Formal Specification and Verification of Safety Interlock Systems: A Comparative Case Study

A THESIS PRESENTED IN PARTIAL FULFILMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

AT THE UNIVERSITY OF STELLENBOSCH

By

Motlatsi Seotsanyana

December 2007

Supervised by: Jaco Geldenhuys

Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ..... Date: .....

Abstract

The ever-increasing reliance of society on computer systems has led to a need for highly reliable systems. There are a number of areas where computer systems perform critical functions and the development of such systems requires a higher level of attention than any other type of system. The appropriate approach in this situation is known as *formal methods*. Formal methods refer to the use of mathematical techniques for the specification, development and verification of software and hardware systems. The two main goals of this thesis are:

1. The design of mathematical models as a basis for the implementation of error-free software for the safety interlock system at iThemba LABS (`http://www.tlabs.ac.za/`).

2. The comparison of formal method techniques that addresses the lack of much-needed empirical studies in the field of *formal methods*.

Mathematical models are developed using model checkers: SPIN, UPPAAL, SMV and a theorem prover PVS. The criteria used for the selection of the tools was based on the popularity of the tools, support of the tools, representation of properties, representativeness of verification techniques, and ease of use.

The procedure for comparing these methods is divided into two phases. Phase one involves the time logging of activities followed by a novice modeler to model check and theorem prove software systems. The results show that it takes more time to learn and use a theorem prover than a model checker. Phase two involves the performance of the tools in relation to the time taken to verify a property, memory used, number of states and transitions generated. In spite of the differences between models, the results are in favor of SMV and this maybe attributed to the nature of the safety interlock system, as it involves a lot of hard-wired lines.

Afrikaans abstract

Die hedendaagse samelewing se steeds-groeiende afhanklikheid op rekenaarstelsels lei tot die behoefte aan hoogsbetroubare sagteware. In verskeie areas verrig rekenaarstelsels kritiese take and die ontwikkeling van sulke stelsels verg 'n hoër vlak van aandag as enige andere. Die mees gepaste benadering vir hierdie geval staan bekend as *formele metodes*, en behels die gebruik van wiskundige tegnieke vir die spesifikasie, ontwerp, en verifikasie van beide sagteware en hardeware. Die twee hoof doelstellings van hierdie tesis is:

1. Die ontwikkeling van wiskundige modelle as 'n basis vir die implementering van foutvrye sagteware vir die *safety interlock system* by iThemba LABS (`http://www.tlbs.ac.za/`)

2. 'n Vergelyking van formele metodes om die ernstige gebrek aan empiriese studies in heirdie veld aan te spreek.

Wiskundige modelle is ontwikkel vir die model toetsers SPIN, UPPAAL, and SMV, en vir die outomatiese stellingbewyser PVS. Die kriteria wat gebruik word vir die vergelyking in die gewildheid van die stelsels, die ondersteuning wat hulle geniet, hul vermoëns om die probleem aan te spreek, hoe verteenwoordigend hulle is, en hul bruikbaarheidgemak.

Acknowledgements

A number of people have helped me to produce this thesis and it is my pleasure to acknowledge them all:

- Firstly, I would like to express my sencire thanks to my supervisor, Jaco Geldenhuys, for his interest, assistance and patience. I feel very lucky to have worked with such a dynamic researcher in the field.
- I am also indebted to Lebelo Serutla with his advice on interpreting the system specification of the case study and productive comments on an earlier draft directly contributed to this study. How can I forget to say many many thanks to his wife Lisema Ramaili, and his two lovely children Paballo and Thabang. Without you I do not know how I would have coped?
- I would also like to thank Jaime Nieto-Camero, Christo van Tubbergh and Cobus Carstens who were always willing to answer my questions regarding the specification of the case study.
- My thanks to the Department of Computer Science at Stellenbosch University and iThemba LABS who have made it possible for me to finish this work.
- I gratefully acknowledge the financial support I received from iThemba LABS.

Finally, many thanks to my family and friends for their loyal support and continuous encouragement. Without you, ...

For my mother — ‘Mabushi Seotsanyana.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ever-increasing reliance of society on computer systems has led to a need for highly reliable software and hardware systems. There are a number of areas where computers perform critical functions ranging from on-line transaction processing systems, such as banking systems and airline reservation systems, to embedded computer systems, such as manufacturing systems, automobiles, air traffic and space vehicle control systems, nuclear power plant safety control systems, medical and military applications. In these areas failure of a computer system may result in just mere inconvenience, economic disruption, loss of time or may even worse cause catastrophic loss of human life. It is clear that the development of such systems requires a higher level of attention than any other type of system and it is also clear that the need for these kind of systems will continue to grow. The appropriate approach in this situation is known as *formal methods*.

Formal methods refer to the use of mathematical techniques for the specification, development and verification of software and hardware systems. A number of success stories about the use of formal methods has been reported in the literature, including [22, 24, 11], but many software practitioners are still skeptical about the use of formal methods in industry [5]. To encourage practitioners to use formal methods, Bowen and Hinchey [13, 14] have proposed ten guidelines in using formal methods in the software development process, while Hall [36], Bowen and Hinchey [12] clarify some myths that people have about formal methods. However, the main problem — which is the focus of this thesis — is the lack of comparative studies undertaken to

compare different formal method techniques [5] such as model checking and theorem proving. Model checking is an automatic verification technique for finite state concurrent systems, while theorem proofing involves the use of deductive methods to develop computer programs in which it can be shown that some statement is a logical consequence of a set of axioms and hypotheses. The system of the case study in this thesis is a safety interlock system at iThemba LABS.

1.1 The iThemba LABS

The iThemba LABS is a multidisciplinary research laboratory which is involved in a number of activities such as basic and applied research using particle beams, particle radiotherapy for the treatment of cancer, and the supply of accelerated-produced radioactive isotopes for nuclear medicine and research. More information about the laboratory can be found at `http://www.tlabs.ac.za/`. At the time of writing of this thesis, iThemba LABS was engaged in a large project referred to as the "Second Beam Line Project" (2BL). This entailed the development of an additional beam line for the treatment of cancer using protons, and formed part of a large system known as the therapy control system (TCS). The components of TCS include a patient positioning system, a supervisory system, a high voltage power supply unit, a dose monitoring system, beam line components, primary and secondary treatment nozzles, a beam analysis and control system, and the therapy safety control system. The full specification can be found in [59].

1.2 Motivation of the research

The two main contributions of this thesis are (1) the design of mathematical models as a basis for the implementation of error-free software for the safety interlock system at iThemba LABS, and (2) a comparison of formal method techniques that addresses the lack of much-needed empirical studies in this field.

The primary purpose of the safety interlock system (SIS) is to enforce safety requirements for the whole TCS. To demonstrate safety, it must be shown that every reachable state of

the system is safe, and ensuring the correctness of the system at the early stages of design is important. Designing an embedded system can be a complex endeavor since it normally consists of a combination of software and (often new) hardware that interact closely. It is useful to take a holistic approach in designing this kind of system and formal methods like model checking and theorem proving can help in both verifying the correctness of and understanding the overall system design, thus improving its reliability. This thesis contains four models of the SIS: three for the model checking tools SPIN, UPPAAL and SMV, and another for the theorem prover PVS. The models differ significantly from each other because the tools differ significantly. They capture different properties of the system or the same property in different ways, they have different levels of ease-of-use, have different interfaces, are based on different logics, etc.

The thesis surveys and compares the above mentioned verification tools. Since the correctness of the SIS is considered important, the comparative analysis is directed towards formal methods that can express both untimed and timed safety properties. The procedure for comparing these methods is divided into two phases. The first phase involves the time logging of activities followed by a novice modeler to model check and theorem prove the models and the second phase involves performance evaluation of the tools with regard to time taken to verify a property, memory usage, number of states and transitions generated during verification of a property.

1.3 Organization of the thesis

Chapter 2: Background and related work presents the theoretical background needed to understand the material in the later chapters. The formal methods covered in this thesis are model checking and theorem proving; others, for example, formal description techniques and program refinement, are not addressed. Three types of model checking are used, namely, linear temporal logic (LTL) model checking, computational tree logic (CTL) model checking, and timed computational tree logic (TCTL) model checking. In the theorem proving category, the thesis considers one higher-order theorem prover called Prototype Verification System (PVS). The chapter ends with a discussion of related work, namely, empirical case studies already

undertaken in formal specification and verification of concurrent reactive systems. Since the main focus of the research is model checking and theorem proving, only those case studies that cover similar ground are considered.

***Chapter 3: Safety Interlock System* (SIS)** presents specification of the system used in the study. This study is a comparative case study and the system that is used to achieve the aim of the research is the safety interlock system at iThemba LABS. Safety interlock system is a real time reactive system which forms part of the whole proton therapy control system (TCS) at iThemba LABS. The detailed description of how the system interacts with other systems is discussed.

Chapter 4: Methodology outlines the approach that is followed to achieve the goals of the study. The justification of the criteria that is followed for the selection of the model checkers and the theorem prover is presented. The graphical representation of a general model the SIS is described, free from influences from any of the tools used in later chapters. This forms the basis for system models that are developed in Chapters 5, 6, 7 and 8.

Chapter 5: The SPIN model checker deals with the design and specification of the SIS in SPIN. SPIN is a model checking tool for verifying the correctness of distributed software such as operating systems, data communication protocols, etc. To make the chapter self-contained, an overview of the PROMELA constructs that are used in the design of the system is given. PROMELA is a specification language for SPIN which is a C-like programming language and is mostly based on Dijkstra's guarded command language.

Chapter 6: The UPPAAL model checker presents the design and specification of the SIS in UPPAAL. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. A timed automaton is a standard finite-state automaton extended with set of real-valued *clock variables* (or just *clocks* in short). The chapter also contains the description of the tool's constructs that are used to design the system.

Chapter 7: The SMV model checker deals with the specification and verification of the SIS in Cadence SMV, just as in Chapter 5 and Chapter 6. Cadence SMV is a verification tool meant for hardware designs, but it is also used in some software systems. The

tool uses Ordered Binary Decision Diagrams (OBDDs) based on a symbolic model checking algorithm [16].

Chapter 8: The PVS theorem prover presents a high-order logic theorem prover called Prototype Verification System (PVS). PVS consists of a specification language, a number of predefined theories, a theorem prover, various utilities, and documentation. In this chapter, it is used to write specifications and constructs proofs about the SIS.

Chapter 9: Discussion of the results compares the tools applied to the design and verification of the SIS. The results are divided into two phases. The first phase outlines the experiences of the modeler, while the second phase discusses the verification results of the tools described in Chapters 5–8.

Chapter 10: Conclusion concludes the thesis by briefly reviewing the main issues addressed in the study, outlining its contribution, and presenting ideas for future work.

Chapter 2

Background

The intention of this chapter is to give a basic overview of the theoretical background and related work as well as harmonising the terminology in formal methods. Further references to more comprehensive studies of the subject are also mentioned. Safety-critical software systems are already an integral part of our everyday lives and their importance is growing rapidly. These systems are inherently large and complex. Avionics, medical systems, automotive systems and railway signalling systems are some examples of such systems whose failure cannot be tolerated. This explains why system validation — that is, the correct design and implementation of such systems — is extremely important. The current practice is that the correctness of such systems is achieved by human inspection: *peer reviews* and *testing* with little or no automation. Peer reviews refer to the inspection of software by a team of engineers that were preferably not involved in the design of the system, while testing refers a process whereby software is executed with some inputs, called *test cases*, along different execution paths known as *runs*.

However, both practices have serious deficiencies when applied to safety-critical systems. In peer review process, it has been shown that it is difficult to catch subtle errors involving concurrency and algorithms [43]. Testing on the other hand is never complete: it is difficult to say when to stop as it is infeasible to check all the runs of a complex system like safety interlock system, and it is easy to omit those runs which may reveal subtle errors. It also has the drawback of only showing the presence of errors, not their absence. The application of

these techniques can be complemented by the use of *formal methods*.

Formal methods refer to the use of mathematical techniques for the specification, development and verification of software and hardware systems. In spite of success stories about the use of formal methods [22, 24, 11], George et al. [5] have shown that some software practitioners are still skeptical about the use of formal methods in industry. Many researchers in the formal methods community have tried very hard to eradicate the perceptions that people have about formal methods. Bowen and Hinchey [13, 14] have come up with ten guidelines in using formal methods in the software development process, while Hall [36], Bowen and Hinchey [12] are encouraging software developers to use formal methods by clarifying myths people have about this approach.

The formal methods discussed in this thesis are model checking and theorem proving. Section 2.1 presents three basic model checking algorithms, that is, Linear Time Logic (LTL), Computational Tree Logic (CTL) and Timed Computational Tree Logic (TCTL) model checking algorithms. Section 2.2 presents the Prototype Verification System (PVS) and Section 2.3 presents related work.

2.1 Model Checking

Model checking is an automatic verification technique for finite state concurrent systems [53] such as safety critical systems, communication protocols, and sequential circuit design. The technique was first introduced around 1980s by two independent groups of researchers, Clarke and Emerson [23], and Queille and Sifakis [52]. Clarke and Emerson [23] are responsible for coining the phrase. Model checking is an attractive alternative to simulation and testing to validate and verify systems. Figure 2.1 depicts the process of model checking. Given a real system (*Example 2.1*) and system specification (*Example 2.2*), a model checker explores the full state-space of an abstract model derived from the real system to check whether or not a given system property is satisfied by the model. The model checker either verifies the given property successfully or generates a counterexample.

Example 2.1: The treatment room at iThemba LABS has two side doors (also called "access

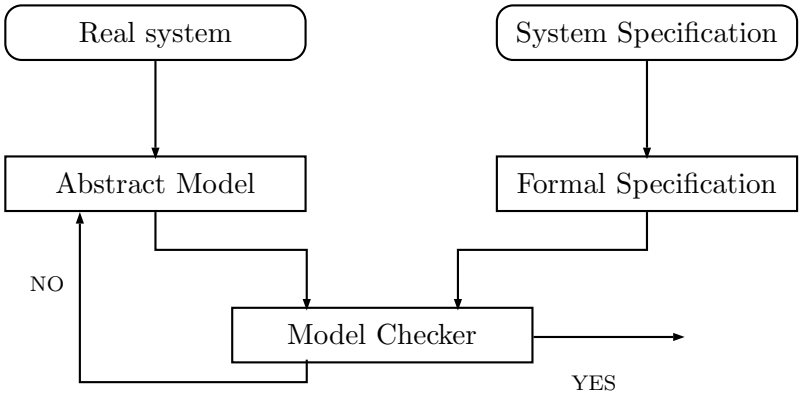


Figure 2.1: Model checking process

doors"), that must be primed and closed within two seconds before the room is evacuated for treatment. This is a function of the room clearance system, which sets a flag to either "true" if the doors are successfully primed and closed or "false" to indicate a failure. One of the properties that can be verified is given in Example 2.2 below.

Example 2.2: It is always the case that if an access door is open, it will eventually be primed.

There are two main advantages of using model checking compared to other formal verification methods. Firstly, it is fully automatic, and secondly, it provides a counterexample whenever the system fails to satisfy a given property. In the process, proper use of abstraction techniques plays an important role as model checking techniques are hindered by the *state-space explosion problem*, where the size of the representation of the behavior of a system grows exponentially with the size of the systems. Often, software systems have infinite state spaces, due to unbounded real and integer input variables and timing constraints, and thus model checking software systems without any abstractions is almost always impossible. In this study only three types of model checking algorithms are considered in Sections 2.1.1, 2.1.2 and 2.1.3.

2.1.1 LTL Model Checking

Logics have been used to precisely describe the properties of concurrent systems. The most widely-used types of logics are *temporal logics*, which were first introduced by Pnueli around 1977 for the specification and verification of computer systems. There are two types of

temporal logics, linear temporal logic (LTL) and computational tree temporal logic (CTL). In this section, the LTL syntax, semantics and basic model checking algorithm are presented and CTL is discussed in section 2.1.2.

LTL Syntax: The syntax of linear temporal logic is defined in terms of atomic propositions, logical connectives and temporal operators. Atomic propositions are the most simple statements that can be made about the system in question and thus take the value *true* or *false*. Examples of atomic propositions are *the door is closed, x is less than 2, etc.* Atomic propositions can be represented by alphabetic symbols such as p and q. The set of atomic propositions is referred to as AP. The boolean operators that are used in the syntax of LTL are $\vee$, $\wedge$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$; in addition, there are several temporal operators, with the following meanings:

- $\Box$ denotes "always",
- $\Diamond$ denotes "eventually",
- U denotes "strong until", and
- X denotes "next".

The structure of a formula of propositional linear temporal logic is given by the following grammar expressed in Backus-Naur Form (BNF) notation:

$$\alpha ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid X\alpha \mid \alpha U \beta$$

The operators $\wedge, \Rightarrow, \Leftrightarrow, true, false, \Diamond$ and $\Box$, which are not mentioned in this syntax, can be thought of merely as abbreviations by using the following rules:

$$\begin{array}{lcllcl}
\alpha \wedge \beta & \equiv & \neg(\neg\alpha \vee \neg\beta) & \alpha \Rightarrow \beta & \equiv & (\neg\alpha \vee \beta) \\
\alpha \Leftrightarrow \beta & \equiv & (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha) & true & \equiv & (\neg\alpha \vee \alpha) \\
false & \equiv & \neg true & \Diamond\alpha & \equiv & true\ U\ \alpha \\
\Box\alpha & \equiv & \neg\Diamond\neg\alpha & & &
\end{array}$$

LTL Semantics: The syntax defines how LTL formulas are constructed, but does not provide an interpretation of the formulas or operators. Formally, LTL formulas are interpreted in terms of a *model* defined as a triple $M = (S, R, Label)$, where

- S is a non-empty countable set of states,
- $R : S \longrightarrow S$, is a function which assigns to each $s \in S$ a unique successor $R(s)$, and
- $Label : S \longrightarrow 2^{AP}$, is a function which assigns to each state $s \in S$ the atomic propositions $Label(s)$ that are valid in s.

The meaning of LTL-formulas are defined in terms of a satisfaction relation, denoted by $\models$, between a model M, a state $s \in S$ and the formulas α and β. Therefore $M, s \models \alpha$ if only if α is valid in the state s of the model M. If it is understood from the context, M is dropped and the satisfaction relation is mathematically defined as follows:

$$
\begin{array}{lll}
s \models p & \text{iff} & p \in Label(s) \\
s \models \neg\alpha & \text{iff} & \neg(s \models \alpha) \\
s \models \alpha \vee \beta & \text{iff} & (s \models \alpha) \vee (s \models \beta) \\
s \models X\alpha & \text{iff} & R(s) \models \alpha \\
s \models \alpha U \beta & \text{iff} & (\exists j \geq 0 : R^j(s) \models \beta) \wedge (\forall 0 \leq k < j : R^k(s) \models \alpha)
\end{array}
$$

Here we have used R^i to denote i applications of the function R. For example, $R^3(s)$ is the same as $R(R(R(s)))$. The formal interpretation of the other connectives, $true, false, \wedge, \Rightarrow, \Diamond$, and $\Box$ can be derived in a similar way from the definitions above.

Finite state automaton

A central component in the model checking of temporal properties is the *finite-state automaton.* A finite state automaton is a model of behaviour composed of states, transitions and actions. Formally, a finite-state automaton M is a tuple $(\Sigma, S, S^0, \rho, F, l)$ where

- Σ is a non-empty set of symbols, that represent atomic propositions,

- S is a finite, non-empty set of states,
- $S^0 \subseteq S$ is a non-empty set of initial states,
- $\rho : S \longrightarrow 2^S$, is a transition relation,
- $F \subseteq S$ is a set of accepting states, and
- $l : S \longrightarrow 2^\Sigma$, the labelling function of states.

A state $s \in S$ stores information (such as the truth values of the atomic propositions) about the system at a specific moment in time. A transition ρ denotes a state change and is an atomic step that makes a system to change a state from one to another. Such transitions sometimes have an enabling condition that needs to be satisfied for the transition to be executable. The $\rho(s)$ is the set of states that the automaton can make a transition into when it is at state s, and we write $s \longrightarrow s'$ if and only if $s' \in \rho(s)$. An *action* is a description of an activity that is performed when a transition executes. A finite-state automaton may either be deterministic or non-deterministic, and it is deterministic if and only if $\|\{s \in S^0 \mid l(s) = a\}\| \leq 1$ for all $a \in \Sigma$, and $\|\{s' \in \rho(s) \mid l(s') = a\}\| \leq 1$ for all $a \in \Sigma$ and all $s \in S$. Figure 2.2(a) depicts an example of deterministic finite-state automaton, whereas Figure 2.2(b) depicts an example of a non-deterministic finite-state automaton.

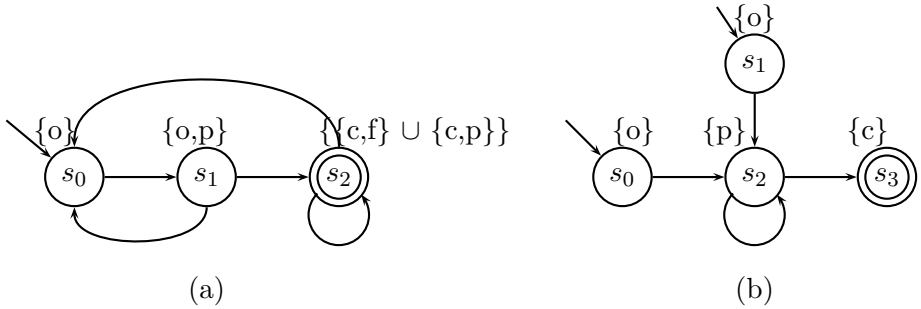


Figure 2.2: Examples of (a) deterministic and (b) non-deterministic finite automata

Example 2.3: The finite-state automaton in Figure 2.2(a) has $\Sigma = \{$*access door is open (o), access door is primed (p), access door is closed (c), flag is true (f)*$\}$, $S = \{s_0,s_1,s_2\}$, $S^0 = \{s_0\}$, $\rho(s_0) = \{s_1\}$, $\rho(s_1) = \{s_0, s_2\}$, $\rho(s_2) = \{s_0, s_2\}$, $F = \{s_2\}$, $l(s_0) = \{o\}$, $l(s_1) = \{o, p\}$, $l(s_2) = \{\{c, f\} \cup \{c, p\}\}$

A *run* of the finite state automaton M is a sequence $\sigma = s_0, s_1, ..., s_n$ such that $s_0 \in S^0$ and $s_i \longrightarrow s_{i+1}$ for all $0 \leq i < n$. A run σ is accepting if and only if $s_n \in F$. The language ($\mathcal{L}$) of the finite state automaton is the (possibly infinite) set of finite words accepted by M, i.e.,

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid w \text{ is accepted by } M\},$$

where Σ^* denotes the set of all finite words over Σ.

Example runs of the finite state automaton depicted in Figure 2.2(b) are s_0, s_2, s_3 and s_0, s_2, s_2, s_3. The accepted words that correspond to these accepting runs are respectively *opc* and *oppc*. The language accepted by this automaton is described by the regular expression op^+c, where p^+ means one or more p's.

Büchi automaton

Model checking of concurrent systems is based on infinite accepting runs of a finite state automaton. However, standard finite state automata cannot describe the continuous behaviour of concurrent systems. Around 1960 Büchi J.R. came up with a special type of finite state automaton called a Büchi automaton. Büchi automata have exactly the same components as finite state automata. However, they differ in how and when runs are accepted. The Büchi-acceptance condition states that if σ^ω is an infinite *run* of the automaton and $inf(\sigma^\omega)$ is the set of states that occur infinitely often in σ^ω, then σ^ω is Büchi-accepting if and only if $inf(\sigma^\omega) \cap F \neq \emptyset$. The set of infinite words accepted by a finite Büchi automaton A is denoted by $\mathcal{L}_\omega(A)$, i.e.,

$$\mathcal{L}_\omega(A) = \{w \in \Sigma^\omega \mid \omega \text{ is accepted by } A\},$$

where Σ^ω denotes the set of infinite words over Σ.

Figure 2.3 depicts examples of Büchi automata. The one Büchi-accepting run of the automaton in Figure 2.3(b) is $s_0, s_1, s_1, s_1, \ldots$ and the corresponding run is $occc\ldots$. This run can be represented by the regular expression oc^ω. Two Büchi automata A_1 and A_2 are equivalent if

and only if $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2)$. Just as before, a Büchi automaton can also either be deterministic or non-deterministic. Unlike standard finite automata, however, non-deterministic Büchi automata are more expressive than deterministic Büchi automata.

