model is defined to work with the verification approach in `Nbac` only. We believe that this problem is dependent on specific tools employed in our combination approach.

Related work. There have been studies to employ abstraction techniques to model checking to overcome the state space explosion problem, for example [14, 17]. The main difference is that the abstraction technique is “embedded” into the model checking algorithm in those works. That means the data structures and algorithms are tailored for the abstraction technique used. Our approach works at a higher level, between existing tools and models of the system being verified. On the one hand, integrating abstraction technique into the model checking algorithm gives us more control and flexibility in the development of a specific model checker. On the other hand, combining existing tools provides the possibility to reuse the solution and part of the implementation work if we are successful with the approach. However, to our knowledge, there has been no successful story of this direction reported until now.

A successful work to develop an independent abstraction tool for explicit model checking approach in `Spin` is reported in [15]. This work leads to the idea of developing a general abstraction tool intentionally for a class of symbolic model checking approach in real-time model checkers such as `PMC`, `Uppaal`, `Hytech`, etc. Such an abstraction tool will be more useful for these model checkers than `Nbac` since `Nbac` is not developed to serve as a general abstraction tool for them.

Future work. This thesis work started from the combination between `Nbac` and `PMC` but ended with the combination between `Nbac` and `Uppaal`. We conclude that it is impossible to proceed the work of combining `Nbac` and `PMC` with the current status of `PMC`. However, it should be possible to

integrate **Nbac** approach and **PMC** approach by starting from scratch, meaning that we have to define the abstraction function suitable for TCTL property based on the theory of abstract interpretation. Then the model checking algorithm based on partition refinement might be modified to work properly. This direction requires a thorough study on the semantics of an abstract model of a real-time system.

The problem we are facing in the combination between **Nbac** and **Uppaal** is that **Uppaal** cannot verify an infinite system that involves unbounded variables except clocks. A typical solution for solving the problem of verifying infinite system is to employ abstraction to reduce the infinite system to a finite one. Besides, a study on the decidability and complexity of verification problems for different classes of infinite automata is recommended to solve this issue.

As previously mentioned, **Nbac** is not very suitable to serve as a general abstraction tool for real-time model checkers. We have to model real-time notion by physical clock tick when using **Nbac** to generate abstract models. In addition, models generated by **Nbac** are often too large since **Nbac** unfolds **XTG** models based on its mechanism. This leads to the fact that if a model is too complicated, even a mature model checker like **Uppaal** may fail to initialize the data structure. We believe that integrating the two approaches directly could be not only less complicated but also more flexible for the verification in this case. For all those reasons, we are not highly motivated to spend more efforts to pursue a workable combined tool between **Nbac** and **Uppaal**.

Bibliography

- [1] Hytech documentation. <http://www-cad.eecs.berkeley.edu/tah/hytech/>.
- [2] The language Lustre. <http://www-verimag.imag.fr/synchrone/lustre-english.html>.
- [3] SMV documentation. <http://www-2.cs.cmu.edu/modelcheck/smv.html>.
- [4] Spin documentation. <http://spinroot.com/spin/whatispin.html>.
- [5] UPPAAL documentation. <http://www.docs.uu.se/docs/rtmv/uppaal/>.
- [6] R. Alur and D. Dill. The theory of timed automata. *Lecture Notes in Computer Science*, 600:45–??, 1992.
- [7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
- [8] J. Bengtsson, K. G. Larsen, F. Larsson, and P. Pettersson. UPPAAL: a tool suite for automatic verification of real-time systems. *Lecture Notes in Computer Science*, 1066, 1996.
- [9] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 319–331. Springer-Verlag, 1998.
- [10] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Trans. Program. Lang. Syst.*, 21(4):747–789, 1999.
- [11] Alan Burns and Jing Chen. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, IEEE press, pages 2–9, 1998.

- [12] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–354. ACM SIGACT and SIGPLAN, ACM Press, 1992.
- [13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM SIGACT and SIGPLAN, ACM Press, 1977.
- [14] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
- [15] Maria del Mar Gallardo, Jesus Martinez, Pedro Merino, and Ernesto Pimentel. A tool for abstraction in model checking. In Rance Cleaveland and Hubert Garavel, editors, *Electronic Notes in Theoretical Computer Science*, volume 66. Elsevier, 2002.
- [16] D.H.Nguyen. Improving efficiency of model checking through systematically combining Nbac and PMC. Technical report, TU Delft, 2004.
- [17] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. *Lecture Notes in Computer Science*, 939:409–??, 1995.
- [18] Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 54–69. Springer-Verlag, 1995.
- [19] B. Berard et al. *Systems and software verification : model-checking techniques and tools*. Springer, 2001.
- [20] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83. Springer-Verlag, 1997.
- [21] B. Jeannet. The NBAC verification / slicing tool. <http://www.irisa.fr/prive/bjeannet/nbac/nbac.html>.
- [22] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of synchronous programs. *Research Report RS-00-38, BRICS*, December 2000.

