Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 73
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 73
Espoo 2002 HUT-TCS-A73

BOUNDED MODEL CHECKING FOR VERIFYING CONCURRENT PROGRAMS

Toni Jussila

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 73
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 73
Espoo 2002 HUT-TCS-A73

BOUNDED MODEL CHECKING FOR VERIFYING CONCURRENT PROGRAMS

Toni Jussila

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-6063-8
ISSN 1457-7615

Otamedia Oy
Espoo 2002

Abstract: A simple, parallel programming language is introduced and an operational semantics for it is developed. The language combines the syntax of C and Java together with the communication primitives taken from PROMELA.

A verification method for specifications given in the language is developed for detecting the violation of temporal reachability and safety properties. The method is known as Bounded Model Checking (BMC) where the idea is to reduce the model checking problem to propositional satisfiability.

A compact boolean encoding of parallel programs is devised, together with the proofs of its soundness and completeness. Encoding of the reachability and safety properties is developed and finally semantical models for strengthening the encoding are discussed.

Keywords: bounded model checking, verification, concurrent program

CONTENTS

1 INTRODUCTION

Semiconductor packaging densities and speeds tend to follow the trend defined by Moore's law [27]. At the same time, the human productivity in software and hardware design has only enjoyed a more modest growth rate. This has caused a situation in which the ability to relegate even more complex tasks to computers is not limited by computing speeds or memory capacities but rather the ability to design and implement complex systems with sufficiently high degree of confidence on their correctness.

Hardware and software systems are nowadays used in places where failure or down time are not options, e.g., air traffic control systems and medical equipment. Therefore, a growing amount of time and effort has to be invested to ensuring correctness. In addition, the costs of correcting errors grow rapidly the later in the design process they are found. This is the problem of *design validation* and it is a major challenge for the industry [11].

Today, most industrial key players still practise simulation and testing as the main method for design validation. Even though effective in the early stages of the development, subtle bugs requiring improbable but possible interactions and/or long operating times may escape the testing sieve. Furthermore, testing only guarantees that the demands in the set of chosen test cases are met, not the more general notion that a certain property is provably true. Similarly, simulation can rarely guarantee that bugs may not remain in the product.

Formal methods provide an attractive alternative to the traditional methods. Where in simulation and testing some of the possible behaviors of the system are explored, formal verification methods exhaustively search all possible execution paths for a violation of a property. Thus, assuming that the verification is correctly conducted, a positive answer is a guarantee for the verified property to hold.

Whereas testing is normally conducted on the actual product, formal methods operate on abstract models, typically state-transition systems. The principal formal methods used are deductive verification and model checking. Deductive verification uses axiom systems and inference rules to prove properties of the abstraction. Its successful application requires expertise in logical reasoning and considerable experience. Furthermore, it cannot be easily automated even though software tools can be used to propose ways of proceeding from the current stage of the proof and to ensure that the proofs obtained are correct. Deductive verification has the benefit that it is possible to reason about potentially infinite state spaces. Currently, the method is only used in critical systems, in which enough resources can be invested for the verification of correctness.

Model checking is most commonly applied to finite state systems. The artifact being analyzed is modelled as a mathematical structure consisting of states and a transition relation between pairs of states (Kripke structures). The restriction to finite state spaces carries the benefit that model checking can be largely automated and even implemented using algorithms with reasonable efficiency. Furthermore, the limitation to finite state spaces is not too restricting, since many interesting applications, like hardware controllers and communication protocols remain amenable to analysis. Finally, in case of a property violation, model checking gives a counterexample, i.e. a path conflicting with the desired property, helping the verification engineer in locating the source of the problem.

As this report is about a model checking technique, the central development steps of the field are traced more minutiously in what follows. The quintessential problem is the explosion of the state space, i.e. systems with realistic complexity create state spaces so large that even with Moore's law present, the memory capacities of computers are not large enough to store them. The steps outlined below are ways of handling the problem and pushing the limit of what is possible further.

1.1 EXPLICIT STATE REPRESENTATIONS

The first model checking algorithms were independently proposed in the early 1980s by Clarke and Emerson [10] in the USA and Quelle and Sifakis in France [11]. They presented an algorithm for verifying properties expressed in computation tree logic (CTL, see Chapter 4). An implementation of the algorithm was provided in a tool named Extended Model Checker (EMC). The state space in the models extracted from the specifications were represented using adjacency lists and the system was able to verify state graphs with up to 10^5 states. This together with the speed of 100 states per second are modest figures with today's standards. However, the system has been used to detect several previously unknown errors in published circuit designs [11].

The model checker SPIN, released in 1991 by Gerald Holzmann, is a tool for specifying systems having concurrent components that can communicate via shared variables and message buffers. Its exhaustive search mode is also based on the explicit representation of states. However, the tool also contains a so called supertrace mode, in which the states are represented using hash values. Using it, exhaustiveness is of course sacrificed, but with sufficiently large hash tables a high coverage is attainable. The SPIN model checker has been used in the verification of many real products and it also provides the basis for a lot of research within the model checking community. One contributing factor to this may be that the design choices and the actual implementation are well documented in [16]. The implementation of the ideas presented in this report is also strongly influenced by the SPIN documentation.

1.2 SYMBOLIC METHOD

By 1987 one million states was the practical limit for the state spaces suitable for model checking. In fact, many formal methods researchers predicted that the method would never be practical for large circuits and protocols [26]. Much larger state spaces became feasible, when Kenneth McMillan, then a graduate student at Carnegie Mellon University, devised a more compact representation for the state transition graphs used in the models. Instead of explicit adjacency lists the transition relation could be encoded symbolically as a boolean formula. Furthermore, *Ordered Binary Decision Diagrams* (OBDDs), provide a compact form to represent them and OBDDs can be efficiently manipulated [7].

McMillan implemented the SMV model checker based on the above idea as part of his doctoral dissertation. An SMV program is a description of hierarchical communicating finite state machines together with the specifications to be verified. The system extracts the symbolic representation of the state-transition system and uses OBDD based search algorithms to check whether the specifications are met. Based on the initial ideas, it was possible to verify state spaces up to 10^{20} and with various refinements, the limit was pushed up to 10^{120} states [11].

1.3 BOUNDED MODEL CHECKING

Though being a major improvement, problems remained in the BDD based approach as well. When systems grew larger, the BDDs also became too large for available computers. In addition, the size of the BDDs depends heavily on a suitable variable ordering, the generation of which may consume a lot of time and require manual intervention. The practical limits of the BDD model checkers lie in hundreds of state variables.

In 1999 Biere, Cimatti, Clarke and Zhu presented an another approach to the problem [3]. The technique, coined as *Bounded Model Checking* (BMC), reduces the model checking problem to propositional satisfiability (SAT). The translation is such that the propositional formula has a model iff the system model has a behavior violating the property to be verified. The term bounded refers to the fact that the length of the counterexamples is limited by a parameter k. The authors show that BMC for Linear Temporal Logic (LTL) can be reduced to propositional satisfiability in polynomial time.

Bounded model checking has several important advantages. Firstly, the best SAT solvers are capable of handling thousands of state variables. Secondly, the ordering of variables is not of crucial importance, usually default splitting heuristics suffices. Thirdly, due to the depth first nature of SAT search procedures counterexamples are often found very fast. In addition, they are of minimal length. And finally, BMC requires less space than the BDD based approach [3].

The disadvantage with the technique is that with the limit k, completeness is naturally sacrificed. That is, a positive answer stating that the desired property holds is not conlusive, since the violating behavior could be detected, if the bound was increased.

BMC has already been applied to finding bugs in the Power PC and Alpha microprocessor designs [4, 5] with encouraging results. This report uses the same idea, but the application domain is a small parallel programming language.

1.4 EXPRESSING PROPERTIES

When conducting exhaustive verification, besides the representation of the model, the language in which the properties the system should fulfill are expressed is of crucial importance. Obviously, it should be formally defined and unambiguous and yet be able to easily and understandably express the properties provided by the engineer typically thinking in a natural language.

To eliminate the ambiguity of natural language arguments, the verifier may formalize the desired properties using first-order logic. Thus, for instance, for an elevator system, a property stating "pressing the button at floor 1 at time t (predicate $pr1$) will eventually lead to the cabin arriving to that floor (predicate $arr1$)" may be formalized as:

$$\forall t(pr1(t) \rightarrow \exists t'(t' > t \wedge arr1(t')))$$

The above notation, however, has the problem of being fairly cumbersome, a fact emphasized in more complex properties. Pnueli proposed in 1977 [6] that another formalism, called *temporal logic* might be better suited for expressing the properties typical in the verification process.

1.5 CONTRIBUTIONS

This report presents a simple, parallel programming language and defines its operational semantics. The main contribution is a presentation of its boolean encoding for bounded model checking. The encoding is proven both sound and complete with respect to the prefixes of the legal execution sequences of the language. In addition, an encoding of reachability properties is presented and a known technique for translating safety properties to reachability ones is discussed. Finally, an enhancement to the language and semantical models capable of abstracting away intermediate states with the necessary modifications to the encoding are presented.

1.6 RELATED WORK

The seminal paper coining the term bounded model checking is [3]. The authors present results of an implementation accepting a subset of SMV as its input language. The language is rather hardware-oriented, a program containing low-level descriptions of communicating finite state machines.