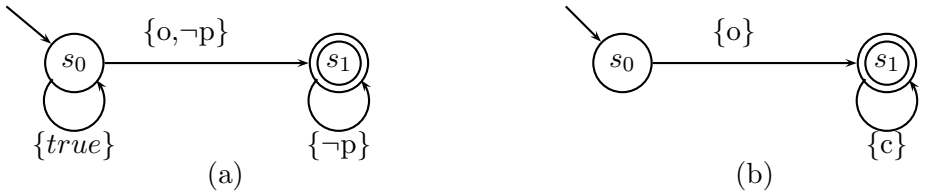


Figure 2.3: Büchi automata

Basic model checking algorithm for LTL

We have now seen all the necessary "mechanics" needed to describe an algorithm for LTL model checking. Around 1983 Wolper, Vardi, and Sistla have shown that for every LTL formula β, there is a corresponding Büchi automaton, denoted by A_β. In order to verify whether a system model M satisfies a given property β, a Büchi automaton M_{sys} is constructed for the model M. The construction of M_{sys} involves adding one initial state which initializes all the atomic propositions and then converting all states to accepting states. Figure 2.4 is a Büchi automaton for the finite-state automaton depicted in Figure 2.2(a).

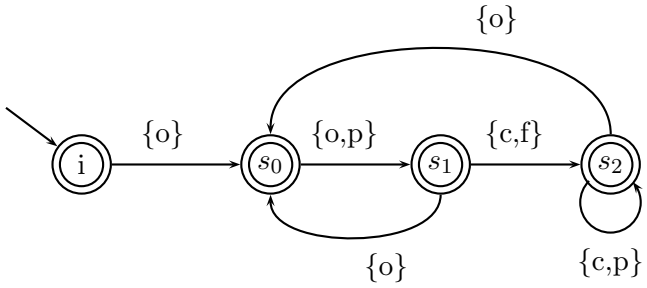


Figure 2.4: Büchi automaton for system model M

The three steps below provide a basic algorithm for verifying that $M \models \beta$:

1. *Construct the Büchi automaton for the LTL-formula β, denoted by A_β.*

2. *Construct the Büch automaton for the model of the system, denoted by* M_{sys}.

3. *Check whether* $\mathcal{L}_\omega(M_{sys}) \subseteq \mathcal{L}_\omega(A_\beta)$.

The accepting runs of M_{sys} correspond to the system model behaviour, while the accepting runs for A_β correspond to the desired behavior of the system model. The third step involves checking the inclusion of the language of $\mathcal{L}_\omega(M_{sys})$ in $\mathcal{L}_\omega(A_\beta)$. We implicitly restrict the alphabet of M_{sys} to be the same as the alphabet of A_β. However, the problem of deciding this inclusion is PSPACE-complete. To overcome this problem we can check whether $\mathcal{L}_\omega(M_{sys}) \cap \mathcal{L}_\omega(\neg A_\beta) = \emptyset$, since

$$\mathcal{L}_\omega(M_{sys}) \subseteq \mathcal{L}_\omega(A_\beta) \iff \mathcal{L}_\omega(M_{sys}) \cap \mathcal{L}_\omega(\neg A_\beta) = \emptyset.$$

Note that $\neg A_\beta$ simulates undesired behavior of the system model and if M_{sys} has an accepting run that is also an accepting run of $\neg A_\beta$, then this means that the system model M does not satisfy the property β. Emerson and Lei [31] have proved that $\mathcal{L}_\omega(M_{sys}) \cap \mathcal{L}_\omega(\neg A_\beta) = \emptyset$ is decidable in linear time. To check whether the intersection is empty, a new automaton, called the synchronous product, is constructed. It is closed under the product operation and it is easy to check that it is empty. However the construction of $\neg A_\beta$ is quadratically exponential, therefore the observation that $\neg A_\beta = A_{\neg\beta}$ is used. In other words, checking whether $\mathcal{L}_\omega(M_{sys}) \cap \mathcal{L}_\omega(A_{\neg\beta}) = \emptyset$ has been reduced to the following two steps:

1. *Construct the product automaton* $M_{sys} \otimes A_{\neg\beta}$ *such that* $\mathcal{L}_\omega(M_{sys} \otimes A_{\neg\beta}) = \mathcal{L}_\omega(M_{sys}) \cap \mathcal{L}_\omega(A_{\neg\beta})$.

2. *Check whether* $\mathcal{L}_\omega(M_{sys} \otimes A_{\neg\beta}) = \emptyset$.

To check the emptiness of the product automaton, it suffices to check whether there is an accepting state reachable from some initial state and also from itself in one or more steps. Put in graph-theoretic terms, $\mathcal{L}_\omega(M_{sys} \otimes A_{\neg\beta})$ is non-empty if and only if there is a cycle that is reachable from an initial state and that contains at least one accepting state. This can be implemented using nested depth-first search as explained in [39].

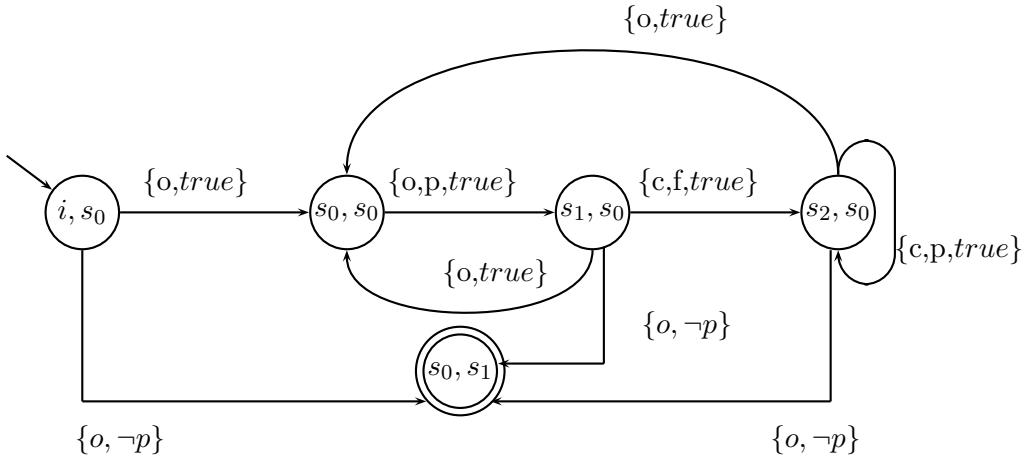


Figure 2.5: Synchronous product finite state automaton

Example 2.4: In Example 2.1 the description of an access-door control system is given and its labelled finite-state automaton, denoted by M, is depicted in Figure 2.2(a). Figure 2.4 depicts a corresponding Büchi automaton, denoted by M_{sys}. Example 2.2 gives a system property, denoted by β. Formally, $\beta = (\Box(o \Rightarrow \Diamond\ p))$ and the corresponding negated Büchi automaton, denoted by $\neg A_\beta$, is depicted in Figure 2.3(a). Figure 2.5 is a synchronous product automaton of the Büchi automata in Figures 2.3(a) and 2.4. Since there is no cycle passing through an accepting state (s_0, s_1), the property that if the door is open, it will eventually be primed is satisfied by the model M — depicted in Figure 2.2(a).

2.1.2 CTL Model Checking

Computational tree logic (CTL) is based on the concept that for each state there are many possible successors, unlike in LTL which is based on a model where each state s has only one successor s'. Because of this branching notion of time, CTL is classified as a *branching temporal logic*. The interpretation of CTL is therefore based on a *tree* rather than a *sequence* as in LTL. In this section, the syntax and semantics of CTL as well as a basic CTL model checking algorithm are discussed.

CTL Syntax: The formulas of CTL consist of atomic propositions, standard boolean connectives of propositional logic, and temporal operators. Each temporal operator is composed

of two parts, a path quantifier (universal $\forall$ or existential $\exists$) followed by a temporal modality ($\Diamond, \Box, X, U$). The temporal modalities have the same meanings as in Section 2.1.1. The syntax is given by the BNF:

$$\alpha ::= p \mid \neg\alpha \mid \alpha \vee \beta \mid \alpha \wedge \beta \mid \exists X\alpha \mid \exists[\alpha U \beta] \mid \forall[\alpha U \beta]$$

CTL Semantics: CTL semantics slightly differs from that one of LTL defined in Section 2.1.1, that is, the notion of a *sequence* is replaced by a notion of a *tree*. The interpretation of CTL is defined by a satisfaction relation $\models$ between a model M, one of its states s and some formula. Let $AP = \{p, q, r\}$ be a set of atomic propositions, $M = (S, R, Label)$ be a CTL-Model, $s \in S$, α and β be CTL-formulas. In order to define the satisfaction relation ($\models$), the following definitions are first given:

- A *path* is an infinite sequence of states $s_0, s_1, s_2...$ such that $(s_i, s_{i+1}) \in R$
- Let $\rho \in S^w$ denotes a path. For $i \geq 0$, $\rho[i]$ denotes the $(i+1)^{th}$ element of ρ, i.e., if $\rho = s_0, s_1, ...$ then $\rho[i] = s_i$
- $P_M(s) = \{\rho \in S^\omega \mid \rho[0] = s\}$ is a set of paths starting at s

Just like in LTL if it is understood from the context, M can be dropped in the satisfaction relation $\models$ defined as follows:

$$\begin{aligned}
s \models p \quad & \text{iff} \quad p \in Label(s) \\
s \models \neg\alpha \quad & \text{iff} \quad \neg(s \models \alpha) \\
s \models \alpha \vee \beta \quad & \text{iff} \quad (s \models \alpha) \vee (s \models \beta) \\
s \models \exists X\alpha \quad & \text{iff} \quad \exists \rho \in P(s) : \rho[1] \models \alpha \\
s \models \exists[\alpha U \beta] \quad & \text{iff} \quad \exists \rho \in P(s) : \exists j \geq 0 : (\rho[j] \models \beta \wedge \forall 0 \leq k < j : \rho[k] \models \alpha) \\
s \models \forall[\alpha U \beta] \quad & \text{iff} \quad \forall \rho \in P(s) : \exists j \geq 0 : (\rho[j] \models \beta \wedge \forall 0 \leq k < j : \rho[k] \models \alpha)
\end{aligned}$$

Basic model checking algorithm for CTL

The idea behind the original model checking algorithm for CTL [21, 23] is to label each state with all the subformulas of the correctness property that holds in the particular state. If a particular state s is labeled with the entire formula that represents the correctness property, then the property is satisfied at that state of the model. Mathematically, subformulas are computed as follows:

$$
\begin{aligned}
Sub(p) &= \{p\} \\
Sub(\neg\alpha) &= Sub(\alpha) \cup \{\neg\alpha\} \\
Sub(\alpha \vee \beta) &= Sub(\alpha) \cup Sub(\beta) \cup \{\alpha, \beta\} \\
Sub(\exists X\alpha) &= Sub(\alpha) \cup \{\exists X\alpha\} \\
Sub(\exists[\alpha U\beta]) &= Sub(\alpha) \cup Sub(\beta) \cup \{\exists[\alpha U\beta]\} \\
Sub(\forall[\alpha U\beta]) &= Sub(\alpha) \cup Sub(\beta) \cup \{\forall[\alpha U\beta]\}
\end{aligned}
$$

The labelling algorithm is inductive, that is, it starts by labelling states with subformulas of length 1 (i.e., atomic propositions) and proceeds to formulas of length $i + 1$ for $1 \leq i <\mid \alpha \mid$.

There are more sophisticated algorithms for verifying CTL properties of models. Two of the most successful approaches is *symbolic model checking* [17, 47] and *bounded model checking* [10]. In very general terms, symbolic model checking computes the set of states that satisfy a given CTL subformula α. It does this by calculating fixpoint expressions that are derived from both the transition relation of the model and the structure of α. Interested readers are referred to the afore-mentioned sources for more information on these interesting techniques. The SMV system, discussed in Chapter 7, makes use of symbolic model checking.

Example 2.5: Using the same example used in Section 2.1.1, the same property is verified, that is, the property that it is always the case that if the door is open, it will eventually be primed. For the model M depicted in 2.2(a) the CTL-formula $\alpha = \forall\Box(o \rightarrow \exists\Diamond p)$ which is equivalent to $\neg\exists\Diamond(o \wedge \exists\Box\neg p)$ is checked. $Sat(o) = \{s_0, s_1\}$, $Sat(\neg p) = \{s_0\}$. In order to compute $Sat(\exists\Box\neg p)$ the set of nontrivial strongly connected components(SCC) are identified, i.e., $SCC = \{\emptyset\}$. The union of the sets in SCC is $T = \emptyset$ are then labelled with $\exists\Box\neg p$, i.e., $Sat(\exists\Box\neg p) = \emptyset$. Then, $Sat(o \wedge \exists\Box\neg p) = Sat(o) \cap Sat(\exists\Box\neg p) = \emptyset$. The set $Sat(\exists\Diamond(o \wedge \exists\Box\neg p))$

is computed by using the converse transition relation, since $Sat(o \wedge \exists\Box\neg p) = \emptyset$*, the set is also empty. Finally,* $Sat(\neg\exists\Diamond(o \wedge \exists\Box\neg p)) = S - Sat(\exists\Diamond(o \wedge \exists\Box\neg p)) = \{s_0, s_1, s_2\}$*. Since the initial state* s_0 *is the element of this set, the property is satisfied by the model.*

2.1.3 TCTL Model Checking

The temporal logics presented in Sections 2.1.1 and 2.1.2 focus on the temporal order of events and do not explicitly state the actual time taken by these events. Time-critical systems necessitate the consideration of quantitative time between the occurrence of events, that is, the correctness of time-critical systems do not only depend on the functional requirements, but also on the time requirements. Typical examples of these systems include communication protocols, radiation control systems, and avionics. In this section, the syntax and semantics of timed computational tree logic (TCTL) are discussed and the basic model checking algorithms for TCTL are also presented.

Timed automata syntax

Finite-state real-time systems are modelled with timed automata. A timed automaton is a standard finite-state automaton extended with set of non-negative real-valued *clock variables* (or just *clocks* in short). Clocks are assumed to proceed at the same rate to measure the time elapsed since they were last reset. In order to formally define a timed automaton, *clocks* and *clock constraints* are first defined as follows:

- A *clock* is variable ranging over $\mathbb{R}^+$ (where $\mathbb{R}^+$ represents non-negative real numbers)
- For set C of clocks with $x, y, z \in C$, a *clock constraint* α over C is defined by

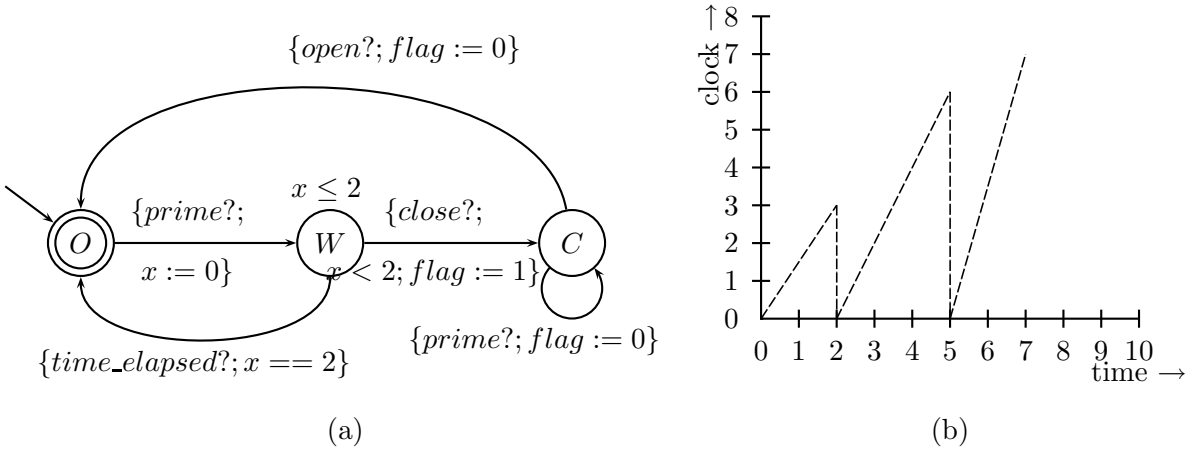
$$\alpha ::= x \prec c \mid x - y \prec c \mid \neg\alpha \mid (\alpha \wedge \alpha), \text{ where } c \in \mathbb{N} \text{ and } \prec \in \{<, \leq\}$$

- $\Psi(C)$ is the set of all possible clock constraints.

Clocks are defined to range over the non-negative real numbers, i.e., $x, y, z \in \mathbb{R}^+$. A state of a timed automaton consists of a *location* and values of clocks. When the system starts, all

variables are initialized to zero and then incremented implicitly at the same rate. The values of the clocks indicate the time elapsed since they have been initialized. Clock constraints are used to label the edges of a timed automaton and represent *guards* that are used to either enable or block transitions between locations. Clock constraints are also used to label locations and such constraints are then *invariants* that limit the amount of time to be spent in a location. Formally, a timed automaton A over set of actions Σ, set of atomic propositions AP and set of clocks C is defined as a tuple $(L, l_0, E, I, Label)$, where:

- L is a non-empty set of locations with the initial location $l_0 \in L$.
- $E \subseteq L$ x $\Psi(C)$ x Σ x 2^C x L corresponds to a set of edges. $(l, g, a, r, l') \in E$ represents an edge from location l to location l' with clock constraint g (also known as enabling condition of the edge or guard) action a to be performed and the set of clocks r to be reset.
- $I : L \rightarrow \Psi(C)$ is a function which assigns a clock constraint (i.e., an invariant) for each location.
- $Label : L \rightarrow 2^{AP}$ is a function which assigns to each locations $l \in L$ set of atomic propositions that hold in the location.



(a) (b)

Figure 2.6: Access door control system

Example 2.6: In Figure 2.6(a), the sets are defined as follows:

- $\Sigma = \{prime?, close?, open?\}$,
- $AP = \{access\ door\ is\ open\ (p),\ access\ door\ is\ primed\ (q),\ access\ door\ is\ closed(r),\ flag\ is\ true\ (f)\}$,
- $C = \{x\}$,
- $L = \{O, W, C\}$,
- $E = \{\ (O, \xrightarrow{true, prime?, x:=0}, W),\ (W, \xrightarrow{x==2, time\_elapsed?}, O),\ (W, \xrightarrow{x<2, close?, flag:=1}, C),\ (C, \xrightarrow{true, prime?, flag:=0}, C),\ (C, \xrightarrow{true, open?, flag:=0}, O)\}$,
- $I(W) = \{x \leq 2\}$, *and*
- $Label(O) = \{p\},\ Label(W) = \{p, q\},\ Label(C) = \{\{r, f\} \cup \{r, q\}\}$.

Here $x == 2$ *is an example of a "guard" and* $x \leq 2$ *is an example of an "invariant". A depiction of the automaton's execution is shown in Figure 2.6(b).*

Timed automaton semantics

The interpretation of a timed automaton is defined in terms of an infinite transition system and in order to formally define the semantics of the timed automaton, the *clock assignment* function and *state* of a timed automaton are defined as follows:

- A *clock valuation (clock assignment)* u for the set of clocks C is a function $u : C \to \mathbb{R}^+$, assigning each clock $x \in C$ its value $u(x)$. Let the set of all clock valuations over C be denoted by $V(C)$. The clock evaluation has the following characteristics:
  - For $u \in V(C)$ and $d \in \mathbb{R}^+$, clock valuation $u + d$ over C means that all clocks are increased by d, that is $(u + d)(x) = u(x) + d$ for all $x \in C$.

  - For $C' \subseteq C, u[C' \to 0]$ means that all the clocks in C' are assigned to zero, that is, all assigned and zero clocks in C' are reset, so that $u[C' \to 0](x) = 0$ for all $x \in C'$ and $u[C' \to 0](x) = u(x)$ for all $x \notin C'$. If C' is the singleton set $\{z\}$, we shall just write $u[z \to 0]$.
  - For a given clock valuation $u \in V(C)$ and a clock constraint $\alpha \in \Psi(C)$, $\alpha(u)$ is a boolean value stating whether or not α is satisfied or not.

- A *state* is a pair (l, u) where l is a location of an automaton A and u is a clock valuation over C.

The operational semantics of a timed automaton $A = (L, E, I, \mathit{Label})$ over the clock set C is therefore defined by an infinite state transition system $M_A = (S, s_0, \to, \mathit{Label})$ where:

- $S = L \times V(C)$ is the set of states,
- s_0 is the initial state of A (l_0, u_0),
- $\to$ is the transition relation with its members defined by the following two rules:
  - *action transition:* $(l, u) \xrightarrow{a} (l', u')$ if there is an edge $(l \xrightarrow{g,a,r} l')$ such that $g(u)$ holds and $u' = u[r \to 0]$, and $\mathit{inv}(u')$ holds for each $\mathit{inv} \in I(l')$;
  - *delay transition:* $(l, u) \xrightarrow{d} (l, u')$ if, for $d \in \mathbb{R}^+$, $u' = u + d$ and $\mathit{inv}(u + d')$ holds for all $d' \leq d$ and all $\mathit{inv} \in I(l)$.
- $\mathit{Label} : S \to 2^{AP}$ is atomic proposition function extended from $\mathit{Label} : L \to 2^{AP}$ simply by $\mathit{Label}(l, u) = \mathit{Label}(l)$.

Example 2.7: Consider Figure 2.6(a). One possible transition sequence or run of the automaton is $(O, 0) \xrightarrow{prime?} (W, 0) \xrightarrow{2} (W, 2), ..., \xrightarrow{time\_elapsed?} (O, 2.1) \xrightarrow{prime?} (W, 0) \xrightarrow{1.2} (W, 1.2) \xrightarrow{close?} (C, 1.2) \xrightarrow{open?} (O, 1.2) \ldots$

There are two important observations to draw from Example 2.7: (1) the set of states of the transition system M_A is infinite and this is due to the real valued clocks, and (2) the successor behaviour of the states forms groups, that is, some sets of states such as $(O, 0)$, $(O, 2)$, and $(O, 1.2)$ do not depend on the clock valuation. However the group of states $(W, u(x))$ has

one possible successor behaviour for $u(x) < 2$ and the other for $u(x) == 2$. The existence of groups of states leads to the introduction of *clock regions*, which are the key to model checking real-time systems. Before we look at this in more detail, the syntax and semantics of timed computational tree logic (TCTL) are given.

TCTL Syntax: The syntax of TCTL is based on the syntax of CTL, extended with clock constraints. In order to clearly define the syntax, the following definitions are given:

- A *path* is an infinite sequence $s_0 a_0 s_1 a_1 ...$ states alternated by transition labels such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \geq 0$, where a_i is either (g, a, r) or d.
- Let $\rho \in S^w$ denotes a path. For $i \geq 0$, $\rho[i]$ denotes the $(i+1)^{th}$ element of ρ (see Section 2.1.2).
- $P_M(s) = \{\rho \in S^\omega \mid \rho[0] = s\}$ is a set of paths starting at s (see Section 2.1.2).
- A *position* of a path is a pair (i, d) such that d equals 0 if $a_i = (g, a, r)$, and equal a_i otherwise.
- Let $Pos(\rho)$ be the set of positions in ρ. For convenience the state $(l_i, v_i + d)$ can also be written as $\rho(i, d)$.
- A total order of positions is defined by:
  - $(i, d) \ll (j, d')$ if and only if $(i < j) \vee (i = j \wedge d \leq d')$.
- Path ρ is called *time-divergent* if $lim_{i \to \infty} \Delta(\rho, i) = \infty$, where $\Delta(\rho, i)$ denote the time elapsed from s_0 to s_i, i.e.,
  $\Delta(\rho, 0) = 0$
  $$\Delta(\rho, i) = \Delta(\rho, i) + \begin{cases} 0 \text{ if } a_i = (g, a, r) \\ a_i \text{ if } a_i \in \mathbb{R}^+ \end{cases}$$
- Let $P_M^\infty(s) = \{\rho \in S^\omega \mid \rho[0] = s\}$ denote the set of time-divergent paths starting at s.