- [23] R.F. Lutje Spelberg. *Model checking Real-Time Systems based on Partition Refinement*. Phd thesis, Delft University of Technology, 2004.

Appendix A

Nbac input systems

A.1 Tiny example

```
state
    init0,ok: bool;
    b0,b1:bool;
    x,y:int;

input
    p1,p2:bool;

transition
    init0' = false;
    ok' = if init0 then true else ok and x>=y;
    b0' = if init0 then false else not b1;
    b1' = if init0 then false else b0;
    x' = if init0 then 0 else if (b0 = b1) then x+1 else x;
    y' = if init0 then 0 else if (b0 <> b1) then y+1 else y;

assertion
    p1 and not p2 or not p1 and p2;

initial init0;

final not init0 and not ok;
```

A.2 Asynchronous reader/writer algorithm

```
state
    s,l2,l3,l4,l5,l6,l7,lr : bool;
    ws,wl2,wl3,wl4,wl5,wl6,wl7,wl8,
    wl9,wl10,wl11,wl12 : bool;
```

```

L1, L2 : int;
r1, Latest,ridx, Prefer1, Prefer2, Precon : int;
j,w1,w2,widx : int;

local
  ls2, l23,l34,l45,l56,l57,l6r,l7r,lrs : bool;
  wls2, wl23,wl34,wl45,wl56,wl57,wl68,wl78,
  wl810, wls9, wl910, wl1011, c1, c2, c3, wl12s : bool;

definition
  ls2 = true; l23 = true; l34 = true; l45 = true; l6r = true;
  l7r = true; lrs = true; l56 = L1=0; l57 = L1=1;

  wls2 = L2=0; wl23 = true; wl34 = true; wl45 = true;
  wl56 = L1=0; wl57 = L1=1; wl68 = true; wl78 = true;
  wl810 = true; wls9 = L2=1; wl910 = true; wl1011 = true;

  c1 = (w2 =3 and j=2) or (w2=3 and j=3) or (w2=2 and j=3);
  c2 = (w2 =1 and j=1) or (w2=3 and j=1) or (w2=1 and j=3);
  c3 = (w2 =2 and j=2) or (w2=1 and j=2) or (w2=2 and j=1);
  wl12s = true;

transition
  s' = if lr then true
        else if s then not l2
        else false ;

  l2' = if s then ls2
        else if l2 then not l23
        else false;

  l3' = if l2 then l23
        else if l3 then not l34
        else false;

  l4' = if l3 then l34
        else if l4 then not l45
        else false;

  l5' = if l4 then l45
        else if l5 then not ( l57 or l56 )
        else false;

  l6' = if l5 then l56
        else if l6 then not l6r
        else false;

  l7' = if l5 then l57
        else if l7 then not l7r

```

```

else false;

lr'= if l6 then l6r
      else if l7 then l7r
      else if lr then not lrs
      else false;

ws'= if w112 then w112s
      else if ws then not (wls2 or wls9)
      else false;

w12' = if ws then wls2
        else if w12 then not w123
        else false;

w13' = if w12 then w123
        else if w13 then not w134
        else false;

w14'= if w13 then w134
        else if w14 then not w145
        else false;

w15' = if w14 then w145
        else if w15 then not (w156 or w157)
        else false;

w16' = if w15 then w156
        else if w16 then not w168
        else false;

w17' = if w15 then w157
        else if w17 then not w178
        else false;

w18' = if w17 then w178
        else if w16 then w168
        else if w18 then not w1810
        else false;

w19'= if ws then wls9
        else if w19 then not w1910
        else false;

w110'= if w19 then w1910
        else if w18 then w1810
        else if w110 then not w11011

```

```

else false;

wl11'= if wl10 then wl1011
      else if wl11 then not (c1 or c2 or c3)
      else false;

wl12'= if wl11 then ( c1 or c2 or c3 )
      else if wl12 then not wl12s
      else false;

L1'= if s and ls2 then 0
     else if (l5 and l56) then 0
     else if l6 and l56 then 1
     else if l5 and l57 then 1

     else if (wl5 and wl56) then 0
     else if wl6 and wl56 then 1
     else if (wl5 and wl57) then 1
     else L1;

L2'= if l2 and l23 then 0
     else if ws and wls2 then 0
     else if ws and wls9 then 1
     else if wl2 and wl23 then 1
     else L2;

r1'= if true then 1
     else if l3 and l34 then Latest
     else r1;

Prefer2'= if true then 1
         else if l4 and l45 then r1
         else Prefer2 ;

Prefer1' = if true then 1
         else if wl4 and wl45 then w1
         else Prefer1;

Latest'= if true then 1
         else if wl3 and wl34 then r1
         else if wl12 and wl12s then widx
         else Latest;

ridx'= if true then 1
       else if l6 and l6r then Prefer2
       else if l7 and l7r then Prefer1
       else ridx;

widx' = if true then 1