Another approach utilizing SMV is in [2]. The idea is to translate Promela specifications to the input language of SMV to provide the possibility of choosing "different logics and verification methods as needed". Their idea does not contain the concept of bounded model checking per se, but combined with the efforts above and in the NuSMV [9] project the relevance to the present work is evident.

The software design group at the Massachusetts Institute of Technology (MIT) has a project in which a lightweight object modelling notation called Alloy has been developed [17]. The language is accompanied by a constraint solver to find bugs in specifications [18]. The solver also operates by translating the problems to boolean formulae, the constraints being the maximum number of loop executions and heap objects. The application domain, however, is not the analysis of concurrent software as in the present work, but rather object models.

1.7 OUTLINE OF THE REPORT

Chapter 2 presents the SPINB syntax together with its operational semantics. In Chapter 3 the boolean encoding (translation to propositional logic) of SPINB programs is discussed. It is further divided into the presentation of the arithmetical elements and then the actual encoding of the statement types. The chapter also proves the soundness and completeness of the chosen approach with respect to the operational semantics presented in Chapter 2. Chapter 4 elaborates on temporal logic, presents a taxonomy on the typical verification properties and the translation of the chosen reachability formulae to propositional logic. Chapter 5 presents an augmentation of the language allowing nondeterministic choice. In addition, semantical models capable of abstracting away unnecessary intermediate states and their relation to the interleaving model are discussed. Some ideas of how the SPINB encoding ought to be modified to accommodate them are presented together with thoughts on future work.

2 SPINB PROGRAMMING LANGUAGE

The key advantage of model checking is the provision of automated algorithms for exhaustive verification. However, to fully realize the benefits for the broadest possible audience of industrial engineers, the modelling language and the formalism used for expressing the properties would have to be familiar and intuitive. SPINB aims at achieving this goal by providing a C-style syntax for control structures combined with elements of SPIN pertaining to process initialization and interprocedural communication. The specifications written in the language are analyzed for the violation of reachability properties.

The language supports basic integer arithmetic having two variable types, `int` and `short`. In principle, their value range could be the same as in modern computer architectures. However, it is presumed that in the target domain, protocol software, the variables are simple counters not assuming very large values. Therefore, in a typical implementation their width would be restricted to match the needed range, since the unnecessary bits would needlessly complicate the formula given to the SAT solver.

Similarly, domain considerations have led to the decision of not supporting floating point arithmetics (together with the recognition of the difficulties such data types may entail). The control flow constructs implemented are `while` and `if...else if` with the semantics similar to those of C [20] and Java. In addition, unconditional jumps to labels are supported.

Interprocedural communication is realized through either shared global variables or FIFO queues. The syntax of the queue declarations are adopted from SPIN [16]. Thus, the following statement `chan i = [2] of {int, int};` declares a queue named `i` having the capacity of two and carrying messages consisting of two integers. The queue send and reception commands use the operators `!` and `?`, respectively. The command `i!1,2;`, for instance, sends a message to the queue declared above. For the complete syntax in Backus-Naur form refer to Table A.1 in Appendix A.

Example 1 Consider a parallel program modelling the traditional producer - consumer problem with a counter counting the messages sent. The SPINB syntax of the program would be the following.

```
chan i = [2] of {int};

int p() {                  int q() {
  int a;                     int b;
  i!2;                       i?b;
  a = a + 1;                 goto q0;
  goto p0;                 }
}

init {
  run p();
  run q();
}
```

2.1 OPERATIONAL SEMANTICS

In order to a programming language to be precisely determined, the syntax description will have to be accompanied by some way of determining the "meaning" of syntactically valid programs, i.e. program semantics. Traditionally, these descriptions were informal, expressed in a narrative form using natural language instead of a rigorous formal notation [28]. However, the vagueness and ambiguities of natural language arguments have proven the approach unsatisfactory. The approach of this report is to define the semantics of SPINB programs using the *operational* approach, i.e. describe the possible execution sequences of valid programs utilizing an abstract machine. The ideas and the notations with the necessary modifications are from Kröger [22].

As the data types SPINB programs operate with are integers, a formal language for there their manipulation is needed. Associate with each SPINB program Π a classical first order language $\mathcal{L}_p$ with the following elements:

1. Variables (integer or queue) formed from the declarations of Π.
2. Constant symbols corresponding to the capacities of the declared queues. (for queue q, q_{cap} is used)
3. For each queue, a variable telling the number of used slots. (for queue q, q_l is used)
4. The binary functions $\{+, -, *, /, \%\}$.
5. The binary relations $\{=, <, \leq, >, \geq, \neq\}$.
6. The unary function $-$.
7. The symbols $\neg$, $\rightarrow$ and $\forall$.

The introduction of integer variables has to take into account the possibility of the lengths of the queues being greater than one as well as nontrivial integer arrays. The queues have a "second dimension" in that the messages may be longer than a single variable. The discussion below uses the notation x_{nm} to refer to the message element m in the queue position n.

To define the semantics of such a language, its structure $\mathbf{S}$ has to be defined. The structure consists of the following elements:

1. A set $|\mathbf{S}| \neq \emptyset$, called universe.
2. For every n-ary function $f \in \mathcal{L}_p$ its interpretation $\mathbf{S}(f)$:$|\mathbf{S}|^n \rightarrow |\mathbf{S}|$.
3. For every n-ary relation $r \in \mathcal{L}_p$ its interpretation $\mathbf{S}(r) \subset |\mathbf{S}|^n$.

Let the universe for the languages associated with SPINB programs be the set of integers $\mathbb{Z}_{2^n}$, where n is the width of the integer type. The interpretation of the binary functions $+$, $-$, $*$, / and % as well as the unary function $-$, will be those of traditional integer arithmetics in the field $\mathbb{Z}_{2^n}$. Similarly, the binary relations will be interpreted according to the ordering of integers.

The relation of the above definitions to the SPINB syntax is that the expressions used, for instance, in statements like `var = expr;` will be precisely the terms of the language $\mathcal{L}_p$ and the boolean conditions used in `if(cond)` and `while(cond)` will be a subset of the formulae of $\mathcal{L}_p$. However, this requires one addition to $\mathcal{L}_p$, namely the introduction of a constant symbol for each element of the set $|\mathbf{S}|$, i.e. each integer.

To be able to uniquely identify the execution points of SPINB programs, the operational semantics assumes each statement in a program to be uniquely labelled. In the definitions below, the letters α and β are typically used. In addition, each parallel component is assumed to terminate to a unique label f_l, serving the purpose of detecting finished components.

Each parallel component Π_i of a SPINB program may be regarded as a statement sequence. Let $\mathcal{M}_\psi$ be the set of labels occurring in the statement sequence ψ (including the label f_l denoting termination) and $\mathcal{F}(\mathcal{L}_p)$ be the set of formulae of the language $\mathcal{L}_p$. To prepare the definition of valid execution sequences, the following sets are associated with each statement sequence ψ:

- entry(ψ) $\in \mathcal{M}_\psi$
- trans(ψ) $\subset \mathcal{M}_\psi \times \mathcal{F}(\mathcal{L}_p) \times \mathcal{M}_\psi$
- exits(ψ) $\subset \mathcal{M}_\psi \times \mathcal{F}(\mathcal{L}_p)$

Their intuitive meaning is that the set entry(ψ) is the first label in a statement sequence, the set trans(ψ) describes the conditions that must be fulfilled for the execution in ψ to proceed from a label to another, and the set exits(ψ) tells the requirements for a statement to terminate.

The sets are then defined inductively based on the statement type. In the definition, the set $\mathcal{E}$ contains the statements of the type `var = expr;` and $\mathcal{Q}$ is the set of queue manipulating statements.

1. $\psi \equiv \alpha : x \in \mathcal{E}$
   entry(ψ) $= \alpha$
   trans(ψ) $= \emptyset$
   exits(ψ) $= (\alpha, \top)$

2. $\psi \equiv \alpha : x!y; \in \mathcal{Q}$
   entry(ψ) $= \alpha$
   trans(ψ) $= \emptyset$
   exits(ψ) $= (\alpha, x_l < x_{cap})$

3. $\psi \equiv \alpha : x?y; \in \mathcal{Q}$
   entry(ψ) $= \alpha$
   trans(ψ) $= \emptyset$
   exits(ψ) $= (\alpha, x_l > 0)$

4. $\psi \equiv \alpha :$ `if(`cnd_1`)` ψ_1 `else if(`cnd_2`)` $\psi_2 \ldots$ `else` ψ_n
   entry(ψ) $= \alpha$
   trans(ψ) =trans(ψ_1) $\cup \cdots \cup$ trans(ψ_n) $\cup$
   $\{\langle \alpha, cnd_1, \text{entry}(\psi_1)\rangle, \langle \alpha, \neg cnd_1 \wedge cnd_2, \text{entry}(\psi_2)\rangle,$
   $\langle \alpha, \neg cnd_1 \wedge \neg cnd_2 \wedge \ldots, \text{entry}(\psi_n)\rangle\}$
   exits(ψ) =exits(ψ_1) $\cup$ exits(ψ_2)$\cup \ldots \cup$ exits(ψ_n)

5. $\psi \equiv \alpha : \texttt{goto}\ \beta;$
   entry$(\psi) = \alpha$
   trans$(\psi) = \{\langle \alpha, \top, \beta \rangle\}$
   exits$(\psi) = \emptyset$

6. $\psi \equiv \psi_1;\ \psi_2$, where ψ_1 is an unlabelled statement and ψ_2 a statement sequence.
   entry$(\psi) =$ entry(ψ_1)
   trans$(\psi) =$trans$(\psi_1) \cup$ trans$(\psi_2) \cup$
   $\{\langle \beta, cond_1,$ entry$(\psi_2)\rangle \mid (\beta, cond_1) \in$ exits$(\psi_1)\}$
   exits$(\psi) =$ exits(ψ_2)

7. $\psi \equiv f_l$:, where f_l is the finishing label.
   entry$(\psi) = f_l$
   trans$(\psi) = \emptyset$
   exits$(\psi) = \emptyset$

The `if` statement in the list above is given in its most complex form. The simplifications for the other cases are straightforward. Notice that the constructs for the `while` statement are not given. This is because it can be regarded as a shorthand definable by the constructs given above. The translation is given in Figure 2.1.

```
L1: while (c1) {                L1: if (c1) {
          stmt 1;                      stmt 1;
          ⋮                            ⋮
          stmt n;                      stmt n;
    }                                  goto L1;
                                  }
```

Figure 2.1: While as a shorthand.