Let $p \in AP$ and D be a non-empty set of clocks that is disjoint from the clocks of A (i.e., D is the set of clocks of the TCTL-formulas and $(C \cap D) = \{\}$), $z \in D$ and $\alpha \in \Psi(C \cap D)$. The TCTL-formulas are then defined by the following BNF:

$$\beta ::= p \mid \alpha \mid \neg\beta \mid \beta \vee \beta \mid z \textbf{ in } \beta \mid \exists[\beta U \beta] \mid \forall[\beta U \beta]$$

A clock constraint α is defined over formula clocks and timed automaton clocks and thus allows comparison of both formula and timed automaton clocks. Clock z is known as a *freeze identifier* and bounds formula clocks in β. For instance, $\forall[\beta\, U_{\leq 4}\, \phi]$ can be defined as z **in** $\forall[(\beta \wedge z \leq 4)\, U\, \phi]$.

TCTL Semantics: For $p \in AP, \alpha \in \Psi(C \cup D)$ is a clock constraint over $C \cup D$, model $M = (S, \rightarrow, L)$ is an infinite transition system, $s \in S$, $w \in V(D)$, and ψ, ϕ TCTL-formulas. The satisfaction relation $\models$, is defined as in [42].

$$\begin{array}{lll}
s, w \models p & \text{iff} & p \in L(s) \\
s, w \models \alpha & \text{iff} & v \cup w \models \alpha \\
s, w \models \neg\phi & \text{iff} & \neg(s, w \models \phi) \\
s, w \models \phi \vee \psi & \text{iff} & (s, w \models \phi) \vee (s, w \models \psi) \\
s, w \models z \textbf{ in } \phi & \text{iff} & s, w[z \rightarrow 0] \models \phi \\
s, w \models \exists[\phi U \psi] & \text{iff} & \exists \rho \in P_M^{\infty}(s) : \exists (i, d) \in Pos(\rho) : (\rho(i, d), w + \Delta(\rho, i) \models \psi \wedge \\
 & & (\forall (j, d') \ll (i, d) : \rho(j, d'), w + \Delta(\rho, j) \models \phi \vee \psi)) \\
s, w \models \forall[\phi U \psi] & \text{iff} & \forall \rho \in P_M^{\infty}(s) : \exists (i, d) \in Pos(\rho) : (\rho(i, d), w + \Delta(\rho, i) \models \psi \wedge \\
 & & (\forall (j, d') \ll (i, d) : \rho(j, d'), w + \Delta(\rho, j) \models \phi \vee \psi))
\end{array}$$

Clock equivalence

The satisfaction relation of TCTL formulas is defined in terms of an infinite transition system and not in terms of a finite state automaton as before. In other words, given a TCTL formula α and a timed automaton A, the satisfiability of α over A is defined as:

$$A \models \alpha \iff M(A), (s_0, w_0) \models \alpha$$

The main obstacle to checking whether $A \models \alpha$ is the potentially infinite state space of $M(A)$ (that is, the fact that $L \times V(C)$ may be infinite). We have already seen an instance of

this problem in Example 2.7. To make model checking of timed automaton possible, Alur and Dill [1] have proposed the idea of an equivalence relation, $\approx$, which has two important properties. The first property involves correctness, that is, it ensures that for the model M equivalent clock valuations satisfy the same TCTL-formula, whereas the second property involves finiteness, that is, replacing clock valuations with finite set equivalent classes. Let $\lfloor d \rfloor$ be the integral part and $frac(d)$ be the fractional part of a real number $d \in \mathbb{R}^+$ and let c_i be largest number (a ceiling) which is compared to a clock variable $i \in C$ of a timed automaton. Then the clock equivalence is defined as follows: $v \approx v'$ if and only if

1. $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or, for all $x \in C$, $v(x) > c_x$ and $v'(x) > c_x$,
2. $frac(v(x)) \leq frac(v(y))$ iff $frac(v'(x)) \leq frac(v'(y))$ for all $x \in C$ with $v(x) \leq c_x$ and $v(y) \leq v(y)$, and
3. $frac(v(x)) = 0$ iff $frac(v'(x)) = 0$ for all $x \in C$ with $v(x) \leq c_x$.

Figure 2.7 depicts an example of clock regions for an automaton with one and two clocks. Two clock valuations are in the same clock region if they satisfy the same clock constraints. The integral part of the clock valuation determines whether the clock constraints are satisfied or not, while the fractional part refers to a clock which will change its integral part first.

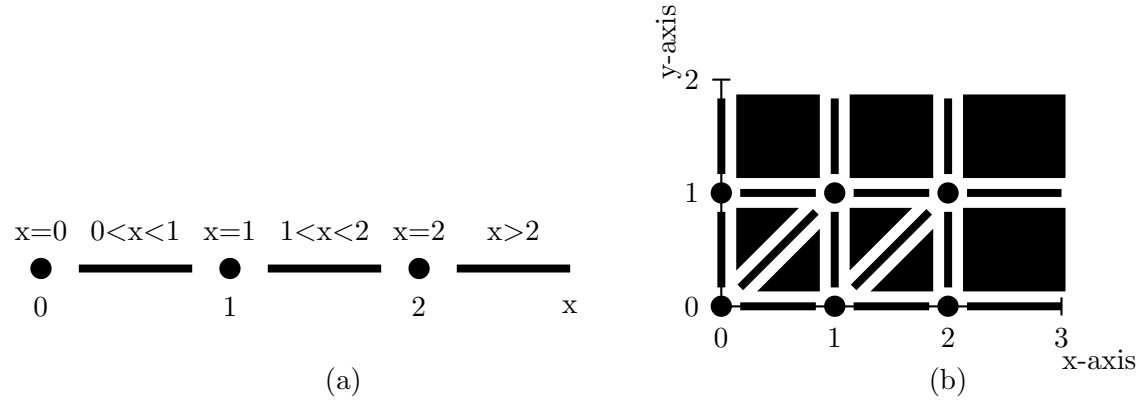


Figure 2.7: Clock regions

Model checking region automaton

Informally, a *region automaton* can be seen as a product of equivalence classes and a timed automaton. Figure 2.8 depicts an example of the region automaton constructed from the equivalence classes depicted in Figure 2.7(a) and the timed automaton depicted in 2.6(a). The resulting *region graph* R_A is finite and each node (or state) of R_A is a pair $(l, [u])$, where l is a location in A and $[u]$ is a clock region. The transition relation of the graph R_A is defined as follows:

- There is a transition $(l, [u']) \xrightarrow{a} (l', [u'])$ in R_A if there is a transition $(l, u) \xrightarrow{a} (l', u')$ in M_A
- There is a transition $(l, [u']) \xrightarrow{d} (l, [u'])$ in R_A if there is a transition $(l, u) \xrightarrow{d} (l, u')$ in M_A

The model checking of a timed automaton involves the following four steps [42]:

1. Determine equivalence class under equivalent relation $\approx$.
2. Construct the region automaton.
3. Apply the CTL model checking algorithm.
4. $A \models \alpha$ if and only if $(l_0, [u_0]) \in Sat^R(\alpha)$.

Unfortunately, the number of regions is exponential in the number of clocks and the number of ceilings of all clocks [8, 30, 64].

Model checking clock constraints

A more compact and efficient representation of the timed automaton is the so-called *zone graph*. This is based on the notion of a clock zone. A clock zone is a conjunction of inequalities comparing either a clock value or a difference between two clock values to an integer. By introducing a special variable x_0 which is always equal to zero, Clarke et al. [30] formally define a clock zone as:

$$x_0 = 0 \wedge \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \prec c_{i,j}$$

In a timed automaton, a clock constraint is either an invariant of a location or a guard of a transition and hence can be used for state reachability analysis. Clarke et al. [30] have shown that three operations — *intersection*, *reset* and *elapsing* — can be applied to clock zones to define the transition relation between states of the zone. In other words, these operations are sufficient to define how successor states are computed from current states.

- Intersection between two clock zones φ and Θ (i.e. $\varphi \wedge \Theta$) is also a clock zone.
- Reset of set of clocks r for a clock constraint φ, results in a clock zone $\varphi[r \rightarrow 0]$.
- Elapsing of time from a clock zone φ, results in a clock zone $\varphi^{\uparrow}$.

A pair (l, φ) is a state (or zone) of a zone graph Z_A, where l is a location and φ is clock zone. The transition relation in Z_A is then defined as follows:

- There is a transition $(l, \varphi) \xrightarrow{a} (l', (\varphi \wedge g)[r \rightarrow 0] \wedge I(l'))$ in Z_A for each edge $(l \xrightarrow{g,a,r} l')$ in M_A.
- There is a transition $(l, \varphi) \xrightarrow{d} (l, \varphi^{\uparrow} \wedge I(l))$ in Z_A for each l of the timed automaton M_A.

An example of a zone graph corresponding to the timed automaton depicted in Figure 2.6(a) and clock region shown in Figure 2.7(a) is shown in Figure 2.9. The graph in Figure 2.9 clearly has far fewer states and transitions than the graph in Figure 2.8, and consequently it is much more efficient to model check.

One data structure that can be used to represent a zone graph is known as a *difference bound matrix* (DBM). Interested readers are referred to [8, 30, 60] for a detailed description of the DBM. The model checking algorithm for the model checker UPPAAL, which is discussed in more detail in Chapter 6, is based on clock zones as described in [8, 30].

2.2 Theorem Proving

Theorem proving is the process of using deductive methods to develop computer programs that show that some statement (i.e., conjecture) is a logical consequence of a set of axioms and hypotheses. Theorem proving is another type of system verification technique that can be applied to formal specifications of models. Conjectures, axioms and hypotheses must be written in a logic which is accepted by an automated theorem prover. Theorem provers, also known as proof assistants, are computer programs that are used to assist users to produce proofs that state why and how the conjectures are derived from the axioms and hypotheses in such a way that it can be agreed upon by everybody.

The main advantages of theorem proving are that it helps users to have a deeper understanding of the system specification and that it is not limited by the size of the state-space, that is, large systems that cannot be verified using the model checkers discussed in Sections 2.1.1, 2.1.2 and 2.1.3, may still be verified by theorem prover. Unfortunately, theorem proving process is generally harder and requires considerable technical expertise and a deep understanding of the specification. It is also generally slower, more error-prone and labour intensive.

Prototype Verification System (PVS) theorem prover is a complex system which consists of a specification language, a number of predefined theories, a theorem prover, various utilities, and documentation with examples that illustrate different ways of using the system in several application areas [48, 50]. The algorithms that are implemented in PVS are beyond the scope of this thesis and are not discussed. The following explanation is taken from [63]:

> PVS [50] is a specification and verification environment developed by SRI International's Computer Science Laboratory. It provides an integrated environment for the development and analysis of formal specifications, and supports a wide range of activities involved in creating, analyzing, modifying, managing and documenting theories and proofs. In distinction to other widely used verification systems, such as HOL [35] and the Boyer-Moore prover [15], PVS supports both a highly expressive specification language and an interactive theorem prover in which most low-level proof steps are automated. The system consists of specification language [48], a parser, a type checker, and an interactive proof checker [49]. The

PVS specification language is based on a richly higher-order logic that permits a type checker to catch a number of semantic errors in specifications. The PVS prover consists of a set of inference steps that can be used to reduce a proof goal to simpler sub goals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps incorporate arithmetic and equality decision procedures, automatic rewriting, and BDD-based boolean simplification.

2.3 Related Work

The success of model checking and theorem proving in the design and verification of software and hardware systems is well documented. Despite the success stories about the application of formal verification in both software and hardware systems, not many practitioners incorporate formal verification in their software development and George et al. [5] claim that one of the reasons which hinders the transfer of technology is the lack of empirical studies. It is well known that among advantages of finite-state verification, early detection of errors leads to a cheaper correction of such errors. Research among the formal methods community seems to have been focused more on the development of efficient model checking algorithms to combat the problem of *state explosion* [33, 34, 45, 54, 57, 58], and there is very little work that has been done in the comparative analysis of verification tools. This claim is supported by George et al. [5], as they clearly show that there is a lack of set of benchmarks which can assist software developers to choose appropriate tools for verification analysis of a particular type of system.

In this thesis an empirical case study is undertaken on three model checkers (SPIN, SMV, and UPPAAL) and a theorem prover (PVS). The study involves the formal verification of the SIS at iThemba LABS, of which its detailed specification is presented in Chapter 3. Some of the empirical case studies on verification techniques were conducted by the following researchers: Corbett [25], Chamillard et al. [18], Mark et al. [3], George et al. [4, 6], Jensen et al. [40], Pasareanu [51], Dong et al. [28, 29], Currie [26], Mariëlle Stoelinga [55], Brard et al. [9], and Devillers [27].

Corbett [25] uses SPIN, SMV and INCA for verifying deadlock freedom in Ada programs. He

uses an automated conversion tools to translate finite-state automata of an Ada-like language into the input languages: INCA, SPIN and SMV. And this is done with the consultation of the developers of these tools. Chamillard et al. [18] later extend Corbett's study by using his translations and application specific properties in addition to the deadlock property and they use the following tools: SPIN, SMV, INCA and FLAVERS. They further use statistical analysis to assess the biasness of Corbett's translators. Although both Corbett and Chamillard et al., notice a considerable variation of both absolute and relative performance of the tools, neither of them are able to conclusively pinpoint the source of the differences [18, 25]. Corbett's study suggests that the communication structure of the application has an effect on the performance of SPIN and SMV, while INCA's performance is influenced primarily by the sizes of tasks to be performed. Chamillard et al., on the other hand, build predictive models for the failure and performance of the tools, but their only conclusion is that while predictive models are fairly good in predicting failure, they cannot identify those features of programs that affect the performance of the tools.

Dong et al. [28] used COSPAN, MURφ, SMV, SPIN and XMC to verify the i-protocol (an optimized sliding-window protocol for GNU UUCP). Among these model checkers, XMC belongs to the authors. Dong et al [28] construct abstract models manually and after verification they conclude that XMC performs better than SPIN and SMV. On the contrary, Holzmann [38] the author of SPIN, conducts the same study using the same abstract models that are used by Dong et al. [28] and he finds that Dong et al. [28] make two mistakes when using SPIN. The first mistake involves the use of parameters, Holzmann finds that they override the default settings of SPIN which hurts its performance. The second mistake is about the equivalence of abstract models. Holzmann modifies the models written in SPIN's input language and discovers that the SPIN model (by Dong et al. [28]) contains more than twice as much per state compared to other model checkers. After these modifications, SPIN performs better than XMC in almost of all the cases. However, the results helps Holzmann to come up with a new SPIN version 3.3.0, which outperformes XMC in all cases. Dong et al. [29] conduct a similar study on i-protocol and the tools used are COSPAN, MURφ, SPIN and XMC of which each of the tools support some kind of explicit-state model checking. The results of the study build upon the results of [38] and replace or improve upon the results presented in [28].

George et al. [6] have compared SPIN, SMV, INCA and FLAVERS by verifying properties of event dispatch mechanism in Chiron [56], which had a deadlock that was never described by its developers. The results show that no matter how equivalent system models are, the tools yield different performance results and they are not able to conclude with certainty as to what could be the source of the differences. Jensen et al. [40] compares SPIN and UPPAAL for verification of Collision Avoidance Protocol for an Ethernet–like medium. Jensen et al., do not measure the performance of the tools rather they compare them in terms of properties which can easily be verified by the other tool as opposed to another. SPIN is used to verify untimed properties while UPPAAL is used to very timed properties. Brard et al. [9] investigate comparative performances of UPPAAL, KRONOS and HYTECH on the Railroad Crossing. The property verified is the safety property that whenever a train is inside the crossing, the gate must be closed. The analysis is based on forward, backward and on-the-fly verification techniques. On both forward and backward analysis KRONOS performs better than HYTECH, while on-the-fly analysis UPPAAL outperforms KRONOS. However, surprisingly, backward analysis for both KRONOS and HYTECH outperforms on-the-fly analysis of UPPAAL. Brard et al. [9] do not conclude with conformity as to what could be the sources of these performace variations.

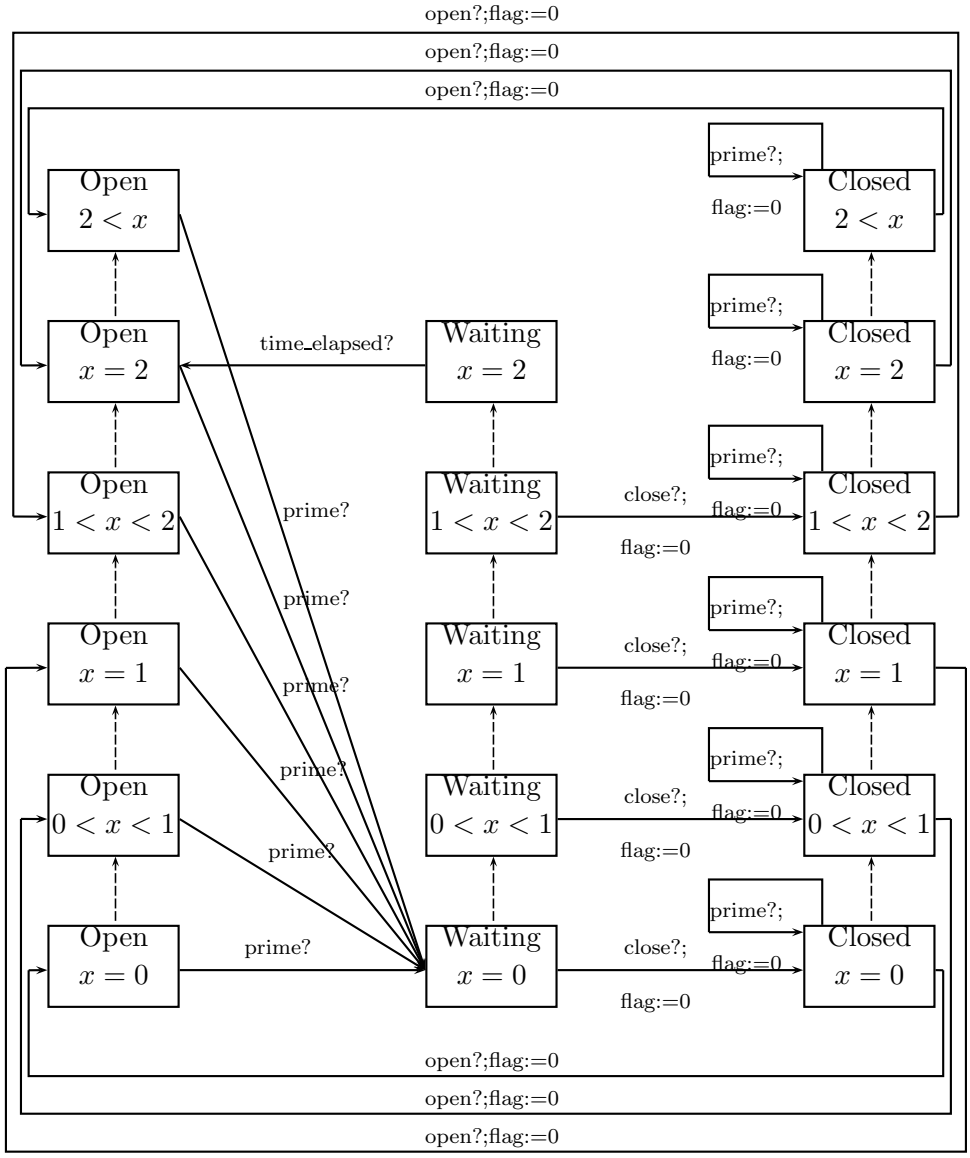


Figure 2.8: Region graph corresponding to the access door control system (see 2.6).

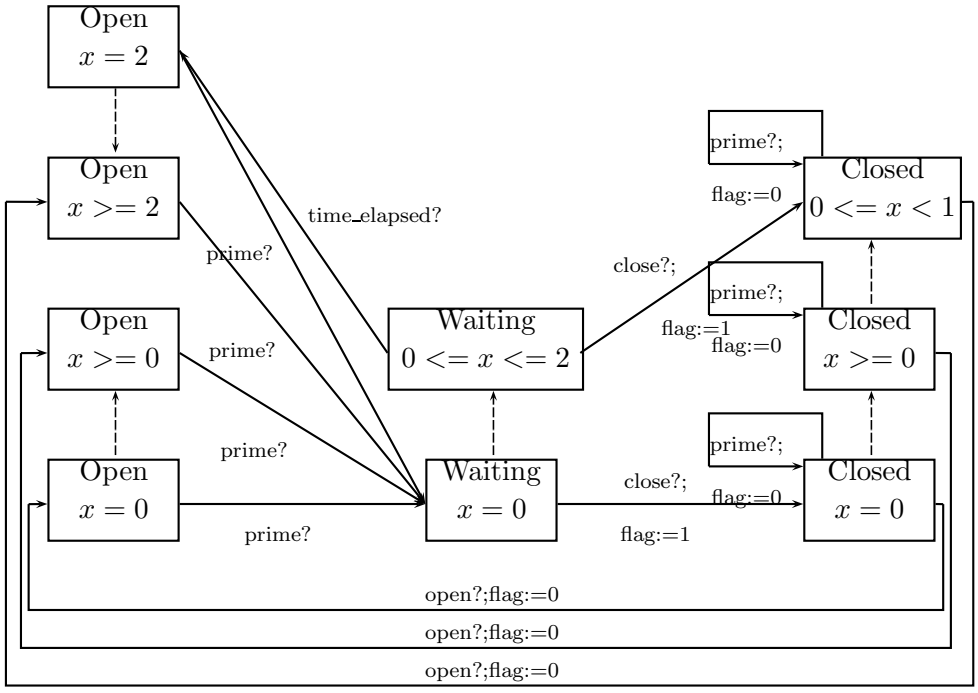


Figure 2.9: Zone graph corresponding to the access door control system (see 2.6).

Chapter 3

Safety Interlock System Specification

This Chapter discusses the system specification of the case study. As already mentioned in Chapter 1, the system of the case study is the safety interlock system at iThemba LABS. The iThemba LABS (`http://www.tlabs.ac.za`) is a multidisciplinary research facility involved in basic and applied research using particle beams, particle radiotherapy for the treatment of cancer, the supply of accelerated-produced radioactive isotopes for nuclear medicine and research, and similar activities. Currently, iThemba is engaged in a new project called "Second Beam Line Project" (2BL), that involves an additional beam line for the treatment of cancer using protons, and the development of a system referred to as the Therapy Control System (TCS). As one may expect, TCS is large and complex and comprises of a number of subsystems and components that work together to achieve the effective and safe treatment of patients. One central feature of the system is the way in which its parts are *interlocked* (in other words, synchronized) to ensure its smooth and safe operation.

Figure 3.1 shows an overview of the system and some of the electronic units and subsystems involved. The thick, dotted arrow that runs from left to right in the middle of the figure shows the path of the physical beam. It originates in accelerator control (AC) system and passes through the beam analysis and control system (through the beam current, beam steering, and energy degrader controller units) and through the primary and secondary treatment nozzles

(through the two ionisation chambers, range modulator, range verifier, and general electronic units) before emerging to treat the patient. At the top right is the patient positioning system, the supervisory system, the dose monitoring system, and the high voltage power supply unit.

The heart of the system is the therapy safety control system, shown in the bottom right of Figure 3.1. It consists of the safety interlock system (described in more detail in Section 3.1) and the master and physics consoles (described in Section 3.1.3). All these parts are connected in two ways: by a 13-line bus and an ethernet network. Finally, there are also control connections between the safety control and the accelerator control system.

TCS operates in three mutually exclusive modes:

1. Patient treatment is conducted in *clinical mode*. In this mode none of the interlocks may be overridden.

2. The *physics mode* is used mainly for physics experiments and it allows more flexibility in the configurations of the beam line and treatment nozzle as the clinical mode. In this mode it is possible to override a limited set of the interlocks to the safety interlock system via the supervisory system.

3. The *test mode* is used mainly for the development and test purposes, and it allows for even more flexibility in the configurations of the system. In this mode it is possible to override an even larger set of interlocks.

The rest of the chapter focuses on the safety interlock system, which is part of the therapy safety control system as shown in Figure 3.1.