```

```

        else if w111 and c1 then 1
        else if w111 and c2 then 2
        else if w111 and c3 then 3
    else widx;

Precon' = if true then 1
        else if w18 and w1810 then w2
        else Precon;

w1' = if true then 1
      else if w13 and w134 then Latest
      else w1;

w2' = if true then 1
      else if w17 and w178 then Prefer2
      else if w16 and w168 then Prefer1
      else if w19 and w1910 then Precon
      else w2;

j' = if true then 1
     else if w110 and w11011 then Latest
     else j;

initial
    (s and not l2 and not l3 and not l4 and
     not l5 and not l6 and not l7 and not lr) and

    (ws and not w12 and not w13 and not w14 and not w15
     and not w16 and not w17 and not w18 and not w19
     and not w110 and not w111 and not w112) ;

final
    (lr and w112) and ((l7 or l6)and w111) ;

```

A.3 Fischer's protocol

```

state
    l10,l11,l12,l13,l20,l21,l22,l23,lock1,lock2:bool;
    c1,c2:int;

input
    p1,p2,tick:bool;

```

```

transition
  l10' = if p1
        then if l12 then not lock1
              else l13
        else l10;

  l11' = if p1 then l10
        else l11;

  l12' = if p1 then l11
        else l12;

  l13' = if p1 then l12 and lock1 and c1>2
        else l13;

  l20' = if p2
        then if l22 then not lock2
              else l23
        else l20;

  l21' = if p2 then l20
        else l21;

  l22' = if p2 then l21
        else l22;

  l23' = if p2 then l22 and lock2 and c2>2
        else l23;

  lock1' = if p1 then
           if l11 then true
             else if l13 then false
                 else lock1
           else if p2 then
             if l21 or l23 then false
               else lock1
           else lock1;

  lock2' = if p2 then
           if l21 then true
             else if l23 then false
                 else lock2
           else if p1 then
             if l11 or l13 then false
               else lock2
           else lock2;

  c1' = if tick then c1+1

```

```

        else if p1 and (l10 or l11) then 0
        else c1;

c2' = if tick then c2+1
      else if p2 and (l20 or l21) then 0
      else c2;

assertion
  (if (l11 and not c1<2) or (l21 and not c2<2) then
  not tick else true) and not (p1 and p2)
  and (if tick then not p1 and not p2 else
       if p1 then not tick and not p2 else
       if p2 then not tick and not p1 else true);

initial
  l10 and not l11 and not l12 and not l13 and
  l20 and not l21 and not l22 and not l23 and
  not lock1 and not lock2;

final
  l13 and l23;

```

A.4 Subway system

```

state
  Init0 : bool;
  b1, PV215_avance_0, PV217_retard_0, b0, PV216_avance_1, PV218_retard_1
: bool;
  P_V329_c, P_V330_c, P_V212_nB_1, P_V61_nS, P_V211_nB_0 : int ;

input
  B_1, S, B_0 : bool;

local
  ok : bool;
  V215_avance_0, V217_retard_0, V327_H_0, V216_avance_1, V218_retard_1,
V328_H_1 : bool;
  V329_c, V213_diff_0, V330_c, V212_nB_1, V214_diff_1, V61_nS, V211_nB_0
: int ;

definition
  V215_avance_0 = ((not Init0) and (if (not PV215_avance_0) then (V213_diff_0
  >= 10) else (V213_diff_0 >= 1)))));
  V217_retard_0 = ((not Init0) and (if (not PV217_retard_0) then (V213_diff_0
  <= (-10)) else (V213_diff_0 <= (-1)))));
  V327_H_0 = (Init0 or (((not PV217_retard_0) or (not S)) and ((not (P_V329_c