In order to define the execution sequences, one further concept is needed. Let program Π have p parallel components. A *program state* for Π is a $(p+2)$-tuple $\eta = (\mu, \lambda_1, \ldots, \lambda_p, \kappa)$ where:

- μ assigns a value to every program variable (an element of the universe of $\mathcal{L}_a$).
- $\lambda_i \in \mathcal{M}_{\psi_i}$
- $\kappa \in \{0, \ldots, p\}$

Informally, μ describes the memory state of the computation, the λ_i:s are the labels of the elements of the respective components to be executed next (control flow) and κ gives the component to be scheduled next, with the special value 0 denoting the last state of the execution.

With the variable valuation μ, it is possible to obtain a value for every term of the language $\mathcal{L}_p$ and a truth value for its every formula. For the term t and the formula f, these are usually denoted $\mathbf{S}^{\mu}(t)$ and $\mathbf{S}^{\mu}(f)$, respectively.

Definition 1 *A program state $\eta = (\mu, \lambda_1, \dots, \lambda_p, \kappa)$ yields the program state $\eta' = (\mu', \lambda'_1, \dots, \lambda'_p, \kappa')$, if the following properties hold:*

- if $\kappa = i$, then $trans(p_i)$ contains an element $\langle \lambda_i, cond, \lambda'_i \rangle$ such that $\mathbf{S}^\mu(cond)$ evaluates to true and all for all $k \neq i$, $\lambda'_k = \lambda_k$.
- if $\kappa = i$ and λ_i is a statement not in $\mathcal{E} \cup \mathcal{Q}$ then $\mu' = \mu$.
- if $\kappa = i$ and λ_i is of the form $x = e; \in \mathcal{E}$ then $\mu'(x) = \mathbf{S}^\mu(e)$ and for all other variables y other than x, $\mu'(y) = \mu(y)$.
- if $\kappa = i$ and λ_i is of the form $x!(e_0, \dots, e_n) \in \mathcal{Q}_s$ then
  $\mu'(x_{x_l 0}) = \mathbf{S}^\mu(e_0), \dots, \mu'(x_{x_l n}) = \mathbf{S}^\mu(e_n)$,
  $\mu'(x_l) = \mu(x_l) + 1$ and
  for all variables $y \notin \{x_{x_{l+1} i}, x_l\}$, $\mu'(y) = \mu(y)$.
- if λ_i is of the form $x?(y_0, \dots, y_n) \in \mathcal{Q}_r$ then
  $\mu'(y_1) = x_{00}, \dots, \mu'(y_n) = x_{0n}$,
  $\mu'(x_{00}) = \mu(x_{10}), \dots, \mu'(x_{0n}) = \mu(x_{1n}), \dots \mu'(x_{(x_l - 1)n}) = \mu(x_{x_l n})$,
  $\mu'(x_l) = \mu(x_l) - 1$ and for all variables
  $y \notin \{y_1, \dots y_n\} \cup \{x_{ij} | 0 < i < x_l - 1, 0 < j < n\} \cup \{x_l\}, \mu(y') = \mu(y)$

The definition above models the effects of an executed statement on the memory state and the control flow. From the first item above it can be seen that the execution of a statement not manipulating the variables or queues causes no changes in μ. The second property says that execution of an assignment statement causes the variable in the left hand side to change its value to what is obtained by evaluating the expression on the right hand side. No other variable is changed.

The queue send operation causes the values of the expressions in the expression list to be copied to the queue slot pointed by the x_l variable whose value is incremented. The variables other than pertaining to the queue operated on retain their value.

The queue reception operation causes the message in the first queue slot to be copied to the variables listed in the operation. In addition, the values in the queue are shifted down so that the message in the second slot is moved to the first etc. The value of x_l is then decremented. The definition of an execution sequence is based on the previous definition:

Definition 2 *An execution sequence of* Π *(w.r.t.* $\mathbf{S}$*) is a (finite or infinite) sequence* $\mathbf{W}_\Pi = \{\eta_0, \eta_1, \dots\}$ *of program states with the following properties:*

- $\eta_0 = (\mu_0, \alpha_0^1, \dots, \alpha_0^p, \kappa_0)$ where μ_0 assigns the initial values to the variables and the α_0^i denote the labels of the first statements in each component. The initial values are assigned according to the declarations in the program text. If a variable is not given a value, it is initialized to zero.
- For any consecutive pair η_j and η_{j+1} of program states, η_j yields η_{j+1}
- if $\mathbf{W}_\Pi$ is finite, its last state is of the form $\eta = (\mu, \lambda_1, \dots, \lambda_p, 0)$, and no other value for the scheduling component would yield any program state.

For the purpose of bounded model checking, a finite prefix of a potentially infinite execution sequence is defined as follows:

Definition 3 *An execution sequence of* Π *(w.r.t* **S***) of length* k *consists of* k *program states that form a prefix of an execution sequence.*

Example 2 Consider the previous example. Here the same program text is given with the labels needed for the definitions of the operational semantics.

```
chan i = [2] of {int};

int p() {                    int q() {
  int a;                       int b;
  p0: i!2;                     q0: i?b;
  p1: a = a + 1;               q1: goto q0;
  p2: goto p0;                 fq:
  fp:                        }
}

init {
  run p();
  run q();
}
```

The set trans(p) and trans(q) can be constructed from the inductive definitions with the following results:

$$\text{trans}(p) = \{\langle p0, i_l < 2, p1\rangle, \langle p1, \top, p2\rangle, \langle p2, \top, p0\rangle\}$$
$$\text{trans}(q) = \{\langle q0, i_l > 0, q1\rangle, \langle q1, \top, q0\rangle\}$$

Since neither component is supposed to terminate, the finishing labels do not appear in any transition. Consider next the possible execution sequences of this program. Initially, its memory state μ_0 is defined as follows:

$$\mu_0 = \{i_{00} = 0, i_{10} = 0, i_l = 0, i_{cap} = 2, a = 0, b = 0\}$$

The control flow positions point to $p0$ and $q0$, the first statements. To complete the initial state consider the available transitions in trans(p) and trans(q). In trans(p) the transition from $p0$ to $p1$ can be taken because $i_l < 2$ evaluates to $0 < 2$. However, in trans(q) the transition from $q0$ is not possible because $i_l > 0$ evaluates to false. Thus the only possible initial state is:

$$\eta_0 = (\mu_0, p0, q0, p)$$

Executing then the first statement in p gives, following the rules of the operational semantics, the following memory state:

$$\mu_1 = \{i_{00} = 2, i_{10} = 0, i_l = 1, i_{cap} = 2, a = 0, b = 0\}$$

In addition, the label $p1$ is reached. In this state, both p and q can be scheduled (due to the boolean conditions in the transitions $\langle p1, \top, p2\rangle$ and

$\langle q0, i_l > 0, q1 \rangle$ evaluating to true). Assuming p is scheduled again, the second execution state is:

$$\eta_1 = (\mu_1, p1, q0, p)$$

Executing the statement pointed to by $p1$, the memory state becomes:

$$\mu_2 = \{i_{00} = 2, i_{10} = 0, i_l = 1, i_{cap} = 2, a = 1, b = 0\}$$

Let q be the scheduled component in the state 2 (possible by the argument above). Then the execution state η_2 is:

$$\eta_2 = (\mu_2, p1, q0, q)$$

Executing the queue read statement pointed to by $q0$ leads to a state where the control flow of q advances to $q1$ and the memory state is the following:

$$\mu_2 = \{i_{00} = 0, i_{10} = 0, i_l = 0, i_{cap} = 2, a = 1, b = 2\}$$

Due to the components not terminating, the execution sequence above can be extended ad infinitum.

3 THE BOOLEAN ENCODING

The key idea in Bounded Model Checking (BMC) is to translate the system model and the specification to a propositional formula having a satisfying truth assignment if and only if the system model has a behavior that violates the specification. Such a translation is based on unrolling the transition relation of the system and thus in order to obtain a finite formula the amount of transitions has to be limited. Therefore, the process has an additional parameter giving the upper bound to the counterexamples produced. This chapter presents the translation of SPINB programs to boolean formulae together with establishing the soundness and completeness of such an encoding with respect to the finite prefixes of the execution sequences defined in the operational semantics. The discussion is complemented in the next chapter with a description of the translation of the class of considered properties.