3.1 Safety Interlock System

The main task of the safety interlock system (SIS) is to monitor and evaluate the safety conditions in the system as a whole, using inputs from therapy safety bus (TSB) and also hard wires from all over the TCS. When the operating conditions of the system are violated, the SIS must send control commands to the accelerator control system to take appropriate action. The SIS consists of the safety interlock computer (SIC), the programmable logic

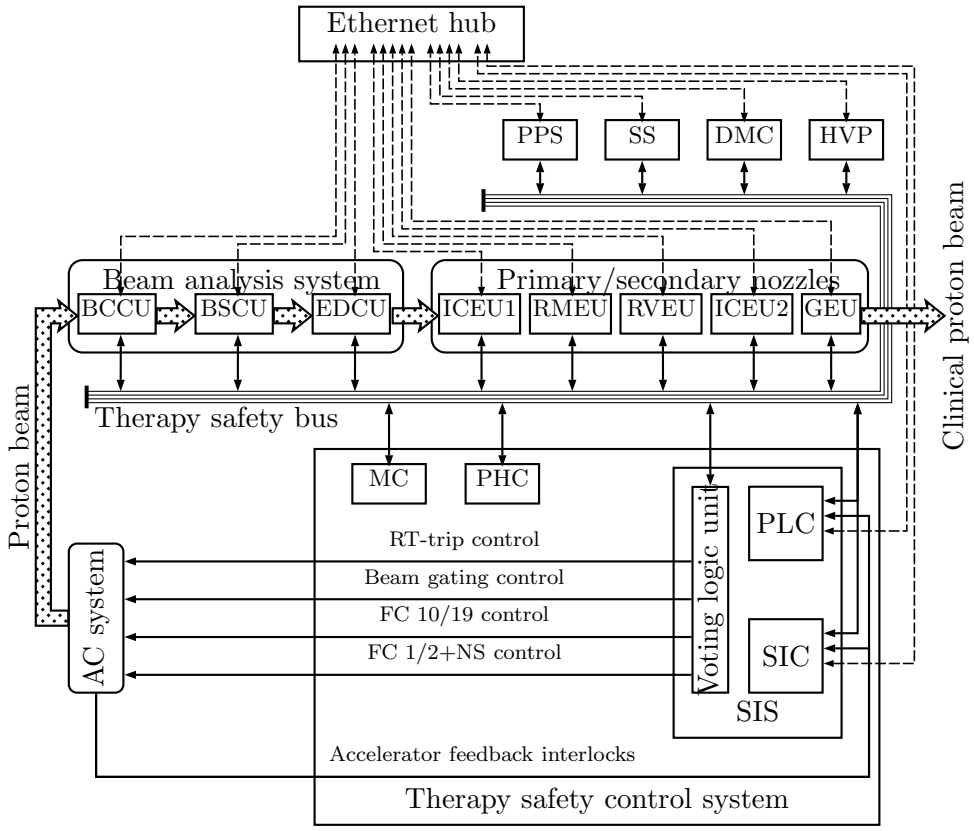


Figure 3.1: Architecture of the 2BL

controller (PLC) and the voting logic unit (VLU). The SIC is a standard personal computer, while the PLC is a special-purpose computer designed for control tasks. One of the goals of the case study is to write a control program such that the TCS systems including SIS can behave as intended. A control program runs on both SIC and PLC and both computers (should) produce the same outputs to the VLU, which then synchronises the outputs and produces the final output to the accelerator control system. The control program runs a permanent loop (i.e., a cycle) and each cycle involves three steps, that is, the first step is to read inputs from TSB and hard wires, the second step is to evaluate conditions of the TSB and hard wires. The last step is to send control commands to the accelerator control system if operating conditions are violated.

The components of the system consist of comprehensive arrangement of relays, switches and transistor-controlled circuits which ensure correct and safe operational conditions at all times.

There are two categories of interlocks namely: *personnel interlocks* and *machine interlocks* that ensure safe operation of the whole TCS. The two categories of interlocks are not completely independent, personnel interlocks provide safe and accurate delivery of patient treatment prescription and the protection of radiotherapy staff and the general public. Machine interlocks, on the other hand, prevent the equipment being operated, that is, this category of interlocks is concerned with safe operation of machines and prevent the damage which can be done to them. In this study, both personnel and machine interlocks are referred to as input interlocks and there are two types of input interlocks namely: discrete interlocks and non-discrete interlocks. Discrete interlock inputs are communicated to the SIS via hard wires while non-discrete interlock inputs are communicated indirectly, that is, through TSB. Indirect communication refers to the fact that the exact detail of which interlock failed is not known to the SIS from TSB status, SIS only knows that there is a failure from reading the status of the TSB.

The main purpose of the TSB is to provide a fast means by which any system of the TCS can communicate its functional and hardware failures detected to the rest of the system, and most importantly to the SIS as shown in Figure 3.1. Sections 3.1.1 and 3.1.2 present the detailed description of the discrete and non-discrete interlocks, respectively, and Section 3.1.3 explains two types of consoles that are used to manipulate the interlocks. Sections 3.2 — 3.4 discusses systems which directly interact with the SIS.

3.1.1 Non-Discrete interlocks

Electronic units in TCS indirectly communicate system status, requests and failures to the SIS via TSB. TSB consists of a number of discrete wires which will henceforth be called TSB lines and each of these lines represents a specific status or request in the TCS. The TSB lines *CONSOLE-ON*, *PRIMARY-NOZZLE*, *SECONDARY-NOZZLE*, and *BEAM-ON* communicate current system configuration status to the SIS while the TSB lines *SABUS* and *HIGH-VOLTAGE-PSU* communicate interlock failures (see Table 3.1). The TSB lines *RF-TRIP OFF*, *BEAM DEFLECTOR OFF*, *FC 1/2 & SHUTTER OUT*, *FC 10/19 OUT*, *PHYSICS MODE*, *TEST MODE* are used to communicate requests to and from the SIS (see Table 3.2). A current source is attached to each of these lines, so a TSB line has a value *true*

if the current is flowing through the line otherwise it has a value *false*.

Table 3.1: Status for TSB Lines

TSB Line	Description
CONSOLE-ON	indicates that either the master or physics console is switched on.
HIGH-VOLTAGE-PSU	indicates a failure of one or more of the high voltage power supplies.
SABUS	indicates that some system has failed. Any system may set the status of this line.
PRIMARY-NOZZLE	indicates that the primary nozzle is active.
SECONDARY-NOZZLE	indicates that the secondary nozzle is active.
BEAM-ON	indicates that the proton beam is on. The beam is on when both the neutron shutter and the high energy FC 1/2 are not in the path of the beam. This status will be calculated from the status feedback received from the Neutron Shutter and the High Energy FC 1/2.

The systems and components that communicate their requests and status through continuous (i.e., Non-Discrete) interlocks are beam current controller unit (BCCU), beam steering controller unit (BSCU), energy degrader controller unit (EDCU), ionization chambers electronic unit 1 (ICEU1), range modulator electronic unit (RMEU), range verifier electronic unit (RVEU), ionization chambers electronic unit 2 (ICEU2), general electronic unit (GEU), high voltage power supply unit (HVP), patient positioning system (PPS) and dose monitoring system (DMC). All of these units can change the status of the lines so, it is not possible for the SIS to identify the interlock failed since the TBS does not provide the details of which interlock(s) failed. If the SIS detects a failure from any of the systems and/or electronic units, it will send the interlock statuses via LAN, to the supervisory system for displaying purposes. Figure 3.1 shows all the electronic units and the systems which communicate non-discrete interlocks to the SIS. Table 3.3 lists all non-discrete interlocks associated with each of the

Table 3.2: Requests for TSB lines

TSB Line	**Description**
RF-TRIP OFF	indicates to the SIS when it should send an RF-trip request signal to the cyclotron (accelerator) control system.
BEAM DEFLECTOR OFF	indicates to the SIS when it should send a request to the beam gating control unit (BGCU) to gate the beam off by deflecting it with the high voltage plates of the beam gating device. This request line is defined separate from the Faraday Cup (FC) request lines since it is necessary to switch the beam off in some cases without dropping the FC cups into place.
FC 10/19 OUT	indicates when the SIS should send a request signal to the Faraday-cups (accelerator) control system to extract the Faraday cups FC-10 and FC-19 from the low-energy beam line.
FC 1/2 & NS OUT	indicates when the SIS should send a request signal to the Faraday-cups (accelerator) control system to extract the Faraday cups FC-1, FC-2 and the Neutron shutter from the low-energy beam line. This request line is defined separate from the faster Faraday Cup 10/19 request lines. The Faraday Cup 10/19 lines will be used with the beam deflector request to switch the beam off when small patient movements are detected. The Faraday Cup 1/2 and shutter line will be used when large patient movements are detected.
PHYSICS MODE	indicates that the physics mode is selected on the active console.
TEST MODE	indicates that the test mode is selected on the active console.

systems and electronic units whose status will be displayed on the supervisory system. The interlocks can be set to a value *true* or *false* depending on whether they are overridden or not due to the mode in which the system is operating in as explained above.

3.1.2 Discrete Interlocks

Discrete interlocks are communicated to the SIS via hard wires and each wire is directly connected from an interlocked device. The components and systems that communicate their interlock status through hard wires include general interlocks (GI), primary and secondary treatment nozzles and beam line interlocks (TNBL), accelerator group interlocks (AGI) and room clearance interlock (RCI). Table 3.4 lists all these interlocks grouped in four categories. All these categories of interlocks are manipulated by TCS — they will henceforth be called *TCS interlocks* — except AG category which is manipulated by accelerator control systems. The interlocks can either have a value *true* or *false* and can also be overridden depending on the mode in which the system is operating in. The SIS will then transmit the status of all these discrete interlocks via LAN to the supervisory system for displaying purposes.

3.1.3 Master and Physics Consoles

In addition to the SIS, therapy safety control system also includes master and physics consoles. The purpose of the master console (MC) is to provide the user (radiation therapist or physicist) with a simple interface to select between physics, test and clinical mode, to start and stop the beam, and to perform an emergency stop at any time. The physics console (PHC) will be physically separated from the MC but it must perform the same functions as the MC. Only one of the consoles must be active at any given time. The Master console should also provide the radiation therapist with feedback of the beam characteristics and dose delivery to the patient. The console shall also indicate the status of the TSB and the room clearance system status. Room clearance system is part of the therapy safety control system which ensures safe evacuation of the treatment room.

Table 3.3: Non-Discrete Interlocks

Category	**Non-Discrete Interlocks**
Beam Current Controller Unit (BCCU)	(1) BCCU Status
	(2) BCCU PID Status
	(3) BCCU Beam-Off Status
Beam Steering Controller Unit (BSCU)	(1) BSCU Status
Energy Degrader Controller Unit (EDCU)	(1) EDCU Status
	(2) EDCU Transition Status
	(3) EDCU Position Sensing Status
Ionization Chambers Electronic Unit 1 (ICEU1)	(1) ICEU1 Status
	(2) Beam Ratio Status
	(3) Beam Alignment Status
Range Modulator Electronic Unit (RMEU)	(1) Range Modulator Status
Range Verifier Electronic Unit (RVEU)	(1) Range Verifier Status
Ionization Chambers Electronic Unit 2 (ICEU2)	(1) ICEU2 Status
	(2) In Plane Symmetry Status
	(3) Cross Plane Status
General Electronic Unit (GEU)	(1) GEU Status
	(2) First Scatter Status
	(3) Second Scatter Status
High Voltage PSU (HVP)	(1) HVP Status
Patient Positioning System (PPS)	(1) PPS Status
Supervisory System	(1) Patient Components Ok
	(2) Patient Identification Ok
Dose Monitoring Control System (DMC)	(1) DMC Ok Status

Table 3.4: Discrete Interlocks

Category	**Discrete Interlocks**
General Interlocks (GI)	(1) Area Radiation Monitor
Treatment Nozzle and Beam Line Interlocks (TNBL)	(1) Pointer Laser Cover Status (2) Snout IN or OUT (3) Patient Collimator Status (4) Patient Compensator Status (5) Beam Defining Lamp IN or OUT (6) Laser IN or OUT (7) Ionization Chamber Set B IN or OUT (8) X-Ray Tube IN or OUT (9) First Scatter IN or OUT (10) Range Modulator Assembly IN or OUT (11) Energy Degrader IN or OUT (12) MWIC IN (13) Second Scatter OUT
Accelerator Group (AG)	(1) ACC Safety Status (2) FC 10 & 19 IN or OUT (3) FC 1 IN or OUT (4) FC 2 IN or OUT (5) Neutron Shutter IN or OUT
Room Clearance Interlock (RCI)	(1) Panic Buttons Status (2) Basement Door Status (3) Left Beam Line Door Status (4) Right Beam Line Door Status (5) Left Door Prime Input (6) Right Door Prime Input (7) Room Prime Input (8) Boom Status (9) Infrared 1 or 2 Status (10) Room Arm Input (11) Room Disarm Input

3.2 Supervisory system

Just like the master and the physics consoles, the purpose of the supervisory system (SS) is to provide the user (radiation therapist or physicist) with the following: (1) a simple interface to select between physics, test and clinical mode, (2) to select between primary and secondary treatment nozzles, (3) to start or stop the control system. In addition, it can also change the status of the TSB lines just like any other systems attached to the bus lines (see Figure 3.1). The system is also responsible for displaying the status of the TSB lines, room clearance system interlocks, TCS interlocks and accelerator control system interlocks.

3.3 Room clearance system

There is a sequence of actions that should be followed to make sure that the treatment room is armed (i.e., ready for patient treatment). Room clearance system (RCS) is responsible for ensuring that this procedure is followed and it is outlined as follows:

1. There are eight emergency buttons that are distributed around the treatment vault. All these eight buttons must be normal (a normal condition is a short circuit) for the treatment to begin, this means that each circuit representing a button must be closed. If at any point in time one of these buttons is pressed (that is a circuit representing a button is open), the room must resort to a safe condition, that is, neither armed nor primed — prime refers to the intermediate preparations of the room, e.g., if an access door is ready to be closed.

2. There is a gate placed across the entrance to the basement of the treatment room and this gate must be closed for the treatment to begin. When the gate is closed its circuit is also closed and when the door is open its circuit is also open. If at any time this gate is open, the room must resort to a safe condition that is neither primed nor armed.

3. There is a door to the annex off the maze and this door must be closed for the treatment to begin. When the door is open its circuit is also open and when it is closed its circuit is also closed. If at any time this door is open, the treatment room must resort to a safe

condition, that is, neither armed nor primed.

4. There are two access doors in the partition on either side of the beam-line. These doors must be primed and closed with 10 seconds before the room is primed. When any of these doors is open, its circuit is also open and when it is closed its circuit is also closed. If any of these doors is not primed and closed within 10 seconds the treatment vault must resort to a safe condition that is neither armed nor primed. If at any time any of these doors is open the vault must resort to a safe condition.

5. If all the conditions from 1 to 4 are satisfied, that is, a circuit for each device is closed, then the room may be primed for evacuation. The priming is done by pressing an exit button in the treatment room.

6. Once the room has been primed, the operators have 40 seconds to leave the room and close the boom gate at the maze exit, a circuit for the boom gate is open when the gate is open and closed when the gate is closed. The boom gate must be primed and closed for the treatment to begin and if at any time the boom gate is open the room must neither be primed nor armed. If the boom gate is not closed within 40 seconds, the room must resort to a safe condition after which the room may be primed once again.

7. If the boom gate is primed within the allowed 40 seconds, the room may be armed at any time after the closing of the boom gate. This means sending a *Ok* signal to the SIS.

Once all the above conditions are satisfied and the room is in the *armed* state, it is easy to return to the state in which the room is ready to be primed, thus removing *Ok* signal to the SIS. The removal of *Ok* to the SIS can be achieved in six different ways:

1. There are two infra-red detectors in the maze. If either of the detectors is tripped, the room goes to a state in which it can be primed.

2. There is a disarm button in the control room and the treatment room can be disarmed by depressing this button.

3. If the boom gate described in items 6 and 7 above is opened, the treatment room goes to a state in which it maybe primed.

4. If either of the access doors described in item 4 above is opened, the room goes to a state in which it is ready to be primed.

5. If the gate across the entrance to the basement which is described in item 2 above is opened, the treatment room returns to a state in which it can be primed.

6. If any of the panic buttons described in item 1 is pressed, the treatment room returns to a state in which it can be primed.

Most of the outputs from room clearance system are informative, that is, they do not have an effect on the operation of the system but just informs the personnel about the status of the RCS. There is however one output which is not only informative but also have some effect on the operation of the system, *Room Armed*. The *Room Armed* output informs the SIS that the treatment room is ready for treatment.

3.4 Accelerator control system

There are two types of beam stop devices: a Faraday cup which is a cup shaped piece of copper and a neutron shutter which is a steel cylinder for shielding radiation. Both Faraday cup and neutron shutter have two micro-switches, associated with each extreme movement of the device, which are used to detect whether the device is in the beam line or not. There are five of these beam-stop devices and they are listed as follows:

1. Faraday Cup 1: can be in or out of the beam line and it is located at the end of the beam line, that is, it is the last beam stop device just before the patient.

2. Faraday Cup 2: can be in or out of the beam line and it is located between the cyclotron and the neutron shutter.

3. Faraday Cup 10: can be in or out of the beam line and it is located next to the injector cyclotron.

4. Faraday Cup 19: can be in or out of the beam line and it is located next to the main cyclotron.

5. Neutron Shutter: can be in or out of the beam line and it is located in the wall of the treatment room, that is, between Faraday cups 1 and 2.

Accelerator control system (ACS) is responsible for the extraction of these devices out of the beam line and their insertion into the beam line, due to commands received from the SIS. The system produces ten feedback outputs to the SIS, that is, whether a device is *in* or *out*, where *in* represents in the beam line (safe condition) and *out* represents out of the beam line (unsafe condition). There are ten outputs since each device produces two outputs.

Chapter 4

Methodology

This chapter presents a methodological approach which is followed in the design and verification of the SIS described in Chapter 3. There are two main goals of the study: (1) the design of mathematical models as a basis for the implementation of error-free software for the SIS at iThemba LABS, and (2) a comparison of formal method techniques that addresses the lack of much-needed empirical studies in this field. To achieve these goals, a five phase approach is followed. Phases one to four will be discussed in this chapter. Phase five will be spread over Chapters 5 — 8, each describing a selected tool. The first phase is about the selection of formal verification tools and it is presented in Section 4.1. Section 4.2, which is the second phase, involves verification criterion and it explains the criterion used to evaluate and compare these tools. Section 4.3 is the third phase and it presents a mathematical framework of the SIS. Section 4.4 involves a tool-independent formulation of system properties.

4.1 Selection of verification tools

Model checking and theorem proving are useful in first, helping to verify the correctness of the system design and thus increasing reliability; second, they are fault avoidance techniques that assist in reducing the number of errors in the systems at the earliest state of system development. There are a number of tools that implement these techniques and they differ in terms of degrees of rigor, that is, different type of properties they can verify, the logic

used etc. In this thesis, model checkers SPIN, UPPAAL, SMV and a theorem prover PVS are selected due to the following criteria:

1. Popularity (Number of case studies in industrial applications, publications indicating application in the real world, etc.)
2. Support of the tool (recent publications, updates from its website, etc.)
3. Representation of properties (Safety, Liveness, deadlock, temporal, real-time, etc.)
4. Representative of verification technique (explicit, symbolic, etc.)
5. Degree of complexity (easy of use; simulation, graphical interface)

Investigation conducted so far in this thesis reveals that SPIN seems to be the most popular model checker since it is used in a number of case studies including [18, 20, 22, 24, 25, 26, 28, 29, 6, 38, 39, 51, 65]. Second to SPIN is SMV which appears in the following case studies: [18, 22, 24, 25, 28, 6, 61, 51, 55, 19], to name a few. The third is UPPAAL, and most of its case studies including [2], appear in its website — `http://www.uppaal.com/`. It should be noted that model checkers are not released in the same year and it is possible that the ranking done may not be accurate. Moreover, it is not the aim of the study to compare and rank the tools. However, SPIN, UPPAAL, and SMV are the most popular model checkers used today. Variety of theorem provers (PVS, HOL, OTTER, etc) also exists and PVS is among the most used theorem provers and this is due to a number of case studies it is applied to, which include [46].

In this study, the support of a tool is based on the recent publications about the tool and the latest release date of the tool. The model checkers SPIN, SMV, and UPPAAL fall among the mostly supported tools. SPIN has an annual workshop which provides an environment for SPIN users to meet and interact directly to exchange experiences, ideas, theories, wishes, improvements, etc., about the tool. At the writing of this thesis, its latest was Version 4.2.7 released on 23 June 2006. UPPAAL is the most supported tool for verification of real time systems and the latest official release of Version 4.0.2 was released on Aug 7, 2006. As for Cadence SMV, it was released in 1996 and it is based on the original SMV from Carnegie

Mellon University by the same author — McMillan. Pvs's latest Version is 3.2 and it is one of the most supported theorem provers.

The selection of tools is also based on the expressive power of representing different system properties. The tools can be categorised into two groups, Spin and Smv can be used to verify qualitative temporal properties, while Uppaal and Pvs can be used to verify quantitative timing constraints. These groups can be used together to complement each other in verifying the properties of the system. Different state-space techniques, such as explicit and symbolic, are adopted in various tools. Spin represents tools using explicit state-space representation, while Smv and Uppaal represent tools that use some form of symbolic representation. The ease of use also plays an important role in achieving the ultimate goal of formal verification, that is, the transfer of technology into the industry, therefore all the tools are quite easy to use except Pvs.

4.2 Verification criterion

In this thesis four tools are used to verify properties of the SIS. Each of these tools is developed by different authors, have different verification algorithms, employ different state-space representation techniques, use different logics etc. In short, the tools are different and to deal with the problem of biasness in the verification process, models are based on the same mathematical framework, discussed in section 4.3. The mathematical framework may favor other tools, but the idea is to model the system with an appropriate framework. However, where possible, similar constructs like *if-statements, message passing etc.,* are used.

The performance analysis of the tools is based on four factors, first the time taken to understand and be able to use a tool; second, the time taken to analyse a property; third, the amount of memory used in verification of a property and fourth, number of states and transitions generated during verification of a property. The effectiveness of a tool is to be able to analyse large systems and this is normally based on time and memory usage as these are the limiting factors of verifying large systems. To assess the use of these tools in industry, subjective conclusions as to which of the tools is easier to learn and use than the other is also considered. The findings and conclusions of the study are discussed in Chapter 9.

4.3 Architectural design of the SIS

Glück and Holzmann [32] stated that "*Flight software is the central nervous system of modern spacecraft*" and the same thing can be said about the SIS in relation to the TCS at iThemba LABS. SIS is the heart of the TCS and it is sitting between the TCS systems and the accelerator control system. The architectural design of the SIS is depicted in Figure 4.1 and an object-oriented approach is followed in this design, where each *box* in the picture refers to an object. The modelling of each object follows the timed automaton framework proposed by Alur et al. [1], which is presented in Section 2.1.3. Section 4.3.1 explains the execution cycle of the system and Sections 4.3.2 — 4.3.7 describe the timed automata for each object (or system), where "!" means send output and "?" means receive input.