```

```

>= 9)) or (not B_0)))));
  V216_avance_1 = ((not Init0) and (if (not PV216_avance_1) then (V214_diff_1
>= 10) else (V214_diff_1 >= 1)))));
  V218_retard_1 = ((not Init0) and (if (not PV218_retard_1) then (V214_diff_1
<= (-10)) else (V214_diff_1 <= (-1)))));
  V328_H_1 = (Init0 or (((not PV218_retard_1) or (not S)) and ((not (P_V330_c
>= 9)) or (not B_1)))));
  V329_c = (if Init0 then 0 else (if (PV215_avance_0 and b1) then (if B_0
then (P_V329_c + 1) else P_V329_c) else 0));
  V213_diff_0 = (V211_nB_0 - V61_nS);
  V330_c = (if Init0 then 0 else (if (PV216_avance_1 and b0) then (if B_1
then (P_V330_c + 1) else P_V330_c) else 0));
  V212_nB_1 = (if Init0 then 0 else (if B_1 then (P_V212_nB_1 + 1) else
P_V212_nB_1));
  V214_diff_1 = (V212_nB_1 - V61_nS);
  V61_nS = (if Init0 then 0 else (if S then (P_V61_nS + 1) else P_V61_nS));
  V211_nB_0 = (if Init0 then 0 else (if B_0 then (P_V211_nB_0 + 1) else
P_V211_nB_0));
  ok = (Init0 or ((-10) <= (P_V211_nB_0 - P_V61_nS)));

```

transition

```

Init0' =not true;
b1' =((not Init0) and PV215_avance_0);
PV215_avance_0' =V215_avance_0;
PV217_retard_0' =V217_retard_0;
b0' =((not Init0) and PV216_avance_1);
PV216_avance_1' =V216_avance_1;
PV218_retard_1' =V218_retard_1;
P_V329_c' =V329_c;
P_V330_c' =V330_c;
P_V212_nB_1' =V212_nB_1;
P_V61_nS' =V61_nS;
P_V211_nB_0' =V211_nB_0;

```

initial

```
Init0;
```

assertion

```
(V328_H_1 and V327_H_0);
```

invariant ok;

Appendix B

Nbac output systems

B.1 Tiny example

```
state
  ok, init, b1, b0 : bool;
  y, x : int;

input
  p2, p1 : bool;

transition
  y' = if (init) and true
    then 0
    else if (not init) and true
      and ((not b1 and not b0) or (b1 and b0))
    then y
    else if (not init) and true
      and ((b1 and not b0) or (not b1 and b0))
    then y+1
    else 0;
  x' = if (init) and true
    then 0
    else if (not init) and true
      and ((not b1 and not b0) or (b1 and b0))
    then x+1
    else if (not init) and true
      and ((b1 and not b0) or (not b1 and b0))
    then x
    else 0;
  b1' = (not init and b0) and true;
  b0' = (not init and not b1) and true;
```

```

ok' = true and true
      and ((not init and not y-x-1>=0) or (init));
init' = true and true
      and (false);
assertion true and true
      and ((not ok and init and not p2 and p1) or
            (not ok and init and p2 and not p1) or
            (ok and not p2 and p1) or (ok and p2 and not p1));
initial (init) and true;
final (not ok and not init) and true;
automaton
  location bad_0 : (not ok and not init) and (y+x-2>=0 and -y+x+2>=0 and
                                                    y-x>=0);

  location init_0 : (init) and true;
  location good_0 : (ok and not init) and (y>=0 and x>=0 and -y+x+1>=0);
  edge (init_0,good_0) : true and true
      and ((not p2 and p1) or (p2 and not p1));
  edge (good_0,bad_0) : true and (y-x-1>=0)
      and ((not p2 and p1) or (p2 and not p1));
  edge (good_0,good_0) : true and (not y-x-1>=0)
      and ((not p2 and p1) or (p2 and not p1));

```

B.2 Subway system

```

state
  Init0, PV40_pok, PV214_avance_0, PV216_retard_0, PV217_retard_1,
  PV215_avance_1, b0, b1 : bool;
  P_V210_nB_0, P_V36_nS, P_V211_nB_1, P_V329_c, P_V328_c : int;

input
  B_0, S, B_1 : bool;

transition
  P_V210_nB_0' = if (Init0) and true
      then 0
      else if (not Init0 and B_0) and true
      then P_V210_nB_0+1
      else if (not Init0 and not B_0) and true
      then P_V210_nB_0
      else 0;
  P_V36_nS' = if (Init0) and true
      then 0
      else if (not Init0 and S) and true
      then P_V36_nS+1
      else if (not Init0 and not S) and true
      then P_V36_nS