The key ideas have been adopted from Kautz and Selman [19]. Following their approach the formulae used are given using predicates and variables together with universal and existential quantification. This is done for clarity and is not contrary to the description of a translation to a propositional formula as the predicates can straight forwardly be translated to propositional atomic formulae by grounding them over the (finite) quantifications. The universal quantifier translates to a conjunction. For instance, the formula $\forall t\, p(t)$ is the same as $p(0) \wedge \cdots \wedge p(k)$, assuming the possible values for t run from zero to k. Similarly, the existential quantifier translates to a disjunction.

In addition, a notational shorthand of the following type is applied:

$$[n, m]\{a_1, \ldots, a_i\}$$

The sentence above is satisfied by a truth assignment, if and only if the amount of true atomic statements in the set $\{a_1, \ldots, a_i\}$ is in the range given by the numbers n and m (the lower and the upper bound, respectively). All the formulae of this form can be expressed using traditional propositional logic. The only combination used in this work is that both n and m have the value one. In propositional logic, this translates to a disjunction (so that at least one of them is true) together with negated conjunctions with the different pairs of literals, ruling out models in which two or more of them would be true.

The encoding involves two rather orthogonal issues. First is the encoding of the binary relations and the arithmetic operations, the elements of the language $\mathcal{L}_p$. The second part is the correct maintenance of the control flow that can further be divided to two parts. The "local" part consists of propositional formulae pertaining only to a single statement, modelling how the control flow advances when a statement is executed. The "global" part collects elements from several statements in demanding, for instance, that only a single statement in a component may be executable at any time point.

The classification above is given to clarify the elements of the encoding and to explain the order of the discussion below. The chapter commences with a brief description and justification of the SPINB predicates. Then, the encoding of the binary relations and the arithmetic operations is analyzed. The translation of the different statement types completes the boolean formula and the process is illustrated by working through a simple example. The chapter concludes with the proofs of soundness and completeness of the encoding.

3.1 SPINB PREDICATES

The predicates used in the translation are described in Table (3.1). The key issues in the verification of SPINB programs are the maintenance of the correct control flow (i.e. statements may not be executed in an incorrect order with respect to the program text) and the correct values of the variables. The correct control flow is maintained using three predicates. The first one gives the position of the control flow within a component and the second one the component chosen for execution (analogous to the position elements λ_i and the scheduling element κ in the operational semantics, respectively). The third one denotes the successful termination of a component.

The values of the variables are represented using a binary encoding, for instance the x_{it} refers to the bit i of the variable x in the state t. Sometimes the notation is abbreviated by using x_t to refer to the entire vector. Thus, a separate predicate is not needed. However, a correct queue manipulation requires the knowledge of the available slots. This is the purpose of the qu-predicate. The limiting values of zero and the queue capacity carry a special role, denoting an empty and a full queue, respectively.

Table 3.1: The predicates of SPINB translation.

Predicate	Meaning
$p(la, t)$	The control flow has reached label la at time t.
$ac(p, t)$	The component p was active at time t.
$f(p, t)$	The component p has finished by time t.
$qu(q, i, t)$	i slots of the queue q are used at time t.

3.2 BINARY RELATIONS

The output of the translation of a SPINB program can be regarded as a symbolic encoding of all the possible interleaving executions of the program. Therefore, the possibilities of making inferences about the actual values of the variables are limited compared to the operational semantics that models the executions explicitly with all the variable values known. Rather, the encoding represents them as formulae having models if and only if the desired binary relation holds. In the following the binary relations of $\mathcal{L}_p$ are presented using the notational convention of relating the expressions l and r each being n bits wide, the numbering of the bits starting from 1 (the least significant bit). A single bit is referred to using a subscript (as in l_i). Notice that the encodings describe the general case, where both of the operands are unknown. If one of them is a constant, the equations can often be drastically simplified by straight forward manipulation according to the rules of propositional logic. The encodings are adopted from standard material of hardware design, also previously applied for formal verification (see, e.g. [35]).

3.2.1 Equality

This is probably the most obvious case. Two expressions are equal, if and only if all their bits are equivalent. Put more formally:

$$(l = r) \equiv_{def} \bigwedge_{1 \leq i \leq n} (l_i \leftrightarrow r_i)$$

3.2.2 Less Than (with Equality)

The relation is a bitwise comparison. The relation $l < r$ is true if the most significant bit of l is zero and the same bit in r is one. Otherwise, if they are the same, the comparison will have to be extended to the next significant bit and apply the same reasoning recursively. As a formula:

$$\begin{aligned} l < r \equiv_{def} & (E_n \leftrightarrow (l_n \leftrightarrow r_n)) \wedge \\ & (L_n \leftrightarrow (\neg l_n \wedge r_n)) \wedge \\ & \bigwedge_{0<i<n} (E_{n-i} \leftrightarrow (E_{n-i+1} \wedge (l_{n-i} \leftrightarrow r_{n-i})) \wedge \\ & \bigwedge_{0<i<n} (L_{n-i} \leftrightarrow (E_{n-i+1} \wedge (\neg l_{n-i} \wedge r_{n-i})) \wedge \\ & \bigvee_{0<i\leq n} L_i \end{aligned}$$

The encoding above uses two sets of temporary bits, the variables E_i have the meaning that the bits from the most significant one to the bit i are equal. A variable L_i is true if and only if all the bits from the most significant one to the bit $i + 1$ are equal, the bit l_i and r_i are zero and one, respectively. If any one of the variables L_i are true, the condition holds. The encoding of the relation with equality requires only the following modification.

$$L_1 \leftrightarrow (E_2 \wedge (\neg l_1 \vee r_1))$$

It states that if all the bits until the least significant position have been equal, the only possibility of the predicate to be violated occurs if the least significant bits in the left and right operands have the values one and zero, respectively.

The introduction of $2n$ temporary variables may be disturbing. However, the representation above is to clearly show the reasoning behind the encoding. One may of course trivially replace the temporary variables with their definitions the tradeoff being ever longer formulae containing identical constituents. The issue probably also depends on the actual width of the operands and the knowledge of the heuristics of the SAT solver.

3.2.3 Other Relational Expressions

The encoding of the remaining relational expressions is based on the encodings above and related by the following rather trivial equations:

$$\begin{aligned} l \neq r &\equiv \neg(l = r) \\ l > r &\equiv r < l \\ l \geq r &\equiv r \leq l \end{aligned}$$

3.3 ARITHMETICS

SPINB supports the standard mathematical operations of addition, subtraction, multiplication, integer division and modulo. The operations obviously differ from the relational operators in that instead of merely having a truth value they have a result. Therefore the notational conventions have to be extended for the result bits. The result is given in a bit vector $x_1, \ldots, x_n$, corresponding to the expression $x = l\ op\ r$. The results of the arithmetic operations may be wider than the width of the variable to which the results is to be stored. SPINB handles the event by simply ignoring the excess bits (so actually the results are $\pmod{2^n}$ where n is the width of the result variable). A similar discussion as in the case of binary relations applies here as well, the equations can be simplified if one of the operands is a constant. It should be noted that several VLSI designs exist for implementing the arithmetic operations. Any one of them could form the basis for the definitions below. The ones selected, albeit not the most effective, are presented due to their simplicity, manifested by analogies to elementary school algebra.

3.3.1 Constant

The simplest arithmetic expression that can be set to a variable is a constant. The encoding simply represents the value as a binary number and sets the variable equivalent the encoding thus obtained.

3.3.2 Addition

The presentation is a formula describing a base two variant of the standard algorithm taught at elementary school with the each bit possibly toggling a carry bit to be set. The carry bits are propagated to the direction of higher order bits. Formally:

$$\begin{aligned} x = l + r \quad \equiv_{def} \quad & (x_1 \leftrightarrow (l_1 \oplus r_1)) \land (c_1 \leftrightarrow (l_1 \land r_1)) \land \\ & \bigwedge_{2 \leq i \leq n} (x_i \leftrightarrow (l_i \oplus r_i \oplus c_{i-1})) \land \\ & \bigwedge_{2 \leq i \leq n} (c_i \leftrightarrow ((l_i \land r_i) \lor (c_{i-1} \land (l_i \lor r_i)))) \end{aligned}$$

3.3.3 Subtraction

Subtraction can be encoded as follows:

$$
\begin{aligned}
x = l - r \quad \equiv_{def} \quad & (x_1 \leftrightarrow (l_1 \oplus r_1)) \wedge (c_1 \leftrightarrow (\neg l_1 \wedge r_1)) \wedge \\
& \bigwedge_{2 \le i \le n} (x_i \leftrightarrow (l_i \oplus r_i \oplus c_{i-1})) \wedge \\
& \bigwedge_{2 \le i \le n} (c_i \leftrightarrow ((l_i \wedge r_i \wedge c_{i-1}) \vee (\neg l_i \wedge (r_i \vee c_{i-1}))))
\end{aligned}
$$