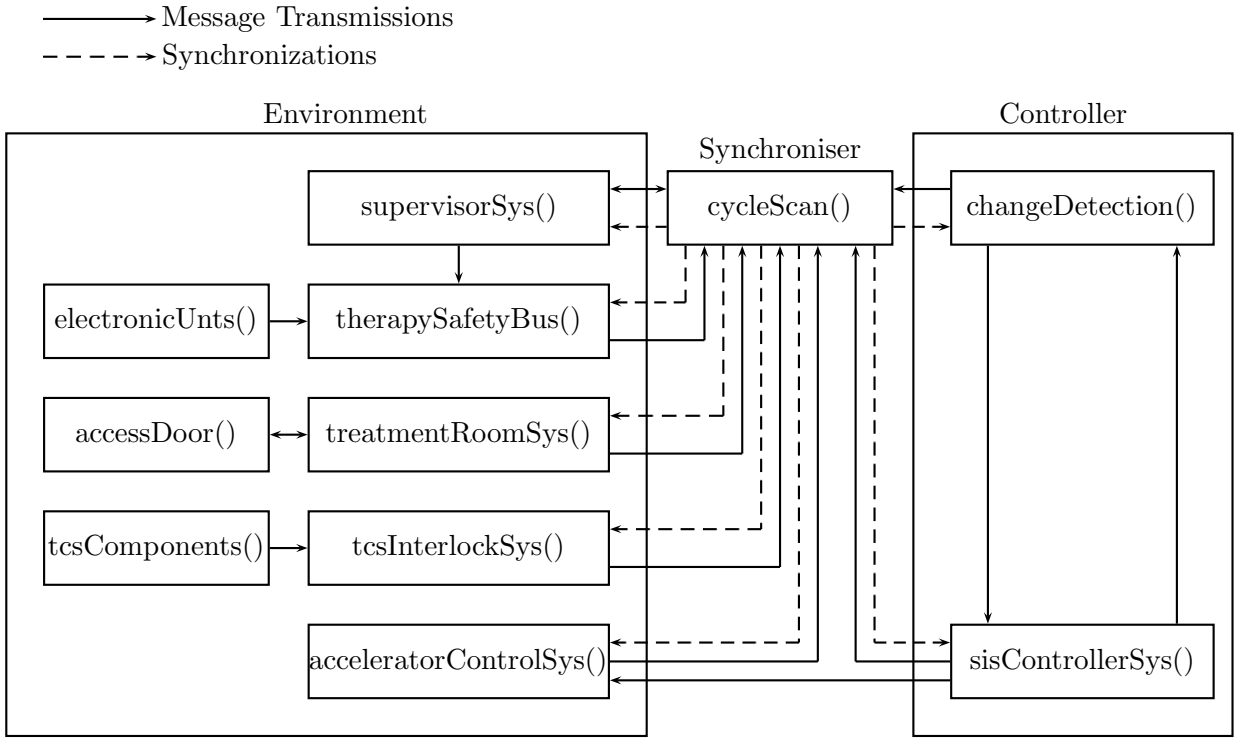


Figure 4.1: Controller Architecture for the SIS

4.3.1 Cycle scan

The control program for SIS runs a permanent loop (i.e., a cycle scan) and in each cycle all the processes providing the inputs and requesting outputs are executed once. The process *cycleScan* manages the synchronisation between all the processes. The advantage of the design is that there is an explicit synchronisation between all the processes and also the controller process (*sisControllerSys*) can only execute when a change of the input line(s) has been detected, thus make a fast controller process. Figure 4.2 depicts the cycle of the system. The *cycleScan* process distributes a token in a cyclic fashion and a process with the token executes its code and returns the token when it has finished its execution. The process is also responsible for managing shared data, the shared data refers to an array variable which stores all the inputs that are required by the controller process to make appropriate decisions.

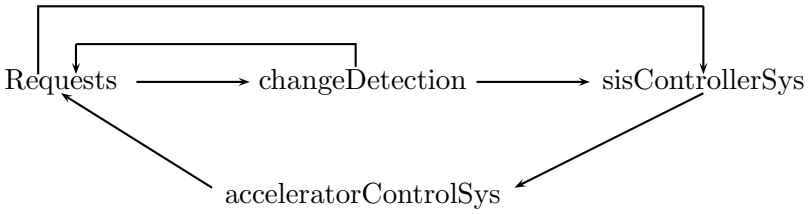


Figure 4.2: A control cycle

The cycle starts by first reading the input requests from the physical environment or receiving a stop command from the *supervisorySys* process. If the system starts with input requests, the *changeDetection* process goes through the updated shared input array to check if there are any changes. If a change is detected, the *changeDetection* process informs the *sisControllerSys* process about the change, otherwise it gives the token back to start a new cycle. When the *sisControllerSys* process receives a request it will eventually send control commands to the *acceleratorControlSys* process and the commands will be processed only when the control token is given to the *acceleratorControlSys* process. If a new cycle is started by receiving the stop command from the *supervisorySys* process, the *changeDetection* process is bypassed and the token is given to the *sisControllerSys* process to synthesis appropriate commands to the *acceleratorControlSys* process, which will then be given the token to process the request.

4.3.2 Supervisory system

The description of the SS is presented in Section 3.2 and Figure 4.3 depicts its automaton.

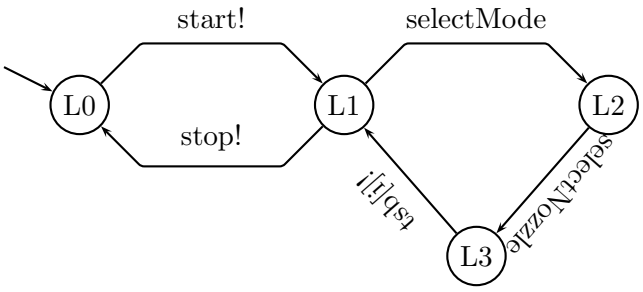


Figure 4.3: Supervisory system automaton

The system goes in synchronised cycle, it begins by first starting the entire control system and allows the selection of operational mode and the treatment nozzle. The system can also updates the TSB lines after which it can either stop the system or wait for another execution cycle. The value of i corresponds to a TSB line and ranges between zero and eleven (i.e., $0 \leq i \leq 11$).

4.3.3 Therapy safety bus system

Therapy safety bus system is responsible for managing the updates of the TSB lines. The corresponding automaton is given in Figure 4.4. The system receives requests from all the systems and electronic components that communicate their hardware and functional failures through TSB. When it is time for it to execute, the system responds with an updated TSB lines to allow the control system to make appropriate decisions. The decisions are mainly control commands to the accelerator control system to either extract or insert Faraday cups and neutron shutter in the beam line.

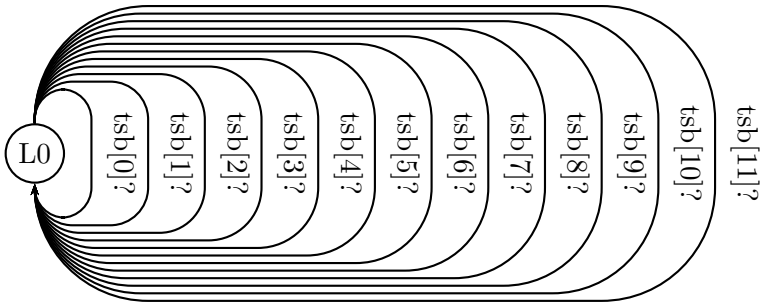


Figure 4.4: Therapy safety bus automaton

4.3.4 Room clearance system

The RCS, which is discussed in Section 3.3, is broken down into smaller and less complex timed automata. The first component is concerned with priming and closing access doors and a timed automaton for the procedure is depicted in Figure 4.5.

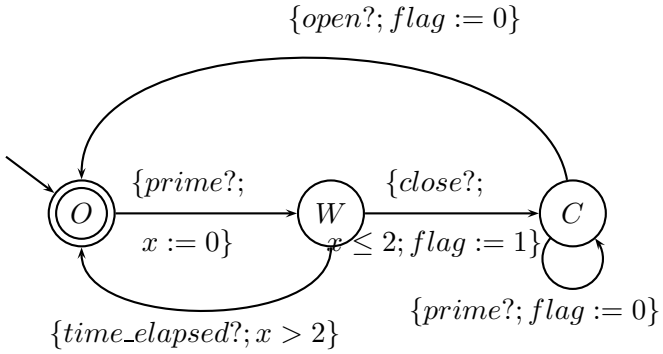


Figure 4.5: Access door control system

The access doors should be primed and closed within x units of time ($x \leq 2$ is an example as shown in Figure 4.5) and the process *accessDoor* models the procedure by setting a *flag* to indicate that the access doors have been primed and closed successfully, otherwise the *flag* is cleared.

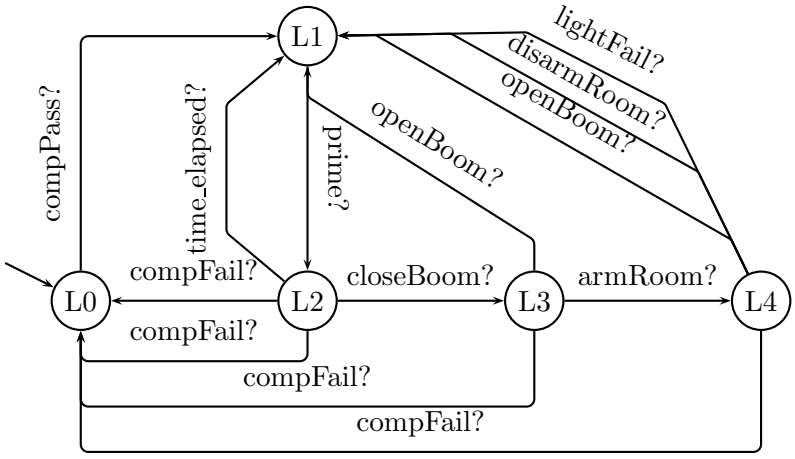


Figure 4.6: Room clearance timed automaton

Figure 4.6 depicts the main timed automaton of the RCS. The process of the automaton calls the process of the automaton — *accessDoor* — every cycle scan to compute the composite *flag* which is set if the room is primed and armed successfully, otherwise cleared to indicate failure.

4.3.5 Therapy control interlock system

Therapy control interlock system is responsible for the management of all the TCS interlocks. Figure 4.7 depicts the corresponding automaton.

The system's behaviour is similar to the therapy safety bus system's behaviour, even though it does some extra work. The system keeps the status of all the discrete interlocks up to date, that is, it receives requests of the systems and electronic devices that change their interlocks and when it is time for it to execute, the system checks the mode in which the entire system is operating and then checks whether the interlocks which are not overridden are all working properly. If the interlocks are working as expected the system responds with a *flag* assigned a value *true*, or a value *false* otherwise. At the writing of this document, the total number of interlocks were not yet finalised, so i in Figure 4.7 represents an interlock in the range $0 \leq i \leq N$, where N is the total number of interlocks.

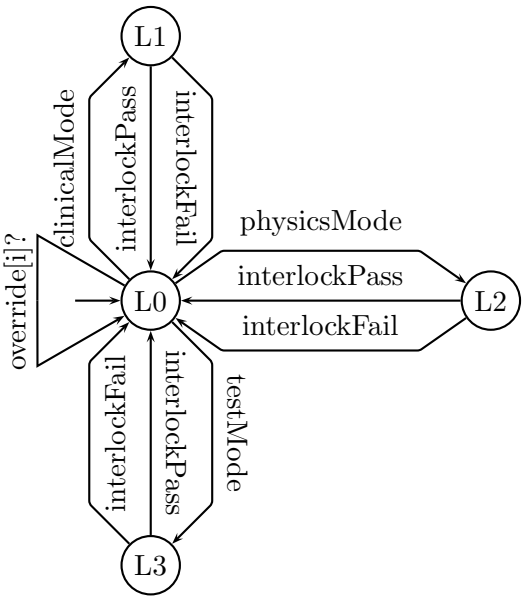


Figure 4.7: Therapy control interlock automaton

4.3.6 Accelerator control system

Accelerator control system is responsible for the extraction and insertion of Faraday cups and a neutron shutter and its description is presented in Section 3.4. Figure 4.8 depicts a timed automaton for the system.

The extraction of the Faraday cups and the neutron shutter can proceed if and only if all the beamline components are working properly and the treatment room is cleared. The system knows about the status of all these conditions from the commands sent by the controller process — *sisControllerSys*. If all the conditions are satisfied the system will extract neutron shutter and then Faraday cups 1, and 2 and finally Faraday cups 10 and 19. If during the extraction any failure is detected either from beam line components or treatment room, then the system should immediately stop extracting the devices and start inserting them in the beam line. The *pass* action in the picture refers to the success of extracting or inserting the devices in the beam. The eDevice (e.g., eFC1) refers to the request for extrating a device while iDevice (e.g., iFC1) refers to the request for inserting the device.

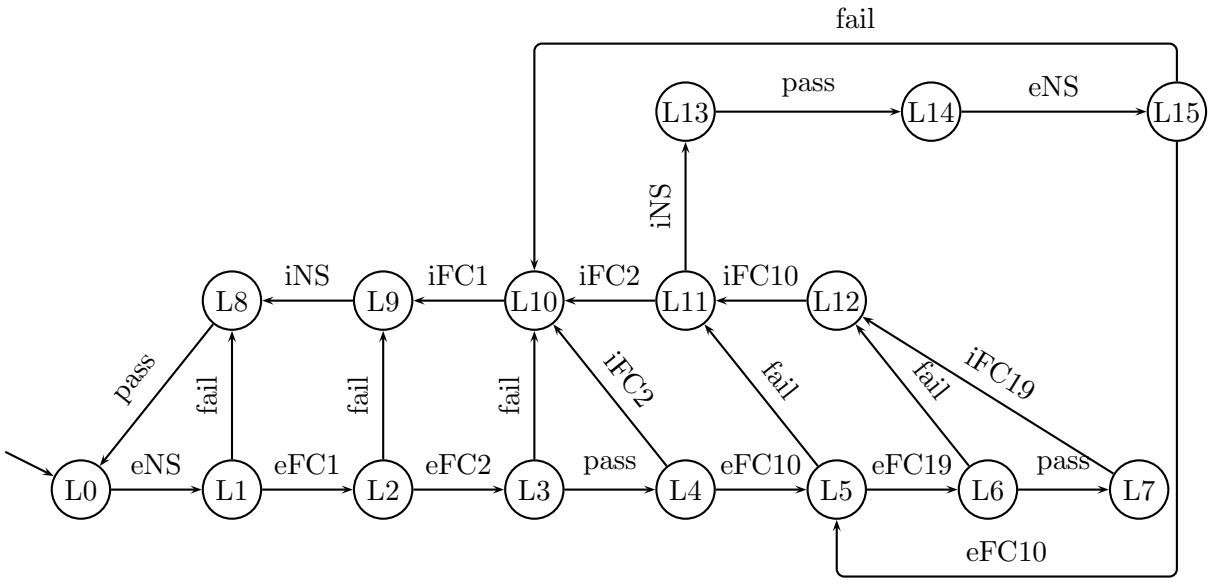


Figure 4.8: Accelerator system automaton

4.3.7 Safety interlock control system

The control system generates three control outputs to the VLU derived from TSB lines, room clearance system inputs, TCS inputs and feedback from accelerator control systems. These control outputs are used to control Faraday Cup 1, Faraday Cup 2, Faraday Cup 10, Faraday Cup 19, Neutron Shutter, Beam Gating and RF-Trip. The control system is defined as a timed automaton, depicted by Figure 4.9 and consists of thirteen control locations: *idle*(L0), *clinical beam off*(L2), *physics beam off*(L1), *test beam off*(L3), *faraday cup 1, faraday cup 2 and neutron extraction*(L4), *switching beam off*(L5), *faraday cups 10 and 19 extraction* (L6), *beam on*(L7), *beam gated*(L8), *switching movement on*(L9), *switching movement off*(L10), *movement*(L11), *failure*(L12).

The system begins at location *idle*(L0) and make a transition to either *clinical beam off*(L2), *physics beam off*(L1) or *test beam off*(L3) locations, depending on whether clinical mode, physics mode or test mode is selected respectively. The *Faraday cup 1, Faraday cup 2 and neutron extraction*(L4) refers to an ordered sequence of extracting Faraday cups 1 and 2 as well as neutron shutter before extracting Faraday cups 10 and 19 out of the beam line. This is done to prevent secondary particle (neutron) production when the proton beam hit Faraday

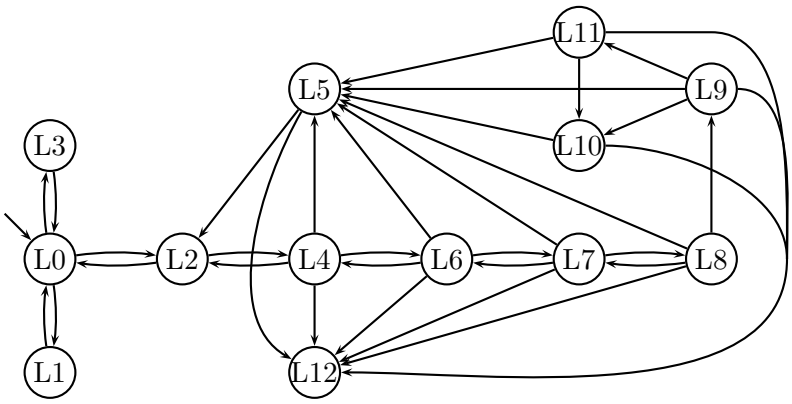


Figure 4.9: Timed automaton for safety interlock system

cups 1 or 2 in case the beam is not deflected. Faraday cup 1, Faraday 2 and neutron shutter must be completely extracted before the extraction of Faraday cups 10 and 19. The extraction of all Faraday cups and neutron shutter is limited to a period of time. In the *Faraday cups 10 and 19 extraction*(L6) location Faraday cups 10 and 19 are extracted and the location changes to *beam on*(L7) when Faraday cups 10 and 19 are completely extracted, that is, completely out of the beam line.

The beam is gated on upon entry into the *beam on*(L7) location and all Faraday cups and neutron shutter are removed out of the beam line, but the beam is not gated! In this location the beam can be on for an unlimited period of time. In the *beam gated*(L8) location all Faraday cups and neutron shutter are removed from the beam line but the beam is gated off. This location can be active for an unlimited period of time. In the *switching movement on*(L9) location the beam is gated off and Faraday cups 10, 19 and neutron shutter are in the process of being inserted in the beam line. This location must be active for a limited period of time. If this time period is exceeded the location should change to *switching beam off*(L5) or *failure*(L12) location. In the *movement*(L11) location the beam is gated off and Faraday cups 10, 19 and neutron shutter are in the beam line. This location can be active for an unlimited period of time. In the *switching movement off*(L10) location, the beam deflection is turned off and Faraday cups 10, 19 and neutron shutter are in the process of being extracted from the beam line. This state is active for a limited period of time and if this time period is exceeded the location should change to *switching beam off*(L5) or *failure*(L12) location.

In the *switching beam off*(L5) location actions are performed that are the opposite of the actions performed in the *Faraday cup 1, Faraday cup 2 and neutron extraction*(L4) and *Faraday cups 10 and 19 extraction*(L6) locations. In the *Faraday cup 1, Faraday cup 2 and neutron extraction*(L4) location, the Faraday cups and the neutron shutter are inserted into the beam line at the same time. The beam is already deflected when this location becomes active. Beam deflection is an exit condition of the previous location. A timer is started when this location is entered. The location changes to the *failure*(L12) location, if the neutron shutter or Faraday cups are not extracted before the timer expired. *Failure*(L12) location is entered due to a failure detected in one of the other locations of the timed automaton. In this location the timed automaton must force the system into a "safe" condition. In a safe condition the beam is gated off and the Faraday cups and neutron shutter are inserted in the beam line. This location can be active for an unlimited period of time. An RF-Trip can also be generated in this location to be even more "safe" by being surer that the beam is not coming through anymore.

4.4 System properties

Section 4.3 presents the architectural design of the SIS that is followed to model the system as well as verifying the correctness properties of the models developed in Chapters 5, 6, 7 and 8. This thesis only considers untimed properties, but the models are designed in such a way that clock variables can be added to allow the verification of timed properties in Uppaal and Pvs verification tools. The following four properties are verified:

1. ***Property:*** The SIS should be free from deadlocks.

   Description: TCS comprises of a number of systems — including SIS— which communicate through wired lines and Ethernet. It is important to make sure that the system is free from deadlocks.

2. ***Property:*** Always, when the access doors are not primed and closed successfully (i.e., on time), then the treatment room cannot be armed.

   Description: Treatment room comprises of a number of components, such as access

doors, panic buttons, etc. Access doors are primed and closed before the room can be primed and armed, therefore the property makes sure that this protocol is followed.

3. ***Property:*** It is always the case that, when the clinical mode is selected from supervisory system and the interlock flag is set to true, it implies that all the TCS interlocks are not overridden.

   Description: The TCS operates in three modes, namely clinical, physics, and test modes. TCS interlocks flag is used to indicate whether or not the appropriate interlocks are overridden due to the mode in which the system is operating.

4. ***Property:*** It is always the case that, when the treatment room is not armed, then the extraction of Faraday Cups and neutron shutter cannot occur.

   Description: This property is used as a representative for all the interlocks that form conditions for the generation of control commands to the accelerator control system. The interlocks are therapy safety bus lines, TCS interlocks flag, treatment room flag, and feedback interlocks from accelerator control system.

Chapter 5

The SPIN model checker

SPIN (Simple Promela INterpreter) is a model checking tool for verifying the correctness of distributed software systems such as operating systems, data communication protocols etc., in a rigorous and mostly automated fashion. The tool originated at AT&T's Bell Labs developed by Gerard Holzmenn. The tool specifically targets software system not hardware systems and the input language is PROMELA (PROtocol MEta-LAnguage). SPIN follows an automata-theoretic approach and the algorithms for model checking LTL-formulas are basically the ones discussed in Chapter 2, Section 2.1.1. The model checker SPIN is a very powerful system as it does not only perform model checking, but also supports a number of features including guided, interactive and random simulation, use of embedded C code as part of model specification, dynamically growing and shrinking numbers of processes, both rendezvous and buffered message passing, and communication through shared memory. The tool also assists in tracing logical errors in the system design and checks logical consistency of the system specification. The other interesting feature is that it does verification on-the-fly, that is, it does not preconstruct the entire state graph as a prerequisite for verification of properties, but constructs as it is required.

The SPIN tool is a very complex system and Figure 5.1 gives an overview of the tool. Interested readers who would like to know more about the architectural design of the tool are encouraged to read the following book [39], or a paper [37] or browse through the website `http://spinroot.com/spin/whatispin.html`. In this chapter the SIS model is developed

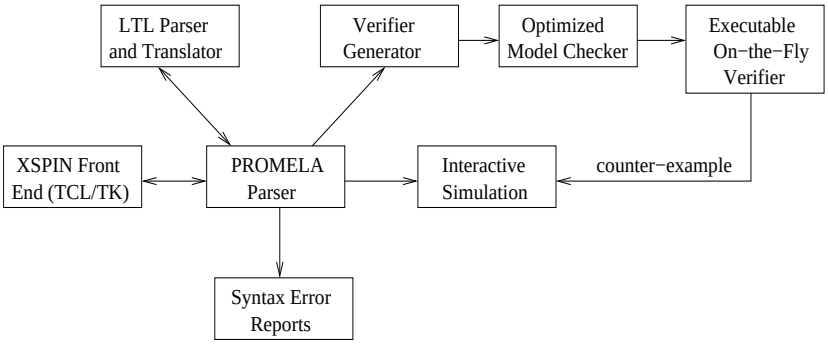


Figure 5.1: Architectural design of SPIN [37].

in PROMELA and verified with SPIN. PROMELA is a C-like programming language and it is mostly based on Dijkstra's guarded command language and in this chapter, only PROMELA constructs that are used in the model are explained and for a complete reference of the language, an interested reader can visit the website — `http://spinroot.com/`.

5.1 An overview of Promela

A PROMELA specification that is presented in this chapter consists of processes, directives, global and local variables, communication channels, run, and init constructs. The constructs can be summarized as follows:

- **#define** directive, global, and local variables have the same semantics as in C/C++ programming language.
- A process **P** is defined as follows
  Proctype P (formal parameters) { local variables; statements }
- If-statement has the form
  if
  :: guard1 → statements
  :: ...
  :: guardn → statements
  fi

If one of the guards is enabled, the corresponding statements will be executed and if more than one guard is enabled the selection of the guards is done non-deterministically.

- Do-statement has the form

  do

  :: guard1 → statements

  :: ...

  :: guardn → statements

  od

  The selection of the guards in the Do-statement is the same as If-statement except that the latter does not terminate after the execution of the statements unless the statements being executed include either **goto** or **break**.

- The constructs **init** and **run** are used to instantiated processes. The processes P and Q can therefore be instantiated as follows

  init{run P(actual parameters); run Q(actual parameters);...}

- Processes communicate between channels, declared with a constructs **chan** and the capacity of the channel can be zero or more. Zero capacity of a channel refers to the blocking communication between processes while more than zero refers to the non-blocking communication. The messages are processed in a first-in first-out manner and sending is indicated by an exclamation mark (!) while reception by a question mark (?).

5.2 The SIS model in Promela

The SIS is the heart of the TCS, and it is sitting between the TCS systems and the accelerator control system. The validation and verification of the SIS is crucial to ensure correct and safe operation of the TCS and this section describes an approach for designing the system in Promela. An object-oriented design is followed, all the systems providing the SIS with inputs are divided into components, as shown in Figure 4.1, of which each component is represented by a process.