```

```

else 0;
P_V211_nB_1' = if (Init0) and true
then 0
else if (not Init0 and B_1) and true
then P_V211_nB_1+1
else if (not Init0 and not B_1) and true
then P_V211_nB_1
else 0;
P_V329_c' = if true and true
and ((not b0) or
(not Init0 and not PV215_avance_1 and b0) or
(Init0 and b0))
then 0
else if (not Init0 and PV215_avance_1 and b0 and B_1) and true
then P_V329_c+1
else if (not Init0 and PV215_avance_1 and b0 and not B_1) and true
then P_V329_c
else 0;
P_V328_c' = if true and true
and ((not Init0 and not PV214_avance_0) or
(not Init0 and PV214_avance_0 and not b1) or (Init0))
then 0
else if (not Init0 and PV214_avance_0 and b1 and B_0) and true
then P_V328_c+1
else if (not Init0 and PV214_avance_0 and b1 and not B_0) and true
then P_V328_c
else 0;
PV40_pok' = true and true
and ((not Init0 and not PV214_avance_0 and not PV216_retard_0) or
(not Init0 and not PV214_avance_0 and PV216_retard_0 and
not B_0 and not P_V210_nB_0-P_V36_nS-10>=0) or
(not Init0 and not PV214_avance_0 and PV216_retard_0 and
B_0 and not P_V210_nB_0-P_V36_nS-9>=0) or
(not Init0 and PV214_avance_0 and not PV216_retard_0 and
not B_0 and not S and P_V210_nB_0-P_V36_nS+9>=0) or
(not Init0 and PV214_avance_0 and not PV216_retard_0 and
not B_0 and S and P_V210_nB_0-P_V36_nS+8>=0) or
(not Init0 and PV214_avance_0 and not PV216_retard_0 and
B_0 and not S and P_V210_nB_0-P_V36_nS+10>=0) or
(not Init0 and PV214_avance_0 and not PV216_retard_0 and
B_0 and S and P_V210_nB_0-P_V36_nS+9>=0) or
(not Init0 and PV214_avance_0 and PV216_retard_0 and
not B_0 and P_V210_nB_0-P_V36_nS>=0 and
not P_V210_nB_0-P_V36_nS-1>=0) or
(not Init0 and PV214_avance_0 and PV216_retard_0 and
B_0 and not P_V210_nB_0-P_V36_nS>=0 and
P_V210_nB_0-P_V36_nS+1>=0) or (Init0));
PV214_avance_0' = (not Init0) and true
and ((not PV214_avance_0 and not B_0 and not S and

```

```

P_V210_nB_0-P_V36_nS-10>=0) or
(not PV214_avance_0 and not B_0 and S and
P_V210_nB_0-P_V36_nS-11>=0) or
(not PV214_avance_0 and B_0 and not S and
P_V210_nB_0-P_V36_nS-9>=0) or
(not PV214_avance_0 and B_0 and S and
P_V210_nB_0-P_V36_nS-10>=0) or
(PV214_avance_0 and not B_0 and not S and
P_V210_nB_0-P_V36_nS-1>=0) or
(PV214_avance_0 and not B_0 and S and
P_V210_nB_0-P_V36_nS-2>=0) or
(PV214_avance_0 and B_0 and not S and
P_V210_nB_0-P_V36_nS>=0) or
(PV214_avance_0 and B_0 and S and
P_V210_nB_0-P_V36_nS-1>=0));
PV216_retard_0' = (not Init0) and true
and ((not PV216_retard_0 and not B_0 and not S and
not P_V210_nB_0-P_V36_nS+9>=0) or
(not PV216_retard_0 and not B_0 and S and
not P_V210_nB_0-P_V36_nS+8>=0) or
(not PV216_retard_0 and B_0 and not S and
not P_V210_nB_0-P_V36_nS+10>=0) or
(not PV216_retard_0 and B_0 and S and
not P_V210_nB_0-P_V36_nS+9>=0) or
(PV216_retard_0 and not B_0 and
not P_V210_nB_0-P_V36_nS>=0) or
(PV216_retard_0 and B_0 and
not P_V210_nB_0-P_V36_nS+1>=0));
PV217_retard_1' = (not Init0) and true
and ((not PV217_retard_1 and not S and not B_1 and
P_V36_nS-P_V211_nB_1-10>=0) or
(not PV217_retard_1 and not S and B_1 and
P_V36_nS-P_V211_nB_1-11>=0) or
(not PV217_retard_1 and S and not B_1 and
P_V36_nS-P_V211_nB_1-9>=0) or
(not PV217_retard_1 and S and B_1 and
P_V36_nS-P_V211_nB_1-10>=0) or
(PV217_retard_1 and not B_1 and
P_V36_nS-P_V211_nB_1-1>=0) or
(PV217_retard_1 and B_1 and
P_V36_nS-P_V211_nB_1-2>=0));
PV215_avance_1' = (not Init0) and true
and ((not PV215_avance_1 and not S and not B_1 and
not P_V36_nS-P_V211_nB_1+9>=0) or
(not PV215_avance_1 and not S and B_1 and
not P_V36_nS-P_V211_nB_1+8>=0) or
(not PV215_avance_1 and S and not B_1 and
not P_V36_nS-P_V211_nB_1+10>=0) or
(not PV215_avance_1 and S and B_1 and