3.3.4 Multiplication

The encoding of multiplication follows also the traditions of elementary school. The algorithm is based on first multiplying each single bit in the right operand with the entire left, the result being stored to temporary variables. The binary single bit multiplication is trivially a simple $\wedge$, the result being one only if and only if both the operands are. The temporary results are then appropriately added (the bit position j in the right operand causes an j-bit shift in the left) to obtain the final result. In the formula below the terms T_{ij} refer to the bit storing the temporary result obtained from the single bit operation $l_i \times r_j$. The additions are denoted using the $\sum$ sign, encoded using the formula in Section (3.3.2). The term T_j refers to the bit vector $\{T_{nj} \ldots T_{1j}\}$. Repeated additions (sums having more than two terms) can be encoded by first computing the results of the addition of first two terms into temporary variables and then summing that result with the third etc.

$$
x = l * r \quad \equiv_{def} \quad \bigwedge_{1 \le i,j \le n} (T_{ij} \leftrightarrow l_i \wedge r_j) \wedge (x \leftrightarrow \sum_{1 \le j \le n} T_j << (j-1))
$$

To further clarify the concept, the elementary school algorithm is illustrated in Figure 3.1.

$$
\begin{array}{rcccc}
 & l_4 & l_3 & l_2 & l_1 \\
\times & r_4 & r_3 & r_2 & r_1 \\
\hline
 & T_{41} & T_{31} & T_{21} & T_{11} \\
 & T_{32} & T_{22} & T_{12} & \\
 & T_{23} & T_{13} & & \\
+ & T_{14} & & & \\
\hline
 & x_4 & x_3 & x_2 & x_1
\end{array}
$$

Figure 3.1: Multiplication with four bits.

3.3.5 Division and Modulo

The encoding of division utilizes also previously defined operations, namely subtraction and the comparison for less than or equality. The operation starts from an initial position in which the bits of the divisor are shifted left $n - 1$ bits. The bits in the quotient are then determined by shifting it right one bit at a time. The bit will be one if the shifted divisor is less than the dividend and zero otherwise. If the bit is one, the shifted divisor will be subtracted from the dividend. For clarity, the operation is presented in a procedural form as follows:

```
$x = l/r \quad \equiv_{def}$
        Let
            $rem_n = \{0, \ldots, 0, l_n, \ldots l_1\}$ ($n$ zeros)
            $div_n = \{0, r_n, \ldots, r_1, 0, \ldots, 0\}$ ($n - 1$ zeros)
            $x_i = div_i \leq rem_i$
            if($x_i$) { $rem_{i-1} = rem_i - div_i$ }
            else { $rem_{i-1} = rem_i$ }
            $div_{i-1} = div_i >> 1$
        in  $\{x_n, \ldots, x_1\}$
```

The if-else-structure can be encoded with implication and conjunction:

$$x_i \rightarrow (\ldots) \wedge \neg x_i \rightarrow (\ldots)$$

The result of the remainder operator ($x = l \,\%\, r$) can be obtained from the same procedure by setting x equivalent to rem_1.

3.4 STATEMENTS

The execution of all the different statement types is triggered by the control flow having reached that statement and the component the statement is in being active. Therefore, the equations modelling the different types have a common part $p(la, t) \wedge ac(p, t) \rightarrow \ldots$. Their effects, however, vary. Since the boolean expressions in SPINB do not have side effects, the only statement types affecting variable values are the assignments and the queue operations whereas `if...else`, `while` and `goto` only affect the control flow.

One central issue in the encoding of planning problems using propositional satisfiability [19] as opposed to for instance logic programs is the tendency to obtain formulae having many models not corresponding to real executions. This is a result of only encoding the effects of the operations in the particular problem domain. Applied to the SPINB translation, one example would be the case of the executed statement being an assignment to a variable. Only encoding the effects would lead to the variable to be assigned to have the correct value in the next step. However, if the models were not in some way restricted the other variables could assume any combination of values and the truth assignment would still form a valid model. This is known as the frame problem [25] and the necessary frame axioms are presented together with the encodings of the statement types. Referring to the classification into local and global elements, the effect part of the statement is local whereas the frame axioms may involve several statements thus being global.

As stated in the beginning of the chapter, the evaluation of the binary relations and the arithmetic expressions is an orthogonal issue to the maintenance of the control flow. As the arithmetic encodings have already been discussed, the following presentation assumes their encoding to be given by a partial function from the terms and formulae of $\mathcal{L}_p$ to boolean formulae. When stating, for instance, that the variable x will have the value equal to the result of the expression e_1 in the state t the notation $enc(x_t = e_1)$ is used. Similarly with boolean conditions, the condition c_1 is encoded to $enc(c_1)$ and if several conditions are combined with boolean connectives, the complete encoding applies the enc function to the atomic parts and combines them with the appropriate connective. Thus, the condition $enc($`c1 && c2`$)$ translates to $enc(c1) \wedge enc(c2)$.

3.4.1 Assignments

According to the SPINB grammar an assigment statement is of the form `x = expr`, where `x` refers to a variable capable of storing integer values and `expr` is an arithmetic expression. Intuitively, the execution of such a statement causes the symbolic evaluation of `expr` using the techniques in the previous section. The results of the evaluation are then set to the binary encoding of `x`.

$$\begin{aligned} p(la,t) \wedge ac(p,t) \quad &\rightarrow \quad (enc(x_{t+1} = expr)) && (3.1)\\ &\text{where} \quad la \text{ is the label of the statement} \\ &\qquad\quad\; t \text{ is the state and} \\ &\qquad\quad\; p \text{ is the component containing } la \\ &\qquad\quad\; x_{t+1} \text{ is the value of } x \text{ at time } t+1 \end{aligned}$$

The execution of an assignment statement also affects the control flow in that it will render the next statement amenable to execution.

$$\begin{aligned} p(la,t) \wedge ac(p,t) \quad &\rightarrow \quad p(lb,t+1) && (3.2)\\ &\text{where} \quad la \text{ is the label of the statement} \\ &\qquad\quad\; t \text{ is the state.} \\ &\qquad\quad\; p \text{ is the component containing } la \\ &\qquad\quad\; lb \text{ is the label of the next statement} \end{aligned}$$

One problem still remains. Namely, were the statement la not executed in time step t the preconditions of the implications would be false and the consequenes could therefore assume any combinations of truth values. One attempt in solving the problem could replace the implications with equivalences in the formulae above. However, this would limit the models too much, since, for instance, in the case of a constant expression, there may be other statements setting x to the same value and furthermore, in some other execution one of them could be executable in the same state. The requirement can be formulated by demanding that either the variable retains its value or some statement modifying it is executed. This translates to the following disjunction, where the ... in the end illustrate the fact that the disjunct has to contain all the statements capable of modifying the variable x in the state t. Following the introduced classification, this is a global frame condition.

$$(enc(x_{t+1} = x_t) \vee ((ac(p,t) \wedge p(la,t)) \vee \dots) \tag{3.3}$$

A similar problem arises with Formula (3.2) where restrictions are required in order to maintain the correct control flow. If the implication was changed to an equivalence, the results would be incorrect due to the fact that the label pb could also be reached by it being a target of an unconditional or conditional jump. A formula similar to (3.3) could be used here too. However, in order to handle other issues pertaining to control flow, the following construct is used:

$$[1,1]\{p(la,t), p(lb,t), \dots, p(ln,t)) | \{la, lb, \dots, ln\} \in p\} \tag{3.4}$$

As already stated, Formula (3.4) is a shorthand restricting the amount of true elements in the set to precisely one. The meaning is to reject models where several statements within one component were possible, i.e. to disallow a split control flow.

3.4.2 Conditional Execution

An `if ...else` structure is encoded using the techniques for relational expressions. Similarly, as in the discussion of the operational semantics, the structure is presented in its most complex form, simplifications being easy. Assume a structure of the form:

```
la:  if(c1) { lb:  } else if(c2) { lc:  } ...else { ln:  }
```

According to the operational semantics, the boolean conditions are simultaneously evaluated and the block of the first one evaluating to true is chosen. The situation can be taken care of with the following formulae.

$$(p(la,t) \wedge ac(p,t)) \rightarrow (enc(c1) \leftrightarrow p(lb,t+1)) \tag{3.5}$$

$$(p(la,t) \wedge ac(p,t) \wedge \neg p(lb,t+1)) \rightarrow (enc(c2) \leftrightarrow p(lc,t+1)) \tag{3.6}$$

$$(p(la,t) \wedge ac(p,t) \wedge \neg p(lb,t+1) \wedge \neg p(lc,t+1) \wedge \cdots) \rightarrow p(ln,t+1) \tag{3.7}$$

In the formulae above all the elements have the common fragment reflecting the requirement of simultaneous evaluation $(p(la,t) \wedge ac(p,t))$. However, the expressions occurring earlier in the program text are prioritized, by having in each subsequent formula the precondition that no previous expression has been evaluated to true (this is the significance of the predicates $\neg p(lb,t+1)$ etc.). If the structure contains no unconditional `else`-block in the end, Formula (3.7) has to point to the first statement after the structure.

3.4.3 Unconditional Jumps

Unconditional jumps are probably the easiest to encode. If the control flow has reached a `goto` statement, this implies that it will reach its target (inferrable from the program text) in the next state:

$$
\begin{array}{rcl}
p(la,t) \wedge ac(p,t) & \rightarrow & p(lb,t+1) \\
& \text{where} & la \text{ is the label of the statement} \\
& & t \text{ is the time step.} \\
& & p \text{ is the component containing } la \\
& & lb \text{ is the label of the jump target}
\end{array}
$$

The predicate expressing the control flow to have reached the statement has to be added to the set of Formula (3.4) restricting the control flow.