5.2.1 Synchronisation process

Figure 5.2 depicts a PROMELA source code for synchronisation cycle between processes. As mentioned in Section 4.3.1, the *scanCycle* process distributes execution time in a cyclic manner to all the processes and thus enables transmission of data between the controller and the environment. More precisely, at the beginning of the cycle, the process receives all the requests from SS, TSB, RCS, TCS interlocks and feed back from ACS (lines 6-10). The process then updates the shared variables of the controller (line 16) and if there is no change detected from the inputs, the execution continues normally, otherwise the controller will send appropriate control commands to the accelerator control system to take necessary action.

```
1  proctype cycleScan(chan request, supervisorySysResponse,...){
2    ...
3    do
4    ::startstop ? startSys(token) ->
5       do
6       :: request ! readSupervisorySysStatus(0);
7          request ! readBuslines(0);
8          request ! readRoomline(0);
9          request ! readTCSInterlocks(0);
10         request ! readFeedbacklines(0) ->
11           supervisorySysResponse ? supervisoryStatus(sysMode,sysNozzle);
12           tsbSysResponse ? tsbStatus(tsb[0],tsb[1],tsb[2],...);
13           rcSysResponse ? roomStatus(roomFlag);
14           tcInterlockSysResponse ? tcsInterlocksStatus(interlocksFlag);
15           acSysResponse ? acceleratorSysStatus(fbk[0],fbk[7],...) ->
16              ...
17              cyclescan ! issueCommands(0) ->  cyclescan ? return(token)
18      :: startstop ? stopSys(token); cyclescan ! issueCommands(0);
19              cyclescan ? return(token); break
20      od
21    od
22 }
```

Figure 5.2: Promela source code for synchronisation cycle

5.2.2 Supervisory system process

The main function of the SS is to communicate configuration data with all the processes interacting with the controller. Figure 5.3 depicts the PROMELA source code for the system. The process first allows the *cycleScan* process to start distributing execution time (line 4). Since in every execution cycle, the *supervisorySys* process is executed first, this ensures that the correct configuration data in that cycle is send to the processes that require it to make accurate decisions (lines 10 and 16). The process is also supposed to send interlocks that are overridden to the *tcsInterlockSys* process and the changes it has made on the TSB lines to the *therapySafetyBus* process, but these "send statements" are not shown in the process. They are absorbed in the destination processes (i.e., *tcsInterlockSys* and *therapySafetyBus*). The process then responds to the *cycleScan* process with the selected system mode and treatment nozzle (line 16). The process can also stop the *cycleScan* process to distribute the execution time (line 17).

```
1  proctype supervisorySys(chan read, response, write, startstop){
2      ...
3      do
4      :: startstop ! startSys(token) ->
5           do
6           :: if
7              :: mode = 1
8              :: mode = 2
9              :: mode = 3
10             fi; write ! operationMode(mode) ->
11             if
12             :: nozzle = 0
13             :: nozzle = 1
14             fi;
15             :: read ? readSupervisorySysStatus(token) ->
16                       response ! supervisoryStatus(mode,nozzle)
17             :: startstop ! stopSys(1);break
18          od
19     od
20 }
```

Figure 5.3: Promela source code for supervisory system

5.2.3 Therapy safety bus lines process

The main purpose of the TSB line process — *therapySafetyBus* — is to model the behavior of the TSB lines. Figure 5.4 depicts the PROMELA source code for the process. The TSB lines can either take a value *true* (lines 4 and 6) or a value *false* (lines 5 and 7) due to functional and hardware failures detected from the TCS systems and the decision taken by the controller process depends on the true values of these lines. An example of a hardware failure maybe when one of the systems on the SABUS racks fails, which then clears the SABUS STATUS TSB line. The line maybe indexed with a value i in an array variable tsb[i] (line 2), where $0 \leq i \leq N$ and N equals the number of TSB lines. In every cycle scan the process responds with the new values of the TSB lines (lines 9-10) to let the controller process take any required decision if the changes are detected.

```
1  proctype therapySafetyBus(chan write, read, response){
2      bit tsb[6];
3      do
4      :: tsb[0] = 1
5      :: tsb[0] = 0
6      :: tsb[1] = 1
7      :: tsb[1] = 0
8      ...
9      :: read ? readBuslines(token) ->
10           response ! tsbStatus(tsb[0],tsb[1],...)
11     od
12 }
```

Figure 5.4: Promela source code for therapy safety bus

5.2.4 Room clearance system process

There is a defined protocol (referred to as "room clearance system") for preparing the treatment room to be ready for patient treatment. The RCS process — *treatmentRoomSys* — is responsible for safe and correct functioning of the protocol. The system is broken down into smaller and less complex processes — *accessDoorSys* and *treatmentRoomSys*.

The *accessDoorSys* process is concerned with priming and closing the side doors and its PROMELA source code is depicted in Figure 5.5. The process keeps a *flag* variable which can either be set or cleared. When the *flag* is set (line 7), it implies that the door has been primed and closed successfully, otherwise it is cleared (lines 4-6). In every cycle, the process responds with the current state of the door, either primed and cleared successfully or not (line 8).

```
1  proctype accessDoorSys(chan read, response){
2      bool S0=1,S1,S2,doorFlag;
3      do
4      :: atomic{input1Location(S2) ->
5            doorFlag=0; outputLocation(S0)}
6      :: atomic{input1Location(S0) -> doorFlag=0; outputLocation(S1)}
7      :: atomic{input1Location(S1) -> doorFlag=1; outputLocation(S2)}
8      :: read ? readDoorstatus(token) -> response ! doorStatus(doorFlag)
9      od
10 }
```

Figure 5.5: Promela source code for access door subsystem

Figure 5.6 depicts the main process of the RCS — *treatmentRoomSys*. The process is responsible for priming and arming the treatment room. It also keeps a *flag* variable to test whether the room is ready or not. The room can only be primed and armed if all other componets, in this case access doors, have been successfully primed and closed. In every cycle, the process reads the *flag* (line 6) of the process *accessDoorSys* and if its value is *true*, it proceeds with priming and arming the treatment room (8-15), otherwise it responds with a *flag* assigned the value *false* to indicate failure.

5.2.5 TCS interlocks process

The *tcsInterlockSys* process is concerned about the interlocked devices which should be working properly before and during the treatment. The inputs to the process, depicted in Figure 5.7, are binary in the sense that the device is either working (i.e., *true*) or not working (i.e., *false*) and if all the input conditions are satisfied, the TCS is allowed to continue operating normally otherwise it is not allowed to proceed, thus switch to a safe state. The process

```
1  proctype treatmentRoomSys(chan read, response){
2      bool S0,S1,S2,S3,S4=1;
3      do
4      :: read ? readRoomline(token) ->
5           response ! readDoorstatus(token);
6              read ? doorStatus(doorFlag) ->
7              do
8              :: atomic{doorFlag && input4Locations(S1,S2,S3,S4) ->
9                     roomFlag=0; outputLocation(S0)}
10             :: atomic{doorFlag && input1Location(S0) ->
11                    roomFlag=0; outputLocation(S1)}
12             :: atomic{doorFlag && input1Location(S1) ->
13                    roomFlag=0; outputLocation(S2)}
14             :: atomic{doorFlag && input1Location(S2) ->
15                    roomFlag=1; outputLocation(S3);}break
16             :: atomic{!(doorFlag) ->
17                    S0=0;S1=0;S2=0;S3=0;S4=1;roomFlag=0;break}
18             od;
19             response ! roomStatus(roomFlag)
20     od
21 }
```

Figure 5.6: Promela source code for room clearance system

is also responsible for monitoring the interlocked devices depending on the mode in which the TCS is operating in (lines 11 — 13), that is, certain interlocks are overridden (lines 4 and 6) depending the operational mode. If all the conditions are satisfied the process responds with a *true flag*, otherwise a *false flag* (line 16) to the *cycleScanSys* process, to let the controller take action.

5.2.6 Accelerator control system process

There are other important systems which are responsible for the extraction and insertion of the devices that control the beam. In this thesis, the description of these systems are not given in detail, they are just highlighted as how they relate to the SIS. Interested readers are encouraged to consult the documentation [59] at iThemba LABS. The accelerator control system process — *acceleratorControlSys* — is used to model the behaviour of the cups control

```
1  proctype tcsInterlockSys(chan write, read, response){
2      ...
3      do
4      :: override[0] = 1
5      :: override[0] = 0
6      :: override[1] = 1
7      :: override[1] = 0
8      :: write ? operationMode(ssMode)
9      :: read ? readTCSInterlocks(token) ->
10         if
11         :: ssMode == 2 && !(override[0]) && !(override[1]) ||
12            ssMode == 1 && override[0] ||
13            ssMode == 3 && override[1] -> interlocksFlag = true
14         :: atomic{else -> interlocksFlag = false}
15         fi;
16         response ! tcsInterlocksStatus(interlocksFlag)
17     od
18 }
```

Figure 5.7: Promela source code for the TCS interlocks

systems, which are responsible for extraction and insertion of the Faraday Cups (i.e., Faraday Cup 1, Faraday Cup 2, Faraday Cup 10 and Faraday Cup 19) and Neutron Shutter. Figure 5.8 depicts the process's fragment PROMELA source code. The process receives commands (line 4) from the controller process and when it is time for it to execute it responds with the state of the stopping beam devices (lines 15 — 16).

5.2.7 The SIS controller process

The SIS controller process — *sisControllerSys* — is responsible for evaluating all the interlocks and sending control commands to the ACS. Figure 5.9 presents its PROMELA source code. The main purpose of this process is to evaluate safety conditions (lines 1 — 9) in the whole therapy control system using inputs received from four categories of interlocks, that is, input from TSB lines, input from RCS, input from TCS interlocks and feedback inputs from ACS. The process produces four control commands to the accelerator control system (lines 17 and 20) and these commands are used to control Faraday Cup 1, Faraday Cup 2, Faraday Cup

```
1  proctype acceleratorControlSys(chan read, readcommands,response){
2    ...
3    do
4    :: readcommands ? ctrlCommands(RF_Trip, Beam_Gated, FC12NS, FC1019)
5    :: read ? readFeedbacklines(token) ->
6       if
7       :: atomic{(!(RF_Trip) && !(Beam_Gated) && FC12NS && FC1019) ->
8              dIn[0]=1; dIn[1]=1; dIn[2]=1; dIn[3]=1; dIn[4]=1;
9              dOut[0]=0; dOut[1]=0; dOut[2]=0; dOut[3]=0; dOut[4]=0}
10      :: atomic{(!(RF_Trip) && !(Beam_Gated) && !(FC12NS) && FC1019) ->
11             dIn[0]=0; dIn[1]=0; dIn[2]=0; dIn[3]=1; dIn[4]=1;
12             dOut[0]=1; dOut[1]=1; dOut[2]=1; dOut[3]=0; dOut[4]=0}
13      ...
14      fi;
15      response ! acceleratorSysStatus(dIn[0], dOut[0], dIn[1], dOut[1],
16                   dIn[2], dOut[2], dIn[3], dOut[3], dIn[4], dOut[4])
17   od
18 }
19
```

Figure 5.8: Promela source code for accelerator control system

10, Faraday Cup 19, Neutron shutter, Beam Gating and RF-Trip. A detailed description of the SIS is presented in Chapter 3.

```
1  #define toClinicalBeamOff() (line[0] && !(line[1]) && !(line[2]) &&
2     !(line[3]) && !(line[4]) && !(line[5]) && line[6] && !(line[7]) &&
3     line[8] && !(line[9]) && line[10] && !(line[11]) && line[12] &&
4     !(line[13]) && line[14] && !(line[15]) && line[16] && line[17])
5
6  #define toFC12NSExtraction()  (line[0] && line[1] && line[2] &&
7     line[3] && !(line[4]) && !(line[5]) && line[6] && !(line[7]) &&
8     line[8] && !(line[9]) && line[10] && !(line[11]) && line[12] &&
9     !(line[13]) && line[14] && !(line[15]) && line[16] && line[17])
10 ...
11 proctype sisControllerSys(chan read, response, cyclescan){
12    ...
13   do
14   :: cyclescan ? issueCommands(token) ->
15       if
16       :: atomic{(state ==  notReady || state == failure) &&
17          toClinicalBeamOff() -> response ! ctrlCommands(0,0,1,1);
18          state = clinicalBeamOff}
19       :: atomic{state == clinicalBeamOff && toFC12NSExtraction();
20          response ! ctrlCommands(0,0,0,1); state = FC12NSextraction}
21       ...
22       fi;
23       cyclescan ! return(token)
24    od
25 }
```

Figure 5.9: Promela source code for a controller process

5.3 Formal specification in LTL

Section 4.4 has outlined requirements of the SIS. The version of SPIN used in this thesis does not have a concept of time, henceforth the requirements verified here are those independent of time. The Properties presented in Section 4.4, can be expressed in LTL formulas as follows:

1. In SPIN, a deadlock property is built-in.

2. Let;

   $p = accessDoorSys@accessDoorClosed$, and

$q = treatmentRoomSys@roomArmed$

$\Box\ (\neg p \rightarrow \neg q)$

However, this property is not valid when verified with SPIN due to asynchronous behaviour of the processes, instead the following stability property is verified;

$$\Box((\neg p \wedge q \wedge \Diamond \neg q) \rightarrow (qU(\Box \neg q)))$$

3. Let;

$p = (ssMode == 2$ && $!(override[0])$ && $!(override[1])$ ||
$ssMode == 1$ && $override[0]$ || $ssMode == 3$ && $override[1])$, and
$q = interlocksFlag == 1$
$\Box\ (\neg p \rightarrow \neg q)$

However, as in item 2, this property is not valid when verified with SPIN due to asynchronous behaviour of the processes, instead the following stability property is verified;

$$\Box((\neg p \wedge q \wedge \Diamond \neg q) \rightarrow (qU(\Box \neg q)))$$

4. Let;

$p = treatmentRoomSys@roomArmed$, and
$q = sisControllerSys@extractionFC12NS$
$\Box\ (\neg p \rightarrow \neg q)$

However, as in items 2 and 3, this property is not valid when verified with SPIN due to asynchronous behaviour of the processes, instead the following stability property is verified;

$$\Box((\neg p \wedge q \wedge \Diamond \neg q) \rightarrow (qU(\Box \neg q)))$$

The results of the properties are discussed in Chapter 9.

Chapter 6

The UPPAAL model checker

Uppaal (UPPsala AALborg) is an integrated tool for modeling, simulation and verification of real-time systems such as real-time controllers and communication protocols in particular [7, 44]. The tool was first released around 1995 with the collaboration of researchers from the University of Uppsala in Sweden and University of Aalborg in Denmark. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. The two main design criteria for Uppaal have been efficiency and ease of usage. The application of on-the-fly searching technique has been crucial to the efficiency of the Uppaal model-checker. Another important key to efficiency is the application of a symbolic technique that reduces verification problems to that of efficient manipulation and solving of constraints.

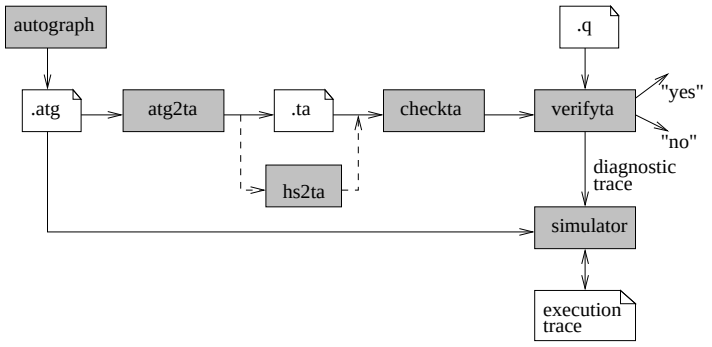


Figure 6.1: Architectural design of Uppaal [44].

Figure 6.1 depicts architectural design of the tool. It comprises of three components, model-checker, analyser and simulator [44]. In modelling, two description languages can be used, that is, graphical description or textual description. Autograph is used to define a system in graphical representation of timed automata (i.e., .atg) and textual format (i.e., .ta) is used as a programming language for timed automata. The compiler *atg2ta* is used to transform a graphical representation of timed automata (i.e., .atg) to textual representation of timed automata (i.e., .ta), which is then checked for syntactical errors by the module *checkta*. The tool UPPAAL also allows modelling, analysis and simulation of simple hybrid systems by using the compiler *hs2ta* to transform a graphical representation of hybrid system to textual representation. Model-checking is done by the module *verifyta* which takes as an input a network of timed automata in textual format (i.e., .ta) and a formula (i.e., .q). In model-checking, a diagnostic trace can be generated to check whether the formula is satisfied or not. The simulator, on the other hand, is used to interactively analyze the dynamic behavior of the system. The difference between the model-checker and the simulator is that, the model-checker explores the entire state-space while the simulator only explores a particular execution trace. It is beyond the scope of this thesis to give an intensive description of UPPAAL. Instead its overview is presented in Section 6.1 and interested readers can visit the tool's website at `http://www.uppaal.com/` for a detailed description of the tool.

6.1 An overview of UPPAAL

A UPPAAL specification consists with a number of features for describing real-time systems. In this chapter only features that are used to describe SIS are explained and they include timed automata, a guard, an assignment, a synchronization, and an invariant:

- *Timed automaton* is a standard finite state machine extended with clocks, integer variables, and synchronization labels.
- *Guard* is a boolean expression defined on clocks, constants, global and local variables that their expressions evaluate to either true or false.
- *Assignment* is a statement that sets or re-sets values to either clocks or other types of

variables.

- *Synchronization* is a mechanism that enables binary handshake between two automata. The handshake is initiated by an exclamation mark (!) while reception by a question mark (?).

- *Invariant* is a condition that should not change and is defined on clocks.

6.2 The SIS model in a network of timed automata

The same procedure, as in Chapter 5, for designing the SIS model in UPPAAL is followed. Based on the architectural design depicted in Figure 4.1, each component (box) is viewed as an object which is then represented by a timed automaton — or just an automaton when the meaning is understood from the context. The model consists of eight timed automata, namely, *cycleScan(), supervisorySys(), therapySafetyBus(), accessDoorSys(), treatmentRoomSys(), tcsInterlockSys(), acceleratorSys()* and *sisControllerSys()* presented in Sections 6.2.1, 6.2.2, 6.2.3, 6.2.4, 6.2.4, 6.2.5, 6.2.6, and 6.2.7 respectively. It should be noted that the term *process* in this chapter is replaced by the term *timed automaton* — or just an *automaton* in short. Only the fragments of the automata that present the main features of the model are highlighted.

6.2.1 Timed automaton for a cycle scan process

The responsibilities of the component *cycleScan* are presented in Section 4.3.1 and it is mentioned that one of its tasks is to enforce synchronization between all other components. In Chapter 5, the corresponding process — *cycleScan()* — does enforce a synchronised cycle between processes. However, in this chapter a slightly different approach is taken due to the problem of state-explosion, which is discussed in detail in Chapter 9. In this case, a boolean variable *cycle* is used to mimic a cyclic behaviour of the system. It toggles between *true* (lines 8-9) and *false* (lines 7 and 10). The true value of *cycle* corresponds to the request of inputs and if there is any change detected the timed automaton *sisControllerSys()* is executed to synthesize control commands to the accelerator control system. Otherwise, *cycle* is assigned

a false value to enable other automata execute their code. Figure 6.2 depicts a snapshot of the automaton.

```
1  process cycleScan() {
2  ...
3  state  S0,S1,S2,S3,S4;
4  commit S4;
5  init   S0;
6  trans
7      S0 -> S1 {sync startSys?; assign cycle:=false;},
8      S1 -> S0 {sync stopSys?; assign cycle:=true;},
9      S1 -> S2 {assign cycle:=true;},
10     S2 -> S1 {guard !(isChange()); assign cycle:=false;},
11     S2 -> S3 {guard isChange(); assign readInputs();},
12     S3 -> S4 {sync toCtrl!;},
13     S4 -> S1 {sync toAc!; assign cycle:=false;};
14 }
```

Figure 6.2: UPPAAL source code for synchronisation cycle

6.2.2 Timed automaton for supervisory system

Figure 6.3 depicts the source code for a supervisory system automaton — *supervisorySys()*. The main task of the automaton is to start (line 5) and stop (line 10) the control system, allow the non-deterministic selections of the system mode (line 6) and treatment nozzle (line 7), as well as non-deterministic selection of interlocks that are overridden (lines 8-9). The automaton achieves the same effect as the process depicted in 5.3.

6.2.3 Timed automaton for therapy safety bus lines

Figure 6.4 depicts an automaton for therapy safety bus lines — *therapySafetyBus()*. The main purpose of the automaton is to model the behavior of the TSB lines. The lines are either assigned a value *true* (line 5) or a value *false* (line 7) by the TCS systems communicating through TSB lines. Section 5.2.3 provide more information about the therapy safety bus lines.

```
1  process supervisorySys() {
2  state S0,S1;
3  init  S0;
4  trans
5    S0 -> S1 {sync startSys!;},
6    S1 -> S1 {select id:int[0,2];guard !(cycle);assign sysMode:=id; },
7    S1 -> S1 {select id:int[0,1];guard !(cycle);assign sysNozzle:=id;},
8    S1 -> S1 {select id:int[0,1];guard !(cycle);sync clearInterlock[id]!;},
9    S1 -> S1 {select id:int[0,1];guard !(cycle);sync setInterlock[id]!;},
10   S1 -> S0 {sync stopSys!;};
11 }
```

Figure 6.3: UPPAAL source code for supervisory system

```
1  process therapySafetyBus() {
2  state S0;
3  init  S0;
4  trans
5      S0 -> S0 {select id:int[0,3]; guard !(cycle); sync setTsb[id]?;
6                assign tsb[id]=1;},
7      S0 -> S0 {select id:int[0,3]; guard !(cycle); sync clearTsb[id]?;
8                assign tsb[id]=0;};
9  }
```

Figure 6.4: UPPAAL source code for therapy safety bus

6.2.4 Timed automaton for room clearance system

There is a defined procedure for preparing a treatment room for patient treatment. The procedure involves priming and closing access doors, as well as, priming and arming the room. A system that enforces the procedure is broken down into access door and room clearance automata.

Figure 6.5 depicts the access door automaton — *accessDoorSys()*. The automaton keeps a *flag* variable which can either be set or cleared. When the *flag* (doorFlag) is set (line 7), it implies that the door has been primed and closed successfully, otherwise it is cleared (lines 8-9). The main automaton — *treatmentRoomSys()* — of the room clearance system is depicted

in Figure 6.8. The automaton also keeps a *flag* (roomFlag) variable to test whether the room is ready or not. When the *flag* (roomFlag) is set (line 12), it implies that the room has been primed and armed successfully, otherwise it is cleared. Sections 3.3 and 5.2.4 present more information about the system.

```
1  process accessDoorSys() {
2  state S0,S1,S2;
3  init  S0;
4  trans
5      S0 -> S1 {sync primeDoor?;},
6      S1 -> S0 {  },
7      S1 -> S2 {sync closeDoor?; assign doorFlag:=1;},
8      S2 -> S2 {sync primeDoor?; assign doorFlag:=0;},
9      S2 -> S0 {sync openDoor?; assign doorFlag:=0;};
10 }
```

Figure 6.5: UPPAAL source code for access door subsystem

6.2.5 Timed automaton for TCS interlocks

Figure 6.7 depicts an automaton for TCS interlock system — *tcsInterlockSys()*. The main purpose of the automaton is to check whether the interlocks overridden agree with the selected system mode. A *flag* (interlockFlag) is either assigned a value *true* (line 7), return by a function *isInterlockOk()*, if correct TCS interlocks are bypassed. Otherwise the flag is assigned a value *false* (line 9).