```

```

        not P_V36_nS-P_V211_nB_1+9>=0) or
(PV215_avance_1 and not S and not B_1 and
not P_V36_nS-P_V211_nB_1>=0) or
(PV215_avance_1 and not S and B_1 and
not P_V36_nS-P_V211_nB_1-1>=0) or
(PV215_avance_1 and S and not B_1 and
not P_V36_nS-P_V211_nB_1+1>=0) or
(PV215_avance_1 and S and B_1 and
not P_V36_nS-P_V211_nB_1>=0));
b0' = (not Init0 and PV215_avance_1) and true;
b1' = (not Init0 and PV214_avance_0) and true;
Init0' = true and true
      and (false);
assertion true and true
      and ((not Init0 and PV40_pok and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and not B_1) or
(not Init0 and PV40_pok and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and B_1 and not P_V329_c-9>=0) or
(not Init0 and PV40_pok and not PV216_retard_0 and
not PV217_retard_1 and B_0 and not B_1 and not P_V328_c-9>=0) or
(not Init0 and PV40_pok and not PV216_retard_0 and
not PV217_retard_1 and B_0 and B_1 and not P_V329_c-9>=0 and
not P_V328_c-9>=0) or
(not Init0 and PV40_pok and not PV216_retard_0 and
PV217_retard_1 and not B_0 and not S and not B_1) or
(not Init0 and PV40_pok and not PV216_retard_0 and
PV217_retard_1 and not B_0 and not S and B_1 and
not P_V329_c-9>=0) or
(not Init0 and PV40_pok and not PV216_retard_0 and
PV217_retard_1 and B_0 and not S and not B_1 and
not P_V328_c-9>=0) or
(not Init0 and PV40_pok and not PV216_retard_0 and
PV217_retard_1 and B_0 and not S and B_1 and
not P_V329_c-9>=0 and not P_V328_c-9>=0) or
(not Init0 and PV40_pok and PV216_retard_0 and not b1 and
not B_0 and not S and not B_1) or
(not Init0 and PV40_pok and PV216_retard_0 and not b1 and
not B_0 and not S and B_1 and not P_V329_c-9>=0) or
(not Init0 and PV40_pok and PV216_retard_0 and not b1 and
B_0 and not S and not B_1 and not P_V328_c-9>=0) or
(not Init0 and PV40_pok and PV216_retard_0 and not b1 and
B_0 and not S and B_1 and not P_V329_c-9>=0 and
not P_V328_c-9>=0) or (Init0));
initial (Init0) and true;
final (not Init0 and not PV40_pok) and true
      and ((not PV214_avance_0 and PV216_retard_0 and b1) or (PV214_avance_0));
automaton
  location bad_0 : (not Init0 and not PV40_pok) and (P_V329_c>=0 and
P_V210_nB_0>=0 and

```

```

P_V210_nB_0+P_V36_nS-10>=0 and
P_V328_c>=0 and
P_V36_nS>=0 and
P_V211_nB_1>=0)
and ((not PV214_avance_0 and PV216_retard_0 and
not PV217_retard_1 and b1) or
(not PV214_avance_0 and PV216_retard_0 and
PV217_retard_1 and not PV215_avance_1 and b1) or
(PV214_avance_0 and not PV216_retard_0 and not b1));
location init_0 : (Init0) and true;
location good_0 : (not Init0 and PV40_pok) and (P_V329_c>=0 and
P_V210_nB_0>=0 and
P_V328_c>=0 and
P_V36_nS>=0 and
P_V211_nB_1>=0)
and ((not PV214_avance_0 and not PV216_retard_0) or
(not PV214_avance_0 and PV216_retard_0 and not b1) or
(PV214_avance_0 and not PV216_retard_0));
edge (init_0,good_0) : true and true;
edge (good_0,bad_0) : true and true
and ((not PV214_avance_0 and PV216_retard_0 and
not b1 and not B_0 and not S and not B_1 and
P_V210_nB_0-P_V36_nS-10>=0) or
(not PV214_avance_0 and PV216_retard_0 and
not b1 and not B_0 and not S and B_1 and
P_V210_nB_0-P_V36_nS-10>=0 and
not P_V329_c-9>=0) or
(not PV214_avance_0 and PV216_retard_0 and
not b1 and B_0 and not S and not B_1 and
P_V210_nB_0-P_V36_nS-9>=0 and not P_V328_c-9>=0) or
(not PV214_avance_0 and PV216_retard_0 and
not b1 and B_0 and not S and B_1 and
P_V210_nB_0-P_V36_nS-9>=0 and
not P_V329_c-9>=0 and not P_V328_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and not S and
not B_1 and not P_V210_nB_0-P_V36_nS+9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and not S and
B_1 and not P_V210_nB_0-P_V36_nS+9>=0 and
not P_V329_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and S and
not B_1 and not P_V210_nB_0-P_V36_nS+8>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and S and
B_1 and not P_V210_nB_0-P_V36_nS+8>=0 and
not P_V329_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and