3.4.4 Conditional Execution with a Loop

The encoding of the `while` structure follows its definition as a shorthand for an `if` with a single condition and an unconditional jump in the end. The encoding exploit Formulae (3.5, 3.7 and 3.8).

3.4.5 Queue Manipulations

Queue statements are similar to the assignments in that they may modify a variable. In addition, they provide the means for communicating components to synchronize. This is achieved by blocking a component in a read statement if the channel is empty and conversely a send statement blocks if the channel is full.

As in the operational semantics, additional elements are needed for the correct maintenance of the queue status. Namely, predicates to indicate how many slots of the queue are occupied with zero resembling the empty queue and queue capacity (statically declared in the program code) corresponding to a full queue. In the formulae below, the usage predicates are denoted $qu(q,n,t)$ meaning that n slots of queue q are used at time t. Then obviously $qu(q,0,t)$ means an empty queue and a special symbol qc is used to denote queue capacity (the predicate full is then $qu(q,qc,t)$). A slot of the queue is denoted with a subscript (as in q_{it}, first the slot number and then the state[1]). The queue send and reception statements are assumed to be of the form $q!x$ and $q?x$ where x denotes the variable list containing the message to be sent or the storage to which the read message is saved, respectively. In case of a send statement the message could also be a constant, the variable being the more general case.

Queue Send

To recognize the demand of the queue not being full when material is sent, to correctly advance the control flow and require the component to be active,

[1]Notice, that in a complete presentation a third element describing the bit would be needed. However, this detail is omitted.

the following formula is needed:

$$\begin{array}{rcll} p(la,t) \wedge ac(p,t) & \rightarrow & \neg qu(q,qc,t) \wedge p(lb,t+1) & (3.8) \\ & \text{where} & la \text{ is the label of the send statement,} & \\ & & t \text{ is the state,} & \\ & & p \text{ is the parallel component containing } la, & \\ & & q \text{ is the queue manipulated,} & \\ & & qc \text{ is the capacity of } q, & \\ & & lb \text{ is the label of the next statement.} & \end{array}$$

To copy the message to the correct slot and to maintain the predicates describing queue usage, the following statements are needed:

$$p(la,t) \wedge ac(p,t) \rightarrow \bigwedge_{0 \leq i \leq qc-1} (qu(q,i,t) \rightarrow enc(q_{(i+1)(t+1)} = x_t) \wedge qu(q,i+1,t+1)) \tag{3.9}$$

$$p(la,t) \wedge ac(p,t) \rightarrow \bigwedge_{1 \leq i \leq qc} (\neg qu(q,i,t+1) \rightarrow enc(q_{i(t+1)} = q_{it})) \tag{3.10}$$

The first formula has the intuitive meaning that if the send statement is executed, then the message is copied to the first available slot and the usage parameter is incremented. The second is to maintain the queue contents of the slots not manipulated. The formula does the unnecessary restriction of maintaining the contents past the current usage count. However, this leads to a more compact representation.

An alert reader may have noticed a problem with Formula (3.9). Namely, so far the queue usage parameter have not been constrained in any way and a valid model may contain several conflicting usage counts. This in turn, could lead to the message to be sent to be copied to several queue slots. The problem is resolved by the following formula:

$$[1,1]\{qu(q,0,t),\ldots,qu(q,qc,t)\} \tag{3.11}$$

As in the case of restricting the control flow, this limits the amount of true usage parameters to precisely one in each state .

Analogous to the assigments to normal integer variables, if a queue is not operated upon, the slots retain their values from the previous state and the usage parameters remain unaltered.

$$\begin{array}{ll} \bigwedge_{0 \leq i \leq qc} & (enc(q_{i(t+1)} = q_{it}) \wedge (qu(q,i,t+1) \leftrightarrow qu(q,i,t)) \vee \\ & ((p(la,t) \wedge ac(p,t)) \vee \ldots) \end{array} \tag{3.12}$$

To correctly maintain the control flow, the queue send statements will also have to be in the control flow restriction set of Formula (3.4).

Queue Receive
A successful queue reception requires the queue not to be empty and implies a progress in the control flow:

$$p(la,t) \wedge ac(p,t) \quad \rightarrow \quad \neg qu(q,0,t) \wedge p(lb,t+1) \tag{3.13}$$

where la is the label of the receive statement,
t is the time step,
q is the queue manipulated,
lb is the label of the next statement.

The message is copied from the first queue slot to the variable(s) the message is to be stored:

$$p(la,t) \wedge ac(p,t) \rightarrow enc(x_{t+1} = q_{1t}) \tag{3.14}$$

Probably the most complex element in the statement is the operation on the usage parameter and the values in the slots. Firstly, a reception requires a formula implying that that after the statement one more slot is available. Secondly, all the elements will have to be shifted forward by one element, so the message in slot 2 becomes the message in slot 1 etc. The issue can be taken care of with the following formulae:

$$p(la,t) \wedge ac(p,t) \quad \rightarrow \quad \bigwedge_{1 \le i \le qc} (qu(q,i,t) \rightarrow qu(q,i-1,t+1)) \tag{3.15}$$

$$p(la,t) \wedge ac(p,t) \quad \rightarrow \quad \bigwedge_{0 \le i \le qc-1} (enc(q_{i(t+1)} = q_{(i+1)t})) \tag{3.16}$$

The same frame conditions as for the queue send statement apply also to the queue reception statement (see Formulae (3.11, 3.12 and 3.4)).

3.5 THE BIG PICTURE

In the sections above the encoding of the different statement types was given together with a discussion about the necessary and sufficient constraints to the modelling. This section discusses the missing pieces required for the complete picture of the interleaving execution model. The encoding is illustrated by working through a simple example.

3.5.1 The Time Argument and the Interleaving Model

Firstly, it should be noted that in the formulae above, the treatment of the time argument present in every predicate has been rather abstract. To concretize it a lower and an upper bound to the possible states during which a statement can be executed is needed. Obviously, it is possible to encode each statement from the initial state to the upper bound k, giving the length of the counterexample, but this would lead to redundant formulae (implications where the precondition would be false in every model). This can be easily improved by local control flow analysis. Assume a control flow graph of a component is constructed [1]. Then, the earliest possible state for a node can be inferred by choosing the lowest figure from the set of values of its immediate predecessor and adding one. This is also the approach taken in the encoding of SPINB. The analysis does not help in determining the upper bound though, since it is dependent on the operations of the other concurrent components. Therefore, the encoding conservatively assumes the upper bound to be k.

Secondly, even though the discussion so far has involved the concept of an interleaving execution model, it has not been present in the encoding. The formulae needed are similar to Formula (3.11), i.e. cardinality constraints restricting the amount of active components to precisely one. This has to apply for every state.

$$\bigwedge_{0 \leq t \leq k} [1,1]\{ac(p_1,t),\ldots,ac(p_n,t)\} \tag{3.17}$$

Thirdly, a frame condition is needed to maintain the control flow in the same position in a component not chosen for execution. The quantification of the labels $\forall la$ obviously refers to the statements within one component.

$$\bigwedge_{0 \leq t \leq k} (\forall p(\forall la | la \in p)(ac(p,t) \vee (p(la,t+1) \leftrightarrow p(la,t)))) \tag{3.18}$$

3.5.2 Initialization and Termination

The discussion so far has not been concerned with the initial memory state and the control flow. The initial memory state consists of the values of the variables and the queue contents. Obviously, when an execution starts each queue should be empty. Therefore, the following formula is needed.

$$\forall q(qu(q,0,0)) \tag{3.19}$$

The cardinality constraint (Formula (3.11)) enforces the other usage predicates to be false. The initial values of the queue variables do not have to be restricted since an empty queue is not read and a send operation is guided by the usage predicates. SPINB assumes, however, that the local and global variables, unless stated otherwise in the program text, are initialized to zero. The encoding trivially sets the bits to zero at time zero.

What remains is the issue of the parameters to the concurrent components. Recall that a syntactically valid SPINB program must contain an init-block initializing the concurrent components (e.g. it may contain a statement of the form `run p(2, 5);`). The syntax was limited to constant arguments. This makes the encoding easy, since the only thing is to set the symbolic elements equal to the constant arguments. That is, assuming a declaration of the form `p(int i, int j) {...}` and 4-bit integers:

$$\neg i_{40} \wedge \neg i_{30} \wedge i_{20} \wedge \neg i_{10} \wedge j_{40} \wedge \neg j_{30} \wedge \neg j_{20} \wedge j_{10}$$

It should be stated that there is often the need to conduct the verification on more than one combination of parameter values or even them all. In

SPINB this can be achieved by relaxing the initialization formulae, in the first case by encoding the desired combinations as a boolean formula stating the required condition and if the entire set of possible values were needed, the initialization would not demand anything from the variables in the initial state. Obviously, the syntax would have to be modified to allow this.