6.2.6 Timed automaton for accelerator control system

There are other important systems which are responsible for the extraction and insertion of the devices that control the beam. As shown in Figure 6.8, the functions of an automaton — *acceleratorSys()* — is to extract or insert Faraday Cup 1, Faraday Cup 2, Faraday Cup 10 and Faraday Cup 19 and Neutron Shutter. Sections 3.4 and 5.2.5 elaborate on the responsibilities of the system.

```
process treatmentRoomSys() {
state S0,S1,S2,S3,S4;
init  S0;
trans
    S1 -> S0 {guard !(cycle)&&!(doorFlag);},
    S4 -> S0 {guard !(cycle)&&!(doorFlag); assign roomFlag:=0;},
    S3 -> S0 {guard !(cycle)&&!(doorFlag);},
    ...
    S3 -> S1 {guard !(cycle)&&doorFlag; sync openBoom?;},
    S3 -> S4 {guard !(cycle)&&doorFlag; sync armRoom?;
              assign roomFlag:=1;},
    S2 -> S3 {guard !(cycle)&&(doorFlag); sync closeBoom?;
              assign roomFlag:=0;},
    S2 -> S0 {guard !(cycle)&&!(doorFlag);},
    S1 -> S2 {guard !(cycle); sync primeRoom?;},
    S2 -> S1 {guard !(cycle);},
    S0 -> S1 {guard !(cycle)&&doorFlag;};
}
```

Figure 6.6: UPPAAL source code for room clearance system

```
process tcsInterlockSys() {
...
state S0;
init  S0;
trans
  S0 -> S0{select id:int[0,1];guard !(cycle); sync clearInterlock[id]?;
           assign overridden[id]:=0,interlockFlag:= isInterlockOk();},
  S0 -> S0{select id:int[0,1];guard !(cycle); sync setInterlock[id]?;
           assign overridden[id]:=1,interlockFlag:= isInterlockOk();};
}
```

Figure 6.7: UPPAAL source code for TCS interlocks system

```
1  process acceleratorSys() {
2  ...
3  state  S0,S1,S2,S3,S4;
4  commit S1,S2,S3,S4;
5  init   S0;
6  trans
7      S0 -> S4 {guard sisFailure(); sync toAc?;},
8      S0 -> S0 {guard beamSwitchedOn(); sync toAc?; assign beamOnOutput();},
9      S0 -> S0 {guard extractAll(); sync toAc?; assign allOutput();},
10     S0 -> S2 {guard extractFC1019(); sync toAc?;},
11     S0 -> S1 {guard extractFC12NS(); sync toAc?;},
12     S0 -> S0 {guard defaultCtrl(); sync toAc?; assign defaultOutput();},
13     S1 -> S3 {  },
14     S1 -> S0 {assign FC12NSOutput();},
15     S2 -> S4 {  },
16     S2 -> S0 {assign FC1019Output();},
17     S3 -> S0 {assign failureOutput();},
18     S4 -> S3 {  };
19 }
```

Figure 6.8: UPPAAL source code for accelerator control system

6.2.7 Timed automaton for the SIS controller

The main purpose of the timed automaton — *sisControllerSys()* — is to evaluate safety conditions of the TCS using inputs received from four categories of interlocks, that is, input from therapy safety bus lines, input from room clearance system, input from interlocks all over the TCS and feedback inputs from accelerator control system. The timed automaton produces four control commands to the accelerator control system. The commands are generated in the functions such as *clinicalBeamOffCtrl()* and *FC12NSExtractionCtrl()* (line 9 and 13) and these commands are used to control Faraday Cup 1, Faraday Cup 2, Faraday Cup 10, Faraday Cup 19, Neutron shutter, Beam Gating and RF-Trip. A detailed description of the SIS is presented in Chapter 3.

```
process sisControllerSys() {
...
state S0,S2,S4,S5,S6,S7,S8,S9,S10,S11,S12;
init  S0;
trans
    S0 -> S0 {guard !(toClinicalBeamOff()); sync toCtrl?;},
    S0 -> S2 {guard toClinicalBeamOff(); sync toCtrl?;
              assign clinicalBeamOffCtrl();},
    S2 -> S0 {guard !(toClinicalBeamOff())&&!(toFailure());
              sync toCtrl?; assign notReadyCtrl();  },
    S2 -> S4 {guard toFC12NSExtraction(); sync toCtrl?;
              assign FC12NSExtractionCtrl();},
    S2 -> S12 {guard toFailure(); sync toCtrl?;},
...
}
```

Figure 6.9: UPPAAL source code for the SIS controller

6.3 Formal specification in TCTL

SIS is a real time system and UPPAAL is a perfect model checking tool for the system. UPPAAL's property specification language is a subset of TCTL (explained in Section 2.1.3) because it does not allow A[], E<>, A<> and E[] to contain one another. It supports only safety, reachability, and bounded liveness properties. The properties listed in Section 4.4 are written in UPPAAL specification language as follows:

1. A[] not deadlock
2. Let;

   $tr = treatmentRoomSys()$

   A[] (not tr.S4 and !(doorFlag) imply not tr.S4)
3. Let;

   $ti = tcsInterlockSys())$

A[]((sysMode==1 && (not ti.overridden[0]) && (not ti.overridden[1])) imply interlockFlag)

4. Let;

   $cs = cycleScan(), si = sisControllerSys()$, and $tr = treatmentRoomSys()$

   A[](cs.S4 and si.S4 imply tr.S4)

The results of the properties are discussed in Chapter 9.

Chapter 7

The SMV model checker

SMV (Symbolic Model Verifier) is model checking tool which was originally meant for verification of synchronous hardware circuits, but it has been applied to a number of systems including communication protocols. SMV was originally developed at Carnegie-Mellon University by Ken McMillan around 1992/1993 for his Ph.D. degree. The tool used in this thesis is referred to as Cadence SMV which is the extension of the original SMV by the same author Ken McMillian. Cadence SMV uses shared variables to communicate between processes and processes can communicate either synchronously or asynchronously. Cadence SMV has a number of features which makes it possible to verify large systems and the techniques include compositional verification, refinement verification, symmetry reduction, temporal case splitting, data type reduction and induction. These techniques are described in [41]. The tool also verifies models written in a modeling language called verilog and it can verify properties written in both LTL and CTL formulas. The model checking algorithm discussed in section 2.1.2 is basically the one used by SMV. The state space of systems modeled in SMV is represented by Reduced Binary Decision Diagrams (RBDD) [16]. This thesis is not intended to give complete introduction of the tool, interested readers are referred to SMV tutorial [41] or browse through the website `http://www.kenmcmillian.com/`.

7.1 An overview of SMV specification language

A Cadence SMV specification has a global structure which includes constructs such as processes, global and local variables. In this section the structure is outlined, the structure only includes constructs that are used in the modeling of SIS.

- *Module* is a self-contained set of states and transitions
- *Main* refers to the main module where the program starts executing.
- *Init* assigns initial values to the variables
- *Next* designates the next value of the variable
- *Case* is the same as branching conditions in programming languages like C/C++.
- *Non-determinism* is a choice from a set of values. For example, the following assignment, CONSOLE_ON_STATUS := {0,1}; then the value of CONSOLE_ON_STATUS is chosen arbitrarily from the set {0,1}.
- *Input* assigns actual parameters to the corresponding formal parameters.
- *Output* assigns formal parameters to the corresponding actual parameters.

7.2 The SIS model in SMV

In Chapter 6 the timed automata — *cycleScan(), supervisorySys(), therapySafetyBus(), accessDoorSys(), treatmentRoomSys(), acceleratorSys()* and *sisControllerSys()* — of the SIS are explained in relation to PROMELA processes discussed in Chapter 5. In this chapter, the same procedure is followed, that is, SMV modules are explained in relation to PROMELA processes presented in Chapter 5 and only important features of the modules are highlighted.

7.2.1 Cycle scan module

The cyclic program execution of the SIS is modeled as a module — *cycleScan(...)* — depicted in Figure 7.1. After the system has started, the program execution goes through three phases:

(1) requests inputs (line 7), (2) the execution control is given to the *sisControllerSys(...)* (line 8) process to send commands to the *acceleratorSys(...)* if there is any change detected from inputs. And (3) the execution control is given to the *acceleratorSys()* (line 9) process and it reacts by adjusting beam control devices accordingly.

```
1  MODULE cycleScan(...){
2     ...
3     next(csstate):=
4      case{
5        csstate = sysStoped & ssstate = sysStarted:sysStarted;
6        csstate = sysStarted & ssstate = sysStoped:sysStoped;
7        csstate = sysStarted & ssstate = sysStarted: requestStatus;
8        csstate = requestStatus:sisControllerSys;
9        csstate = sisControllerSys:acceleratorControlSys;
10       csstate = acceleratorControlSys:sysStarted;
11       default:csstate;
12     };
13    ...
14 }
```

Figure 7.1: Cycle scan module

7.2.2 Supervisory module

The detailed description of supervisory system is discussed in Chapter 3, Section 3.2 and Figure 7.2 depicts part of the system module — *supervisorySys(...)*. The module has arguments (line 1) that are the outputs of the process and only system mode output (line 3) is shown. All the outputs (system mode, system nozzle and bypassed interlocks) are non-deterministically assigned values, and line 7 shows an example of the system mode selected non-deterministically.

7.2.3 Therapy safety bus module

The main function of therapy safety bus is to facilitate a fast means of communication between TCS systems. The description of the bus is presented in Section 4.3.3 and Figure 7.3 depicts

```
1  MODULE supervisorySys(ssstate,csstate,ssMode,ssNozzle,override){
2    ...
3    OUTPUT ssMode:   {clinicalMode, physicsMode, testMode};
4    ...
5    next(ssMode):=
6      case{
7        ssstate = sysStarted: {clinicalMode,physicsMode,testMode};
8        default:ssMode;
9      };
10    ...
11   FAIRNESS running;
12 }
```

Figure 7.2: Supervisory system module

SMV source code for the process. The output of the module is a boolean array of twelve therapy safety bus lines (line 1). The true values of the lines are assigned non-deterministically (line 5).

```
1  MODULE therapySafetyBus(tsbstate,csstate,tsb){
2     ...
3    next(tsb[0]):=
4      case{
5        tsbstate = idle: {0,1};
6        default:tsb[0];
7      };
8    ...
9    FAIRNESS running;
10 }
```

Figure 7.3: Therapy safety bus module

7.2.4 Room clearance system module

There are defined steps for preparing the treatment vault for patient treatment and the room clearance system is responsible for ensuring that the steps are followed as required. Further details about the system are discussed in Chapter 3, Section 3.3 and Figures 7.4 and 7.5

present the SMV source code for the system modules — *accessDoorSys()* and *treatmentRoomSys(...)*, respectively.

The *accessDoorSys()* module is responsible for priming and closing the side doors (access doors), while the *treatmentRoomSys(...)* module is concerned with priming and arming the treatment room. The output of the module *accessDoorSys()* is the status of the access door (line 3) and if the door is primed and closed successfully (line 9), the treatment room can then be primed and armed, otherwise the room cannot be armed.

```
1  MODULE accessDoorSys(adstate,adstatus){
2    ...
3    OUTPUT adstatus: {dooropen, waiting, doorclosed};
4    ...
5    init(adstatus):= dooropen;
6    next(adstatus):=
7      case{
8        adstatus = dooropen: waiting;
9        adstatus = waiting:{dooropen,doorclosed};
10       adstatus = doorclosed:dooropen;
11       default:adstatus;
12     };
13   FAIRNESS running;
14 }
```

Figure 7.4: Access door system module

The main output (line 3) of the *treatmentRoomSys()* module is a *flag* variable, which tests whether the room is ready for treatment or not. The room can only be primed and armed if all other componets of the treatment room are working properly, in this case, a successful prime and closure of the access doors.

7.2.5 TCS interlock module

The responsibility of the TCS interlock system — presented in Section 4.3.5 — is to monitor all the interlocked devices of the TCS. An SMV module for the system is depicted in Figure 7.6 — *tcsInterlockSys(...)*. The module has one output (line 3), a *flag* variable, which is assigned

```
1  MODULE treatmentRoomSys(rcsstate,rcsstatus,adstatus,csstate,roomFlag){
2    ...
3    OUTPUT roomFlag: boolean;
4
5    ...
6    init(roomFlag):= 0;
7    ...
8    next(roomFlag):=
9      case{
10        rcsstate = idle: (rcsstatus = roomArmed) ? 1: 0;
11        default: roomFlag;
12     };
13   FAIRNESS running;
14 }
```

Figure 7.5: Room clearance system module

a value *true* if all the interlocks that are not supposed to be bypassed are working properly and a value *false* otherwise. To compute the required results, the module needs two inputs (lines 4—5): (1) a system mode and (2) a list of interlocks which should not be bypassed in that mode.

7.2.6 Accelerator system module

Accelerator control system is responsible for the extraction and insertion of the Faraday Cups as well as the Neutron Shutter. An SMV module — *acceleratorControlSys(...)* — for the system is depicted in Figure 7.7. These devices are used to control the beam, that is, either to extract or insert (lines 11 — 12) in a sequence of steps and then updates (15— 16) feedback inputs to SIS. More details about the description of the system is given in Sections 3.4.

7.2.7 The SIS controller module

SIS is responsible for the evaluation of the TCS interlocks and sending of control commands to the accelerator control system. An SMV module — *sisControllerSys(...)* — for the system is depicted in Figure 7.8. In the previous chapters it is already mentioned that the main

```
1  MODULE tcsInterlockSys(tcsistate,csstate,tiMode,tiOverride,interlockFlag){
2    ...
3    OUTPUT interlockFlag:boolean;
4    INPUT tiMode:{clinicalMode,physicsMode,testMode};
5    INPUT tiOverride: array 0..1 of boolean;
6    ...
7    next(interlockFlag):=
8      case{
9         tcsistate = idle & tiMode=clinicalMode:
10                    !(tiOverride[0] | tiOverride[1]) ? 1 : 0;
11        tcsistate = idle & tiMode=physicsMode: !(tiOverride[0]) ? 1 : 0;
12        tcsistate = idle & tiMode=testMode: !(tiOverride[1]) ? 1 : 0;
13        default:interlockFlag;
14     };
15   FAIRNESS running;
16 }
```

Figure 7.6: TCS interlock module

purpose of this module is to evaluate safety conditions (lines 7 — 14) of the TCS using inputs received from four categories of interlocks, i.e., input from TSB lines, input from RCS, input from TCS interlocks and feedback inputs from ACS. The module then produces four control commands to the accelerator control system (line 18) and these commands are used to control the beam stopping devices.

```
1  MODULE acceleratorControlSys(acstate,acstatus,csstate,dIn,dOut,FC12_NS,
2                               FC1019,RF_TRIP,BEAM_GATED){
3   OUTPUT acstate: {idle, readfbkStatus};
4   OUTPUT acstatus:{waiting, extractingNS, extractingFC1, extractingFC2,
5                 FC12NSout, extractingFC10, extractingFC19, insertingNS,
6                 insertingFC1, insertingFC2, insertingFC10, insertingFC19,
7                 beamOn};
8   ...
9   next(acstatus):=
10    case{
11      acstate = idle & acstatus = extractingNS:
12      acstate = idle & acstatus = insertingFC1:insertingNS;
13      ...
14    };
15  (next(dIn[0]), next(dOut[0]),next(dIn[1]), next(dOut[1]),next(dIn[2]),
16   next(dOut[2]),next(dIn[3]), next(dOut[3]),next(dIn[4]), next(dOut[4])):=
17    case{
18     acstate = idle & next(acstatus) = extractingNS : (1,0,1,0,1,0,1,0,1,0);
19     acstate = idle & next(acstatus) = insertingNS : (1,0,1,0,1,0,1,0,1,0);
20     ...
21   };
22  FAIRNESS running;
23 }
```

Figure 7.7: Accelerator system module

7.3 Formal specification in CTL

The CTL formula below corresponds to the untimed property described in Section 4.4. The verification results of the property in SMV are discussed in Chapter 9.

1. The deadlock property is not verified.

2. Let;

   $$ad = processaccessDoorSys(...), tr = treatmentRoomSys(...)$$

   assert G(((G !(ad.adstatus = doorclosed)) & (tr.rcsstatus = roomArmed) &
   (F !(tr.rcsstatus = roomArmed))) →

(tr.rcsstatus = roomArmed) U (G !(tr.rcsstatus = roomArmed)));

3. Let;

   $ti = tcsInterlockSys(...))$

   assert G((G !(ti.tiMode=clinicalMode & !(ti.tiOverride[0]) & !(ti.tiOverride[0])) &
   ti.interlockFlag & (F !(ti.interlockFlag))) →
   (ti.interlockFlag) U (G !(ti.interlockFlag)));

4. Let;

   $si = sisControllerSys(...)$

   assert G(((G !(inputlines[22])) & (si.sisstate = FC12NSextraction) &
   (F !(si.sisstate = FC12NSextraction))) →
   (si.sisstate = FC12NSextraction) U (G !(si.sisstate = FC12NSextraction)));

```
1  MODULE sisControllerSys(sisstate,FC12_NS,FC1019,RF_TRIP,BEAM_GATED,csstate,
2                          line,sysMode,sysNozzle){
3   ...
4   next(sisstate):=
5    case{
6      ...
7      csstate = sisControllerSys & sisstate = notReady & line[0] &
8      line[1] & line[2] & !(line[3]) & ((line[4])^(line[5])) &
9      (line[4] & sysNozzle = primaryNozzle | line[5] &
10     sysNozzle = secondaryNozzle) & line[9] & !(line[10]) &
11     !(line[11]) & sysMode = clinicalMode & line[12] &
12     !(line[13]) & line[14] & !(line[15]) & line[16] & !(line[17]) &
13     line[18] & !(line[19]) & line[20] & !(line[21]) & line[22] &
14     line[23]: clinicalBeamOff;
15     ...
16     default:failure;
17   };
18   (next(RF_TRIP),next(BEAM_GATED),next(FC12_NS),next(FC1019)):=
19     case{
20       csstate = sisControllerSys & next(sisstate) = notReady:(0,0,1,1);
21       csstate = sisControllerSys & next(sisstate) = physicsBeamOff:(0,0,1,1);
22       csstate = sisControllerSys & next(sisstate) = clinicalBeamOff:(0,0,1,1);
23       ...
24       default:(RF_TRIP,BEAM_GATED,FC12_NS,FC1019);
25    };
26   FAIRNESS running;
27  }
```

Figure 7.8: SIS controller module

Chapter 8

The PVS theorem prover

Three previous chapters — Chapters 5, 6, and 7 — presented SIS formal models in SPIN, UPPAAL, and SMV respectively. These tools implement model checking for verification of properties. However, model checking does not solve all the problems in formal analysis of systems, as Qingguo Xu and Huaikou Miao [62] state that, it is usually impossible to model check the entire system such as SIS due to the state explosion problem. They even added that for system specifications involving parameters, model checking can only check a subset of the domain value rather than the entire domain. This chapter therefore discusses the process of specifying and verifying SIS in a theorem prover called Prototype Verification System (PVS), which does not experience the problems mentioned above. The framework adopted is detailed in [62] and it is based on the theory of timed automata by Alur and Dill [1]. The specification of the SIS fits well in the framework as both safety and real-time liveness properties can be checked on the entire formal specification of the system. Section 8.1 discusses the process of formally specifying the system while Section 8.1.4 discusses a theorem for formalising safety properties of the system, all the PVS language features that are used are explained thoroughly to make the chapter self-contained.

8.1 Formal specification of SIS

The state of the SIS is a collection of attributes categorised in clock variables, therapy safety bus lines, hard wired lines, feedback input lines. These attributes determine whether the system is operating safely in one of its modes — *clinical*, *physics* and *test* modes. The SIS comprises of nine PVS theories, namely, *cycleScan, supervisorySys, therapySafetyBus, accessDoorSys, treatmentRoomSys, tcsInterlockSys, acceleratorSys, sisControllerSys,* and *tcsGlobal*. All the theories, except *tcsGlobal* and *tcsProduct*, correspond to the processes, automata and modules discussed in Chapters 5, 6, and 7 respectively, so their descriptions are not presented in this Chapter. However, since a different approach is followed in designing the SIS in PVS, Section 8.1.2 presents the *sisControllerSys* theory as an example of illustrating how other theories are implemented. Section 8.1.4 discusses the product theory — *tcsProduct*.

8.1.1 Global declarations

The description of SIS is divided into a collection of modules and each module is represented by a theory in PVS. The system *Actions, Locations* and *States* are shared in all the modules of the system and are declared in the following fragment of the theory *tcsGlobal*:

```
1  tcsGlobal: THEORY
2  BEGIN
3    N: posnat = 23
4    Time: TYPE+ = nonneg_real
5    ...
```

Figure 8.1: Global declaration theory

PVS provides a construct *Datatype* as a mechanism for defining abstract datatype and it includes *constructors, accessors* and *recognizers*. The fragment in Figure 8.2 depicts *Actions* abstract datatype with constructors *setTsb, clearTsb, toClinicalBeamOff*, etc., accessors *id:below[N]* and *del:posreal* as well as recognizers *setTsb?, clearTsb?, toClinicalBeamOff?*, etc. The datatype *Actions* is shared in all the theories and actions that do not occur in a certain theory are regarded as stuttering.

```
1    Actions: DATATYPE
2      BEGIN
3        ...
4        setTsb(id:below[N]):  setTsb?
5        clearTsb(id:below[N]):clearTsb?
6        ...
7        toClinicalBeamOff:toClinicalBeamOff?
8        toFC12NS:toFC12NS?
9        ...
10       delay(del:posreal):delay?
11     END Actions
```

Figure 8.2: The *Actions* datatype

In Pvs, a type *Locations* can be defined with values that are shared among the theories in the system.

```
1   Locations: TYPE+ = {clinicalBeamOff, physicsBeamOff, testBeamOff,
2                       FC12NSExtraction, FC1019Extraction, beamOn,
3                       beamGated, switchingMovementOn, movement,...}
```

Figure 8.3: The type *Locations*

The values (clinicalBeamOff, physicsBeamOff, testBeamOff, etc.) represent the locations of the timed automata — (*sisControllerSys*).

The *States* of the system are the attributes that represent the system's operations. The set of attributes that define the behaviour of SIS are defined as a record. In Pvs, the symbols [#...#] are records brackets and the *States* type is defined as follows:

```
1  States:TYPE = [#loc:Locations, x, now:Time,
2                 ...,
3                 roomFlag:bool,interlockFlag:bool,
4                 tsb:[below[12]->[#value:bool#]],
5                 ...
```

Figure 8.4: The type *States*

The record shows that the *state* of the system contains; location, local clock variables, global time and a record of input lines.

8.1.2 The SIS control theory in PVS

The inputs to the *sisControllerSys* are all lines which can either be *on* or *off*. In Pvs this can be represented by a vector *below[N]* which is a predefined Pvs type and it is the range of natural numbers [0...N-1]. The variable N represents all the input lines to the system. A compact way of representing the switching on and off of the lines is a function with domain *below[N]* and range *bool*, if a *tsb* is a vector then *tsb*(0) denotes the boolean value of the first line. The TYPE+ (unlike just TYPE) means that the type is non-empty and Pvs requires a proof that at least there is one element of the type. The following is an example of the function type for a vector representing the therapy safety bus lines to the system.

tsb: TYPE+ = [below[12] → [#value:bool#]]

8.1.3 Timed automata functions

The design of the predicates *Init, Pre,* and *Effect* are defined based on the timed automata framework presented in [62]. The *Init* function is used to define an initial state of the timed automata. For example, the *Init* function for the theory *sisControllerSys* is defined as follows;

Init(s):bool = loc(s) = notReady

Note that there are two forms of record access, that is, if *s* is of type *States*, *loc* field maybe accessed using either *loc(s)* or *s'loc*.

A Precondition function

The precondition function (*Pre*) is defined on each action of the timed automata to check whether an action is enabled or not. In Figure 8.5 for instance, at location *notReady* an automaton can make an infinite delay transition while at location *FC12NSExtraction* the system can only stay for less than 10 units of time. The automaton can make a switch

transition from *notReady* location to *clinicalBeamOff* location if the system is at the location *notReady* and the edge between these locations is enabled.

```
1   Pre(a)(s):bool =
2     CASES a OF
3      delay(d):
4        IF NOT(loc(s) = FC12NSExtraction & loc(s) = FC1019Extraction) THEN
5           TRUE
6        ELSE
7           x(s)+d < 10
8        ENDIF,
9
10     toNotReady:
11      loc(s) = clinicalBeamOff & NOT((s‘tsb(0)‘value) & (s‘tsb(1)‘value) &
12      (s‘tsb(2)‘value) & (s‘tsb(3)‘value & NOT s‘tsb(4)‘value &
13      s‘nozzle = primaryNozzle) OR (NOT s‘tsb(3)‘value & s‘tsb(4)‘value &
14      s‘nozzle = secondaryNozzle) & (NOT s‘tsb(5)‘value) &
15      (NOT s‘tsb(6)‘value) & (NOT s‘tsb(7)‘value) & (NOT s‘tsb(8)‘value) &
16      (s‘tsb(9)‘value) & (NOT s‘tsb(10)‘value) & (NOT s‘tsb(11)‘value) &
17      (s‘fbk(0)‘value) & (NOT s‘fbk(1)‘value) & (s‘fbk(2)‘value) &
18      (NOT s‘fbk(3)‘value) & (s‘fbk(4)‘value) & (NOT s‘fbk(5)‘value) &
19      (s‘fbk(6)‘value) & (NOT s‘fbk(7)‘value) & (s‘fbk(8)‘value) &
20      (NOT s‘fbk(9)‘value) & s‘roomFlag & s‘interlockFlag),
21      ...
22    ENDCASES
```

Figure 8.5: Precondition function

An Effect function

The Effect function (*Effect*), depicted in Figure 8.6, is also defined on each action of the timed automata. The function updates the values of the record *States* if the precondition for an action holds.