```

```

not PV217_retard_1 and B_0 and not S and
not B_1 and not P_V210_nB_0-P_V36_nS+10>=0 and
not P_V328_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and B_0 and not S and
B_1 and not P_V210_nB_0-P_V36_nS+10>=0 and
not P_V329_c-9>=0 and not P_V328_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and B_0 and S and
not B_1 and not P_V210_nB_0-P_V36_nS+9>=0 and
not P_V328_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and B_0 and S and B_1 and
not P_V210_nB_0-P_V36_nS+9>=0 and
not P_V329_c-9>=0 and not P_V328_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
PV217_retard_1 and not B_0 and not S and
not B_1 and not P_V210_nB_0-P_V36_nS+9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
PV217_retard_1 and not B_0 and not S and
B_1 and not P_V210_nB_0-P_V36_nS+9>=0 and
not P_V329_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
PV217_retard_1 and B_0 and not S and
not B_1 and not P_V210_nB_0-P_V36_nS+10>=0 and
not P_V328_c-9>=0) or
(PV214_avance_0 and not PV216_retard_0 and
PV217_retard_1 and B_0 and not S and B_1 and
not P_V210_nB_0-P_V36_nS+10>=0 and
not P_V329_c-9>=0 and not P_V328_c-9>=0));
edge (good_0,good_0) : true and true
and ((not PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and not B_1) or
(not PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and not B_0 and B_1 and
not P_V329_c-9>=0) or
(not PV214_avance_0 and not PV216_retard_0 and
not PV217_retard_1 and B_0 and not B_1 and
not P_V328_c-9>=0) or...

```

(Too long to be completely printed)

Appendix C

Original models

C.1 Fischer's protocol in PMC specification language

```
system fisher
constants u 1
properties EF (f1@CS1 and f2@CS1)
state integer k:=0
processes P1 f1;
          P2 f2
composition f1 | f2
graph P1
state
clock x:=0
init A1
locations
  A1
  { when k==0
    do x:=0
    goto B1
  }
  B1 inv x<=1
  { when true
    do x:=0; k:=1
    goto C1
  }
  C1
  { when x>=u and k==1
```

```

        goto CS1
    when k!=1
        goto A1
    }
CS1
{ when true
  do k:=0
  goto A1
}

graph P2
state
clock x:=0
init A1
locations
  A1
  { when k==0
    do x:=0
    goto B1
  }
  B1 inv x<=1
  { when true
    do x:=0; k:=2
    goto C1
  }
  C1
  { when x>=2 and k==2
    goto CS1
    when k!=2
    goto A1
  }
  CS1
  { when true
    do k:=0
    goto A1
  }
}

```

C.2 Subway system in Lustre specification language

```

--const N=3; -- The number of trains
const N=3;
-- Train controller
node controleur(nB,nS:int) returns (diff:int; avance,retard:bool);
let
  diff = nB-nS;
  avance = false ->

```

```

if not (pre avance) then diff >= 10 else diff >= 1;
  retard = false ->
if not (pre retard) then (diff <= -10) else (diff <= (-1));
tel

-- Hypothesis about the environment
node hypothese(B,S,avance,retard:bool) returns (ok:bool);
var c:int;
let
  ok = true ->
  (if pre retard then not S else true) and
  (if pre c >= 9 then not B else true);
  c = 0 ->
if pre avance and pre (false -> pre avance) then
if B then pre c + 1 else pre c
else
0;
tel

-- Main node (without yet properties)
node main(B : bool^N; S : bool)
returns (ast: bool; nB : int^N; nS : int; diff:int^N; avance,retard : bool^N);
var H:bool^N;
let
  nB = 0^N -> if B then pre nB + 1^N else pre nB;
  nS = 0 -> if S then pre nS + 1 else pre nS;
  H = hypothese(B,S^N,avance,retard);
  (diff,avance,retard) = controleur(nB,nS^N);
  ast = AND(N,H);
tel

-- *****
-- Properties
-- *****

node prop1(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;
  pok:bool;
let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  pok = true -> not (pre avance[0] and retard[0] or pre retard[0] and avance[0]);
  ok = true -> pre pok;
  assert ast;
tel

node prop2(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;