The issue of termination concerns the case when a particular component finishes within the bound. If this was not considered, the cardinality constraint above could be satisfied by choosing an already completed component to be scheduled, thus effectively idling or incorrectly resuming execution from an arbitrary position. The issue is resolved by having statements from which a component may terminate set a special atomic formula $f(p,t)$, true. Then, a component being active would imply that $f(p,t)$ would not be true thus denying idling models. The formula is formalized as follows:

$$\bigwedge_{0 \leq t \leq k} \forall p(ac(p,t) \rightarrow \neg f(p,t)) \qquad (3.20)$$

Furthermore, in order to avoid models, where $f(p,t)$ would be set without there being a real cause, the predicate has to be added to the constrained set in Formula (3.4) and to the set of Formula (3.18) to be maintained once having become true.

3.5.3 The Complete Formula

The complete encoding is a conjunction of all the elements presented in the chapter. In estimating its efficiency, the following result can be given:

Lemma 1 *The encoding of a concurrent SPINB program is a conjunction of $\mathcal{O}((l + v + p) * k)$ formulae where l is the number of statements, v is the number of variables, p is the amount of parallel components and k is the bound.*

Proof: The term $l * k$ stems from the fact that the effects of the statements have to be encoded for k states in the worst case. For each of them a constant number of statements is required. The same bound applies to the requirement of only one statement being amenable to execution in each time step. The term $v * k$ is caused by the need to maintain variable values if they are not modified (Formula (3.3)). The last term, $p * k$ comes from the requirement of maintaining the control flow in inactive components and not allowing finished components to be active. The initial conditions add only a constant number of elements. $\square$

The lemma above would not be very useful per se, since the conjuncts could be of arbitrary complexity. In the case of the SPINB encoding this could indeed be the case since unrestricted arithmetic expressions are allowed in assignments. However, it is assumed that those do not occur in practise and it should be noted that the presented encodings grow only polynomially with respect to the number of arguments. The length of the conjuncts maintaining the values of the variables and control flow are limited by the number of places in which a variable can be modified and the number of statements in a component, respectively.

3.5.4 An Example

Figure 3.2 shows a simple SPINB program with the labels used in the encoding displayed for clarity. Notice also that the component p could be regarded as being initially written using a `while` statement and then manipulated for encoding to the presented form. In the following, the encoding of the program for executions of three steps is constructed assuming integer variables 4 bits wide. Since neither of the parallel components do not have any parameters, the initializations reduce to setting i and j to zero and to setting the first statements in both components schedulable.

```
int i,j;

int p() {                          int q() {
  pa: if (j < 4) {                   qa: if (i > 1) {
  pb:   i = i + 1;                   qb:   j = 4;
  pc:   goto pa;                         }
      }                              qc: j = 2;
  fp:                                fq:
}                                  }

init() {
  run p();
  run q();
}
```

Figure 3.2: A simple example

$$(\neg i_{40} \wedge \neg i_{30} \wedge \neg i_{20} \wedge \neg i_{10} \wedge \neg j_{40} \wedge \neg j_{30} \wedge \neg j_{20} \wedge \neg j_{10}) \tag{3.21}$$
$$p(pa, 0) \wedge p(qa, 0) \tag{3.22}$$

When the effects of the statements are then encoded, it is first noticed that the control flow may reach pa from the initial state on (indeed, it is required by the initialization). The encoding is based on the equations in Section (3.4.2). However, the comparison is now between an unknown variable and a constant resulting in an easier encoding since the bits of the right operand are now known. In fact, the encoding of $j < 4$ requires only that the two most significant bits of j are zero. If the condition is not met the component terminates, thus setting the f-predicate.

$$p(pa, 0) \wedge ac(p, 0) \rightarrow (\neg j_{40} \wedge \neg j_{30} \leftrightarrow p(pb, 1)) \tag{3.23}$$
$$(p(pa, 0) \wedge ac(p, 0) \wedge \neg p(pb, 1)) \rightarrow f(p, 1) \tag{3.24}$$

$$p(pa, 1) \wedge ac(p, 1) \wedge (\neg j_{41} \wedge \neg j_{31} \leftrightarrow p(pb, 2)) \tag{3.25}$$
$$(p(pa, 1) \wedge ac(p, 1) \wedge \neg p(pb, 2)) \rightarrow f(p, 2) \tag{3.26}$$

$$p(pa, 2) \wedge ac(p, 2) \wedge (\neg j_{42} \wedge \neg j_{32} \leftrightarrow p(pb, 3)) \tag{3.27}$$
$$(p(pa, 2) \wedge ac(p, 2) \wedge \neg p(pb, 3)) \rightarrow f(p, 3) \tag{3.28}$$

The statement *pb* then is an assignment the right side incrementing the value of i. Following then the encoding in Section (3.4.1), the result of the arithmetic expression has to be evaluated to temporary variables. Notice again that since the other element of the addition is one, the encoding is a bit simpler than the general case. The control flow will advance to the label *pc*. Since the if condition has to be evaluated before the statement can be executed, the earliest state for it is 1.

$$\begin{aligned}
p(pb,1) \wedge ac(p,1) \rightarrow &((T_{11} \leftrightarrow \neg i_{11}) \wedge (c_{11} \leftrightarrow i_{11})\wedge \\
&(T_{21} \leftrightarrow (i_{21} \veebar c_{11})) \wedge (c_{21} \leftrightarrow (i_{21} \wedge c_{11}))\wedge \\
&(T_{31} \leftrightarrow (i_{31} \veebar c_{21})) \wedge (c_{31} \leftrightarrow (i_{31} \wedge c_{21}))\wedge \\
&(T_{41} \leftrightarrow (T_{31} \wedge c_{31}))) \qquad (3.29) \\
p(pb,1) \wedge ac(p,1) \rightarrow &((i_{12} \leftrightarrow T_{11}) \wedge (i_{22} \leftrightarrow T_{21})\wedge \\
&(i_{32} \leftrightarrow T_{31}) \wedge (i_{42} \leftrightarrow T_{41})) \qquad (3.30) \\
p(pb,1) \wedge ac(p,1) \rightarrow &\ p(pc,2) \qquad (3.31)
\end{aligned}$$

$$\begin{aligned}
p(pb,2) \wedge ac(p,2) \rightarrow &((T_{12} \leftrightarrow \neg i_{12}) \wedge (c_{12} \leftrightarrow i_{12})\wedge \\
&(T_{22} \leftrightarrow (i_{22} \veebar c_{12})) \wedge (c_{22} \leftrightarrow (i_{22} \wedge c_{12}))\wedge \\
&(T_{32} \leftrightarrow (i_{32} \veebar c_{22})) \wedge (c_{32} \leftrightarrow (i_{32} \wedge c_{22}))\wedge \\
&(T_{42} \leftrightarrow (T_{32} \veebar c_{32}))) \qquad (3.32) \\
p(pb,2) \wedge ac(p,2) \rightarrow &((i_{13} \leftrightarrow T_{12}) \wedge (i_{23} \leftrightarrow T_{22})\wedge \\
&(i_{33} \leftrightarrow T_{32}) \wedge (i_{43} \leftrightarrow T_{42})) \qquad (3.33) \\
p(pb,2) \wedge ac(p,2) \rightarrow &\ p(pc,3) \qquad (3.34)
\end{aligned}$$

The `goto` statement labelled *pc* is then encoded using the formula in Section (3.4.3). The earliest state for it to be executable is two and a with the given bound a single formula suffices:

$$p(pc,2) \wedge ac(p,2) \rightarrow p(pa,3) \qquad (3.35)$$

Moving on the component q the statement $q0$ is encoded similarly than the statement *pa*. Again the boolean condition is easier than the general setting due to the comparison with a constant. The condition $i > 1$ is true if one or more of its three most significant bits are one. In this case, the if-block is followed by the statement *qc*.

$$\begin{aligned}
p(qa,0) \wedge ac(q,0) \rightarrow ((i_{40} \vee i_{30} \vee i_{20}) \leftrightarrow p(qb,1)) \qquad (3.36) \\
(p(qa,0) \wedge ac(q,0) \wedge \neg p(qb,1)) \rightarrow p(qc,1) \qquad (3.37)
\end{aligned}$$

$$\begin{aligned}
p(qa,1) \wedge ac(q,1) \rightarrow ((i_{41} \vee i_{31} \vee i_{21}) \leftrightarrow p(qb,2)) \qquad (3.38) \\
(p(qa,1) \wedge ac(q,1) \wedge \neg p(qb,2)) \rightarrow p(qc,2) \qquad (3.39)
\end{aligned}$$

$$\begin{aligned}
p(qa,2) \wedge ac(q,2) \rightarrow ((i_{42} \vee i_{32} \vee i_{22}) \leftrightarrow p(qb,3)) \qquad (3.40) \\
(p(qa,2) \wedge ac(q,2) \wedge \neg p(qb,3)) \rightarrow p(qc,3) \qquad (3.41)
\end{aligned}$$

The statement qb assigns a constant to the variable j. Since the condition of the statement qa must in any case be evaluated before the control flow can reach it, the effects are modelled from the state one.

$$p(qb,1) \wedge ac(q,1) \;\rightarrow\; \neg j_{42} \wedge j_{32} \wedge \neg j_{22} \wedge \neg j_{12} \tag{3.42}$$
$$p(qb,1) \wedge ac(q,1) \;\rightarrow\; p(qc,2) \tag{3.43}$$
$$p(qb,2) \wedge ac(q,2) \;\rightarrow\; \neg j_{43} \wedge j_{33} \wedge \neg j_{23} \wedge \neg j_{13} \tag{3.44}$$
$$p(qb,2) \wedge ac(q,3) \;\rightarrow\; p(qc,3) \tag{3.45}$$