8.1.4 The product automaton of the system

In order to be able to verify the properties of the system, a product timed automaton is defined. The automaton also comprises of three functions *Init, Pre* and *Effect* and each of the functions is defined as a product of the corresponding functions, for example, the *Init* function

```
1  Effect(s0,a,s1):bool = CASES a OF
2    delay(d):s1 = s0 WITH[x := x(s0)+d,now:=now(s0)+d],
3    toNotReady: s1 = s0 WITH[loc := notReady, RF_Trip:=FALSE,
4                         BeamGated:=FALSE, FC12NS:=TRUE, FC1019:=TRUE],
5    ...
6    ELSE
7       s1=s0
8   ENDCASES
```

Figure 8.6: Effect function

is defined as a product of *Init* functions defined in other theories of the system — *cycleScan, supervisorySys, therapySafetyBus, accessDoorSys, treatmentRoomSys, tcsInterlockSys, acceleratorSys, sisControllerSys,* and *tcsGlobal*.

```
1  tcsProduct: THEORY
2  BEGIN
3   IMPORTING cycleScan AS cs
4   IMPORTING supervisorySys AS ss
5   IMPORTING therapySafetyBus AS ts
6   IMPORTING accessDoorSys AS ad
7   IMPORTING treatmentRoomSys AS rc
8   IMPORTING tcsInterlockSys AS ti
9   IMPORTING acceleratorSys AS ac
10  IMPORTING sisControllerSys AS si
11  IMPORTING tcsGlobal AS G
12  ...
13  Init(s):bool = cs.Init(s) & ss.Init(s) & ts.Init(s) & ad.Init(s) &
14                 rc.Init(s) & ti.Init(s) & ac.Init(s) & si.Init(s)
15  ...
```

Figure 8.7: First part of the product theory

To complete the product automaton, a template theory Timed automata is imported to define the behaviour of the system - SIS.

```
1   ...
2   IMPORTING TimedAutomata[States, Actions, Init, Pre, Effect, now]
3  END tcsProduct
```

Figure 8.8: Last part of the product theory

8.2 Theorem proving

In Chapters 5, 6, and 7 the selected properties are formalized in LTL, TCTL and CTL respectively. In this section the property similar to property four (4. ***Property:***) in Section 4.4 is verified with PVS. In Figure 8.9, *safe* is the safety property which must hold at the initial state of the product automaton as well as in the subsequent states of the automaton. The results of the proof are discussed in Chapter 9.

```
1  safe(s): bool =
2     (((s‘tsb(0)‘value) & (s‘tsb(1)‘value) & (s‘tsb(2)‘value) &
3     (s‘tsb(3)‘value & NOT s‘tsb(4)‘value &
4     s‘nozzle = primaryNozzle) OR (NOT s‘tsb(3)‘value &
5     s‘tsb(4)‘value & s‘nozzle = secondaryNozzle) &
6     (NOT s‘tsb(5)‘value) & (s‘tsb(6)‘value) & (s‘tsb(7)‘value) &
7     (s‘tsb(8)‘value) & (s‘tsb(9)‘value) & (NOT s‘tsb(10)‘value) &
8     (NOT s‘tsb(11)‘value) & (NOT s‘fbk(0)‘value) & (s‘fbk(1)‘value) &
9     (NOT s‘fbk(2)‘value) & (s‘fbk(3)‘value) & (NOT s‘fbk(4)‘value) &
10    (s‘fbk(5)‘value) & (s‘fbk(6)‘value) & (NOT s‘fbk(7)‘value) &
11    (s‘fbk(8)‘value) & (NOT s‘fbk(9)‘value) & s‘roomFlag &
12    s‘interlockFlag) =>
13                  loc(s) = FC12NSExtraction)
14
15 Safety: THEOREM FORALL r, i: LET s = states(r)(i) IN safe(s)
```

Figure 8.9: Theorem for the safety property

Chapter 9

Discussion

This chapter discusses the comparison of the model checking tools: SPIN, UPPAAL, and SMV together with a theorem prover PVS presented in Chapters 5, 6, 7 and 8, respectively. In spite all the differences of verification techniques, every verification tool has its strengths and weaknesses. And it should, therefore, be the goal of everyone in the formal methods community to realise all these strengths and weaknesses, so that, all verification tools can deal with each other and join forces to combine their strengths. Fortunately, there are signs that this time is coming as some softwere engineers and hardware industries are becoming more interested in formal methods and are also supporting joint projects with academics — this thesis presents an example of such projects. However, the problem is that, the interest is more based in single case studies rather than comparison between verification tools. Comparison among verification tools is the area of study where people in formal methods community should pay a lot of attention [5] as this will ease the use of verification tools in the long run.

It should be noted that in this study more attention is on the verification tools rather than underlying verification algorithms implemented by the tools. The attention is paid to the tools mainly because they are the ones being used by the practitioners for verification of systems. The procedure for comparing these methods is divided into two phases. The first phase involves the time logging of activities followed by a novice modeler to model check and theorem prove software systems, in particular, the SIS at iThemba LABS. This is referred to as understandability analysis as it involves understanding activities involved in model

checking and theorem proving of software systems. The activities include learning verification tools, modelling the system and verifying safety properties of the system. These results are summarised in Table 9.1 and the details of each activity is elaborated in the subsequent sections. The second phase involves the performance of the tools in relation to time taken to verify a property, memory usage, number of states and transitions generated during the verification process.

9.1 Understandability analysis

This section presents the percentage effort for each activity. The work involving these activities spreads over approximately 18 months. Table 9.1 depicts the activities, effort percentage and general comments on each activity. It should be noted that the time spent in learning the system, that is, the SIS, is not included in the effort analysis as the attention is on the verification tools.

9.1.1 Learning modelling languages and tools

The first objective was to learn the modelling languages of the tools and how to use them to write abstract models. All the documents about the tools listed in the bibliography were thoroughly reviewed and some examples relating to the tools were completed. The modelling languages for SPIN and SMV are similar to the C programming language and the expectation is that any individual who has C programming experience and some basic knowledge of finite state machines will find the tools straightforward and easy to learn. Another feature which makes both tools easy to understand are graphical user interfaces they provide, which also include simulation environments. Specifications written in UPPAAL are in timed automata which are comparatively easy to understand and model. The tool provides a good graphical user interface and it accepts both hybrid and timed automata inputs. The tool also provides a good simulation environment and it has shown to be applied to a number of large and complex real-time verification problems. It is expected that anyone with the understanding of the theory of timed automata by Alur and Dill [1] will be able to use the tool to model real-time systems. However, a learner without prior knowledge of the theory is expected to

Table 9.1: Percentage time for each activity

Tools	Activity	Effort(%)	Comments
SPIN	Learning PROMELA	5%	Familiar with the tool, Section 2.1.1
	Modelling	9%	This refers to Chapter 5, Section 5.2
	Verification	3%	This refers to Chapter 5, Section 5.3
UPPAAL	Learning TA	12%	Not familiar with the tool, Section 2.1.3
	Modelling	8%	This refers to Chapter 6, Section 6.2
	Verification	3%	This refers to Chapter 6, Section 6.3
SMV	Learning SMV	10%	Not familiar with the tool, Section 2.1.2
	Modelling	6%	This refers to Chapter 7, Section 7.2
	Verification	3%	This refers to Chapter 7, Section 7.3
PVS	Learning PVS	15%	Not familiar with the tool, Section 2.2
	Modelling	13%	This refers to Chapter 8, Section 8.1
	Verification	13%	This refers to Chapter 8, Section 8.2
TOTAL		100%	

take longer time compared to those already familiar with the theory. As shown in Table 9.1, the modeler took longer time to be able to use UPPAAL as compared to other modelling tools since (s)he was not familiar with the theory. Comparatively the theorem prover PVS was the most difficult to understand and learning the tool took almost the same amount of time to learn the tools: SPIN, UPPAAL and SMV. It is expected that learning the tools one has to be comfortable with discrete mathematics and logic.

9.1.2 Modelling the system

The goal of this activity was to generate models of the system written in SPIN, UPPAAL, SMV and PVS. The modeler was already familiar with SPIN/PROMELA from the model checking course, so the system was first modelled in SPIN/PROMELA. Modeling of the system was a very iterative process as the modeler had to go back to the designers of the specification to clarify some of the ambiguities identified during modelling process. The state explotion problem was an obstacle in making a quick progress to other tools as the modeler wanted to use the PROMELA model as the basis to generate other models. The model was kept module to facilitate changes and when it was felt that the model has reached a fair level of stability, it was manually converted to other modelling languages. The conversion to UPPAAL and SMV was easy, but the conversion to PVS was difficult. With the help of verification framework in [62], it was then easy to generate a PVS model as the modeler already understood theory of timed automata. The observation is that converting a PROMELA model to UPPAAL is comparatively easier than converting the model to both SMV and PVS. However, the conversion from UPPAAL to PVS was much easier than from PROMELA to PVS and this is due to the framework proposed in [62]. SPIN and UPPAAL were good at modelling a synchronised communication between processes. It seemed natural to model the system with SMV as it is good at modelling hardware systems, that is, the system involving bus and hard-wired communication. PVS was good as it did not suffer from state explotion problem when adding clock variables to the model and it is observed to be the ideal tool to model the entire system, maybe in conjunction with UPPAAL so as to avoid errors which maybe caused by error-prone nature of theorem provers, in this case PVS.

9.1.3 Verifying the system

The objective of this activity was to identify safety properties of the system and express them in verification languages such as Linear Temporal Logic (LTL), Computational Tree Logic (CTL), Timed Computational Tree Logic (TCTL) and High Order Logic (HOL). The modeler was only familiar with LTL and had to learn other types of logics used in this study. LTL and CTL are comparatively the easiest to understand and apply. These property specification

languages are presented as boolean formulas on the system safety properties. TCTL is the same as CTL except that formulas are extended with clock variables. In addition to these temporal logics, PVS HOL was also learned. Some expertise is required as this logic is very expressive and thus extreme care is needed for formulating property claims correctly.

9.2 The case study

The second phase to understandability is the case study results presented in Tables 9.2 — 9.4. Tools are compared in terms of number of states, number of transitions, memory usage and time taken to verify a property. However, it must be noted that the models are not necessarily the same, the aim was to develop models in an efficient way for all the tools, so that it can be concluded as to which of the tools can best model the system in question.

9.2.1 Verification results

Verification process was done on a personal computer with 1024MB of RAM and speed of 1.5 MHz. The results for SPIN, UPPAAL, and SMV are shown in Tables 9.2, 9.3, and 9.4, respectively. The results are explained based on two factors: (1) the configuration parameters of the tool, and (2) the number interlocks considered in each model. In SPIN the supertrace search mode is used with patial-order reduction in order to cover a large state space of the model and there are 18 total number of interlocks (boolean variables) manipulated by the controller process — *sisControllerSys* — of the SPIN model.

Table 9.2: SPIN Verification results

Properties	States(m)	Transitions(m)	Memory(MB)	Time(sec)
$\boldsymbol{Property_1}$	3.09	97.94	108.85	222.66
$\boldsymbol{Property_2}$	3.44	10.01	209.65	48.85
$\boldsymbol{Property_3}$	2.96	10.10	182.56	468.85
$\boldsymbol{Property_4}$	2.95	87.87	444.77	1607.8

In UPPAAL the bit-state hashing with conservative state space reduction algorithm is used to represent the state space of the model. The breadth first search order is used as this is the most efficient option when the complete state space must be searched. There are 9 total number of interlocks manipulated by the controller process in UPPAAL model.

Table 9.3: UPPAAL Verification results

Properties	States explored	Memory(MB)	Time(sec)
$\boldsymbol{Property}_1$	$1,968,214$	509.75	203.74
$\boldsymbol{Property}_2$	$1,969,912$	508.82	91.94
$\boldsymbol{Property}_3$	$1,957,387$	482.70	92.23
$\boldsymbol{Property}_4$	$1,957,321$	482.97	90.98

Verification of SMV model did not experience any state explosion problem (like SPIN and UPPAAL) and there are 24 interlocks that are manipulated by the corresponding controller module. The state reduction techniques: compositional verification, refinement verification, symmetry reduction, temporal case splitting, data type reduction and induction [41] were not even considered.

Table 9.4: SMV Verification results

Properties	BDD nodes	Memory(MB)	Time(sec)
$\boldsymbol{Property}_2$	$10,010$	5.90	0.21
$\boldsymbol{Property}_3$	$30,097$	6.34	0.33
$\boldsymbol{Property}_4$	$521,906$	$16,21$	24.29

In all the properties verified, the results show that SMV has generated fewer number of states (nodes) compared to SPIN and UPPAAL. The difference maybe attributed to the fact that SMV was developed to verify hardware systems, that is, SIS involves TSB and hard wired lines which are directly declared with constructs such as *input* and *output* in SMV. On the other hand SPIN and UPPAAL are developed to verify software systems and they are likely

to perform badly for hardware systems involving many hard wired lines. It took a fewer seconds to verify the property with SMV than it took with SPIN and UPPAAL due to the size of state space generated by the respective tools. However, looking only at SPIN and UPPAAL, SPIN has performed better than UPPAAL and this maybe attributed to the combination of supertrace search mode and partial-order reduction technique.

Table 9.5: PVS Verification results

	First Run(Real time)	**Re-Run(Real time)**
Property	10.173	0.964

PVS is treated differently from the model checking tools — SPIN, UPPAAL, and SMV — as it does not suffer from the problem of state space explotion and only the real time of the verification is considered. It takes a longer time to prove the property with PVS for the first time, but takes fewer seconds when the proof is rerun. Table 9.5 shows the results of the proof.

Chapter 10

Conclusion

This thesis presents a comparative case study between three model checking tools: SPIN, UPPAAL and SMV as well as a theorem prover PVS. The case study is conducted with the SIS at iThemba LABS and the specification of the system is presented in Chapter 3. The criterion used for the selection of these tools is discussed in Chapter 4. Chapters 5 — 8 summarise the features of each tool and the models developed in SPIN, UPPAAL, SMV and PVS, respectively. There are two main goals to this study, the first one is to compare the tools so that some important observations can be outlined to assist software engineering practitioners with the easy selection of a tool for specifying and verifying systems similar to safety interlock system (SIS). Section 10.1 below summarises some observations derived from the discussion presented in Chapter 9 and Section 10.2 proposes some future work.

10.1 Observations

It must be emphasised that the aim of the study is to bring out the pro's and con's of each tool in relation to the SIS and hence provide insight and understanding of the tool. The comparison does not necessarily indicate that one tool is better than the other. In fact the tools are complementary not competing. Final remarks are then outlined as follows:

- It was observed that representing the system in graphical state machines was found

useful for three reasons. First, graphical notation facilitate communication with the other designers and other stakeholders, help in learning the system, and simple to use and follow. Second, modular graphical structure facilitate an easy changing of the system. Third, the conversion from state machines to any of the tools was easy.

- Learning the model checking tools was easy as the modeler was familiar with C programming. However, the theorem prover Pvs was more difficult and needed some expertise and experience. Specifying system properties was also challenging in the sense that extreme care was needed to make sure that a formalised property represents what is actually meant. The exercise requires some expertise and experience. Once the properties were stated, checking them was done in seconds.

10.2 Recommendations and Future work

The SIS presented in this study is not complete. Only clinical mode was considered in detail. The next step is to expand the models, incoperating physics and tests modes. However, this phase has already been taken care of as the models are designed in such a way that it would be easy to expand without changing the design.

It is recommended that the combination of Uppaal and Pvs can be an ideal when expanding the model as the system is a real-time system and both tools are good at modeling such systems. In order to be able to use Uppaal effectively, a computer with a maximum RAM available is needed.

Bibliography

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] P. Appelhaus and S. Engell. Design and implementation of an extended observer for the polymerization of polyethylenterephthalate. *Chemical Engineering Science*, 51(10):1919–1926, 1996.

[3] Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. A framework for evaluating specification methods for reactive systems: Experience report. *IEEE Transactions on Software Engineering*, 22(6):378–389, 1996.

[4] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *Software Engineering*, 17(11):1204–1222, 1991.

[5] George S. Avrunin, James C. Corbett, and Matthew B. Dwyer. Benchmarking finite-state verifiers. *International Journal on Software Tools for Technology Transfer*, 2(4):317–320, 2000.

[6] George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Pâsâreanu, and Stephen F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts, November 1999.

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[8] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[9] B. Bérard and L. Sierra. Comparing verification with HYTECH, KRONOS and UPPAAL on the railroad crossing example. Technical Report LSV–00–2, Laboratoire Spécification et Vérification, ENS Cachan, 2000.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag.

[11] J. P. Bowen and M. G. Hinchey. The use of industrial-strength formal methods. *IEEE Computer Society Press*, pages 332–337, 13–15 August 1997.

[12] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Softw.*, 12(4):34–41, 1995.

[13] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[14] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[15] R. Boyer and J Moore. *A Computational Logic*. Academic Press, 1979.

[16] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[17] J. R. Burch, E. R. Clarke, K. L. McMillan, D. L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[18] A. T. Chamillard, Lori A. Clarke, and George S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report TR-96-84, Department of Computer Science, University of Massachusetts, 1996. Revised May 1997.

[19] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. Technical Report 96-04-02, Department of Computer Science and Engineering, University of Washington, 1996.

[20] Alessandro Cimatti, Fausto Giunchiglia, Giorgio Mongardi, Dario Romano, Fernando Torielli, and Paolo Traverso. Model checking safety critical software with SPIN: An application to a railway interlocking system. In *SAFECOMP '98: Proceedings of the 17th International Conference on Computer Safety, Reliability and Security*, pages 284–295, London, UK, 1998. Springer-Verlag.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[22] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33:61–67, 1996. Invited article.

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

[24] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[25] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[26] A. J. Currie. A comparison of three model checkers applied to a distributed database problem. In *Proceedings of the 4th Irish Workshop on Formal Methods (IWFM)*, 2000.

[27] Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In E. L. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, number 1275 in LNCS, pages 89–104, Murray Hill, New Jersey, 1997. Springer-Verlag.

[28] Yifei Dong, Xiaogun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David Scott Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 74–88, 1999.

[29] Yifei Dong, Xiaoqun Du, Gerard J. Holzmann, and Scott A. Smolka. Fighting livelock in the gnu i-protocol: A case study in explicit-state model checking. *International Journal on Software Tools for Technology Transfer*, 4(4):505–528, 2003.

[30] Jr Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, London England, 1st edition, 1999.

[31] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[32] Peter R. Glück and Gerard J. Holzmann. Using Spin model checking for flight software verification. In *Proceedings of the 2002 Aerospace Conference*, pages 1–105–1–113. IEEE, 2002.

[33] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An approach to the State-Explosion Problem.*, volume LNCS 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[34] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 438–449, London, UK, 1993. Springer-Verlag.

[35] M. J. C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[36] Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.

[37] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[38] Gerard J. Holzmann. The engineering of a model checker: The gnu i-protocol case study revisited. In *Proceedings of the 5th and 6th International* SPIN *Workshops on Theoretical and Practical Aspects of* SPIN *Model Checking*, pages 232–244, 1999.

[39] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Lucent Technologies Inc. Bell Laboratories, 2004.

[40] Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. In *In Proceedings of the 2nd* SPIN *Workshop.*, University, New Jersey, USA, aug 1996.

[41] McMillan K. L. Getting started with SMV, March 1999.

[42] Joost-Pieter Katoen. Principles of model checking. Lecture Notes for 2001-2002.

[43] Joost-Pieter Katoen. Concepts, algorithms, and tools for model checking, 1998. Lecture Notes of the course Mechanized Validation of Parallel Systems.

[44] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[45] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *ICSE '89: Proceedings of the 11th international conference on Software engineering*, pages 44–52, New York, NY, USA, 1989. ACM Press.

[46] Petr Matousek. Protocol proving using PVS: A case study. In *Proceedings of the 35th Spring Conference: Modelling and Simulation of Systems - MOSIS'01*, pages 67–73, 2001.

[47] Kenneth L McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993.

[48] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS *Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[49] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS *Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[50] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS *System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, November 2001.

[51] Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. Assume-guarantee model checking of software: A comparative case study. In *Proceedings of the 5th and 6th International* SPIN *Workshops on Theoretical and Practical Aspects of* SPIN *Model Checking*, pages 168–183, 1999.

[52] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[53] Theo C. Ruys. Towards effective model checking, 2001.

[54] Sol M. Shatz, Shengru Tu, Tadao Murata, and Sastry Duri. An application of Petri net reduction for ada tasking deadlock analysis. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1307–1322, 1996.

[55] Mariëlle Stoelinga. Fun with FireWire: A comparative study of formal verification methods applied to the IEEE 1394 root contention protocol. *Formal Aspects of Computing*, 14(3):328–337, 2003.

[56] Richard N. Taylor, Kari A. Nies, Gregory Alan Bolcer, Craig A. MacFarlane, Kenneth M. Anderson, and Gregory F. Johnson. Chiron-1: a software architecture for user interface development, maintenance, and run-time support. *ACM Trans. Comput.-Hum. Interact.*, 2(2):105–144, 1995.

[57] Antti Valmari. A stubborn attack on state explosion. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165, London, UK, 1991. Springer-Verlag.

[58] Antti Valmari. The state explosion problem. *LNCS: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[59] Christopher van Tubbergh and Evan A de Kock. Conceptual design of the proton beam treatment facilty at iThemba LABS, 2005. DRAFT COPY Version 1.00.

[60] Libor Waszniowski. *Formal Verification of Multitasking Applications Based on a Timed Automata Model*. PhD thesis, Czech Technical University in Prague, 2006.

[61] Jeannette M. Wing and Mandana Vaziri-Farahani. A case study in model checking software systems. Technical Report CMU-CS-96-124, Department of Computer Science, Carnegie Mellon University, 1996.

[62] Qingguo Xu and Huaikou Miao. Formal verification framework for safety of real-time system based on timed automata model in PVS. In *IASTED Conf. on Software Engineering*, pages 107–112, 2006.

[63] Qingguo Xu and Huaikou Miao. Modeling timed automata theory in PVS. In *Software Engineering Research and Practice*, pages 205–211, 2006.

[64] Sergio Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.

[65] Wenhui Zhang. Model checking operator procedures. In *Proceedings of the 5th and 6th International* SPIN *Workshops on Theoretical and Practical Aspects of* SPIN *Model Checking*, pages 200–215, London, UK, 1999. Springer-Verlag.

Bibliographic crossreference

[1] 24, 49, 91, 99

[2] 47

[3] 28

[4] 28

[5] 1, 2, 7, 28, 98

[6] 28, 30, 47

[7] 71

[8] 25, 26

[9] 28, 30

[10] 17

[11] 1, 7

[12] 1, 7

[13] 1, 7

[14] 1, 7

[15] 27

[16] 5, 81

[17] 17

[18] 28, 29, 47

[19] 47

[20] 47

[21] 17

[22] 1, 7, 47

[23] 7, 17

[24] 1, 7, 47

[25] 28, 29, 47

[26] 28, 47

[27] 28

[28] 28, 29, 47

[29] 28, 29, 47

[30] 25, 26

[31] 14

[32] 49

[33] 28

[34] 28

[35] 27

[36] 1, 7

[37] xiv, 59, 60

[38] 29, 47

[39] 14, 47, 59

[40] 28, 30

[41] 81, 103

[42] 23, 25

[43] 6

[44] xiv, 71, 72

[45] 28

[46] 47

[47] 17

[48] 27

[49] 27

[50] 27

[51] 28, 47

[52] 7

[53] 7

[54] 28

[55] 28, 47

[56] 30

[57] 28

[58] 28

[59] 2, 66

[60] 26

[61] 47

[62] 91, 94, 101

[63] 27

[64] 25

[65] 47