```

```

let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  ok = true -> -10 <= pre diff[0] and pre(diff[0]) <= 19;
  assert ast;
tel

node prop2a(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;
let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  ok = true -> -10 <= pre(nB[0]) - pre(nS);
  assert ast;
tel

node prop2b(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;
let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  ok = true -> pre(diff[0]) <= 20;
  assert ast;
tel

node prop3(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;
let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  ok = true -> pre(nB[1]) - pre(nB[0]) <= 40;
  assert ast;
tel

node prop4(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;
let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  ok = true -> (pre(nB[0]) - pre (nB[1])) <= 29;
  assert ast;
tel

node prop5(B : bool^N; S : bool)
returns (ok:bool;ast:bool);
var nB : int^N; nS : int; diff:int^N; avance,retard : bool^N;
let
  (ast,nB,nS,diff,avance,retard) = main(B,S);
  ok = true -> ((pre(nB[0]) - pre (nB[1])) <= 30) and ((pre(nB[1]) - pre (nB[0])) <= 30);
  assert ast;

```

```
tel

node AND(const n:int; vect:bool^n) returns (AND:bool);
let
AND = with n=1 then
    vect[0]
else
    vect[n-1] and (AND(n-1,vect[0..n-2]));
tel;
```

Appendix D

PMC XML model

D.1 XML representation of a simple PMC model

```
<?xml version="1.0" encoding="UTF-8"?>
<Diagram>
  <Name>example</Name>
  <Properties>
    <Property>
      <EF>
        <And>
          <LeftProperty>
            <Comparison>
              <Greater>
                <LeftValue>
                  <Value type="Name">t.x</Value>
                </LeftValue>
                <RightValue>
                  <Value type="Integer">0</Value>
                </RightValue>
              </Greater>
            </Comparison>
          </LeftProperty>
          <RightProperty>
            <Comparison>
              <Greater>
                <LeftValue>
                  <Value type="Name">t.y</Value>
                </LeftValue>
                <RightValue>
                  <Value type="Integer">0</Value>
                </RightValue>
              </Greater>
            </Comparison>
          </RightProperty>
        </And>
      </EF>
    </Property>
  </Properties>
</Diagram>
```

```

        </Comparison>
      </RightProperty>
    </And>
  </EF>
</Property>
</Properties>
<Behaviour>
  <Name>t</Name>
</Behaviour>
<Processes>
  <Process>
    <Name>t</Name>
    <Chart>Tiny</Chart>
  </Process>
</Processes>
<Charts>
  <XTGChart>
    <Name>Tiny</Name>
    <Variables>
      <Variable>
        <Name>x</Name>
        <Datatype Name="Integer"/>
        <InitialValue>
          <Value type="Integer">0</Value>
        </InitialValue>
      </Variable>
      <Variable>
        <Name>y</Name>
        <Datatype Name="Integer"/>
        <InitialValue>
          <Value type="Integer">0</Value>
        </InitialValue>
      </Variable>
      <Variable>
        <Name>b</Name>
        <Datatype Name="Integer"/>
        <InitialValue>
          <Value type="Integer">0</Value>
        </InitialValue>
      </Variable>
    </Variables>
    <States>
      <State>
        <Name>s0</Name>
        <Invariant>
          <Comparison>
            <Greater>
              <LeftValue>
                <Value type="Name">y</Value>

```

```

        </LeftValue>
        <RightValue>
            <Value type="Integer">0</Value>
        </RightValue>
    </Greater>
</Comparison>
</Invariant>
<Committed>>false</Committed>
</State>
<State>
    <Name>s1</Name>
    <Committed>>false</Committed>
</State>
</States>
<InitialState>s0</InitialState>
<Relations>
    <Relation>
        <InputState>s0</InputState>
        <OutputState>s1</OutputState>
        <BooleanExpression>
            <Comparison>
                <Greater>
                    <LeftValue>
                        <Value type="Name">x</Value>
                    </LeftValue>
                    <RightValue>
                        <Value type="Integer">0</Value>
                    </RightValue>
                </Greater>
            </Comparison>
        </BooleanExpression>
        <ValueAssignment>
            <VariableName>y</VariableName>
            <Value>
                <BinaryPlus>
                    <LeftValue>
                        <Value type="Name">x</Value>
                    </LeftValue>
                    <RightValue>
                        <Value type="Integer">1</Value>
                    </RightValue>
                </BinaryPlus>
            </Value>
        </ValueAssignment>
        <ValueAssignment>
            <VariableName>b</VariableName>
            <Value>
                <Value type="Integer">1</Value>
            </Value>
        </ValueAssignment>
    </Relation>
</Relations>

```

```
        </ValueAssignment>
        <Urgent>>false</Urgent>
    </Relation>
</Relations>
</XTGChart>
</Charts>
</Diagram>
```