The encoding of the statement qc is identical except for the label and the value of the constant. Since it is the last statement, its execution will set the f-predicate. It should be noted that since the execution of qb is conditional, the first state qc may be executed is also one.

$$p(qc,1) \wedge ac(q,1) \;\rightarrow\; \neg j_{42} \wedge j_{32} \wedge j_{22} \wedge \neg j_{12} \tag{3.46}$$
$$p(qc,1) \wedge ac(q,1) \;\rightarrow\; f(q,2) \tag{3.47}$$
$$p(qc,2) \wedge ac(q,2) \;\rightarrow\; \neg j_{43} \wedge j_{33} \wedge j_{23} \wedge \neg j_{13} \tag{3.48}$$
$$p(qc,2) \wedge ac(q,2) \;\rightarrow\; f(q,3) \tag{3.49}$$

Having thus completed the encodings of the statements it is the time to address the frame axioms. In this case the values of i and j will have to be maintained if not manipulated and the control flow in inactive components has to remain in the same position. Furthermore, the control flow within p and q has to be unambiguous in all states.

$$((i_{41} \leftrightarrow i_{40}) \wedge (i_{31} \leftrightarrow i_{30}) \wedge (i_{21} \leftrightarrow i_{20}) \wedge (i_{11} \leftrightarrow i_{10})) \vee (ac(p,0) \wedge p(pb,0)) \tag{3.50}$$
$$((i_{42} \leftrightarrow i_{41}) \wedge (i_{32} \leftrightarrow i_{31}) \wedge (i_{22} \leftrightarrow i_{21}) \wedge (i_{12} \leftrightarrow i_{11})) \vee (ac(p,1) \wedge p(pb,1)) \tag{3.51}$$
$$((i_{43} \leftrightarrow i_{42}) \wedge (i_{33} \leftrightarrow i_{32}) \wedge (i_{23} \leftrightarrow i_{22}) \wedge (i_{13} \leftrightarrow i_{12})) \vee (ac(p,2) \wedge p(pb,2)) \tag{3.52}$$

$$((j_{41} \leftrightarrow j_{40}) \wedge (j_{31} \leftrightarrow j_{30}) \wedge (j_{21} \leftrightarrow j_{20}) \wedge (j_{11} \leftrightarrow j_{10})) \vee (ac(q,0) \wedge (p(qb,0) \vee p(qc,0))) \tag{3.53}$$
$$((j_{42} \leftrightarrow j_{41}) \wedge (j_{32} \leftrightarrow j_{31}) \wedge (j_{22} \leftrightarrow j_{21}) \wedge (j_{12} \leftrightarrow j_{11})) \vee (ac(q,1) \wedge (p(qb,1) \vee p(qc,1))) \tag{3.54}$$
$$((j_{43} \leftrightarrow j_{42}) \wedge (j_{33} \leftrightarrow j_{32}) \wedge (j_{23} \leftrightarrow j_{22}) \wedge (j_{13} \leftrightarrow j_{12})) \vee (ac(q,2) \wedge (p(qb,2) \vee p(qc,2))) \tag{3.55}$$

$$ac(p,0) \vee ((p(pa,1) \leftrightarrow p(pa,0)) \wedge (p(pb,1) \leftrightarrow p(pb,0)) \wedge (p(pc,1) \leftrightarrow p(pc,0))) \tag{3.56}$$
$$ac(p,1) \vee ((p(pa,2) \leftrightarrow p(pa,1)) \wedge (p(pb,2) \leftrightarrow p(pb,1)) \wedge (p(pc,2) \leftrightarrow p(pc,1))) \tag{3.57}$$
$$ac(p,2) \vee ((p(pa,3) \leftrightarrow p(pa,2)) \wedge (p(pb,3) \leftrightarrow p(pb,2)) \wedge (p(pc,2) \leftrightarrow p(pc,1))) \tag{3.58}$$

$$\begin{aligned} ac(q,0) \lor ((p(qa,1) \leftrightarrow p(qa,0)) \land (p(qb,1) \leftrightarrow p(qb,0)) \land \\ (p(qc,1) \leftrightarrow p(qc,0))) \quad (3.59) \\ ac(q,1) \lor ((p(qa,2) \leftrightarrow p(qa,1)) \land (p(qb,2) \leftrightarrow p(qb,1)) \land \\ (p(qc,2) \leftrightarrow p(qc,1))) \quad (3.60) \\ ac(q,2) \lor ((p(qa,3) \leftrightarrow p(qa,2)) \land (p(qb,3) \leftrightarrow p(qb,2)) \land \\ (p(qc,3) \leftrightarrow p(qc,2))) \quad (3.61) \end{aligned}$$

$$ac(p,0) \rightarrow \neg f(p,0) \quad (3.62)$$
$$ac(p,1) \rightarrow \neg f(p,1) \quad (3.63)$$
$$ac(p,2) \rightarrow \neg f(p,2) \quad (3.64)$$

$$ac(q,0) \rightarrow \neg f(q,0) \quad (3.65)$$
$$ac(q,1) \rightarrow \neg f(q,1) \quad (3.66)$$
$$ac(q,2) \rightarrow \neg f(q,2) \quad (3.67)$$

$$[1,1]\{p(pa,0), p(pb,0), p(pc,0), f(p,0)\} \quad (3.68)$$
$$[1,1]\{p(pa,1), p(pb,1), p(pc,1), f(p,1)\} \quad (3.69)$$
$$[1,1]\{p(pa,2), p(pb,2), p(pc,2), f(p,2)\} \quad (3.70)$$
$$[1,1]\{p(qa,0), p(qb,0), p(qc,0), f(q,0)\} \quad (3.71)$$
$$[1,1]\{p(qa,1), p(qb,1), p(qc,1), f(q,1)\} \quad (3.72)$$
$$[1,1]\{p(qa,2), p(qb,2), p(qc,2), f(q,2)\} \quad (3.73)$$

To complete the encoding and enforce the interleaving model of execution it is required that precisely one component is active in all states:

$$[1,1]\{ac(p,0), ac(q,0)\} \quad (3.74)$$
$$[1,1]\{ac(p,1), ac(q,1)\} \quad (3.75)$$
$$[1,1]\{ac(p,2), ac(q,2)\} \quad (3.76)$$

The complete encoding is then the conjunction of the elements above. To show that it bears resemblance (without claiming anything about soundness and completeness) one complete model is discussed.

Formula (3.74) requires that precisely one component has to be active. Assume that it is p, thus setting $ac(p,0)$. In addition j was initially set to zero. Therefore the preconditions for the Formula (3.23) are met and the left side of the implication evaluates to true. Therefore, $p(pb,1)$ has to be set (so the control flow in p advances). The cardinality constraint in Formula (3.69) enforces $\neg p(pa,1), \neg p(pc,1)$ and $\neg f(p,1)$. Since the component q is not active Formula (3.59) requires its control flow to remain in the same position. Since no variable is changed, Formulae (3.50 and 3.53) will cause the values of the variables to be carried over to the next step.

Assume that p is the component scheduled also in the second state, corresponding to $ac(p,1)$ being true. Since $p(pb,1)$ was set to true the statement *pb* will be executed, its effects being modelled in Formulae (3.29, 3.30

and 3.31). Their evaluation corresponds to the incrementation of the variable i leading to i_{12} to be included in the model. The control flow is again advanced reaching pc in state 2. Formula (3.60) will cause the control flow of q to be carried over from the previous state thus $p(qa, 2)$ will be true. Since j is not changed, Formula (3.54) will demand the value to remain the same.

For the last step, assume the component q is chosen. Thus $ac(q, 2)$ will be set to true and $ac(p, 2)$ to false. Since q is active and the control flow is at qa, the statement pointed to will be executed based on Formulae (3.40 and 3.41). Since i was incremented to 1, the most significant bits will be zero and the negative condition will prevail. This will cause $p(qc, 3)$ to be set to true and the three steps have been taken.

Consider the possibility of the presented true atoms leading to contradiction (an unsatisfied conjunct) and thus not being a model. Firstly all the cardinality constraints were satisfied, Formulae (3.74, 3.75 and 3.76) by choice and Formulae (3.68–3.73) by the fact that the description required only a single p-predicate to be true. The statements encoding the effects of the not executed statements were also satisfied due to their preconditions not being met. Neither were any conflicts found in satisfying the frame axioms. Since neither component finished, all the f-predicates are false and thus Formulae (3.62–3.67) are satisfied.

In order to express an execution in terms of the operational semantics, the sets trans(p) and trans(q) will have to be collected. Following the inductive defitinition in Chapter 2 the following sets are obtained:

$$\begin{aligned} \text{trans}(p) &= \{\langle pa, j < 4, pb\rangle, \langle pb, \top, pc\rangle, \langle pc, \top, pa\rangle\} \\ \text{trans}(q) &= \{\langle qa, i > 1, qb\rangle, \langle qa, \neg(i > 1), qc\rangle, \langle qb, \top, qc\rangle, \langle qc, \top, fq\rangle\} \end{aligned}$$

In this case the program state is a 4-tuple $(\mu, \lambda_p, \lambda_q, \kappa)$. The following list gives the complete execution the model describes:

0. $(\mu_0, pa, qa, p), \mu_0(i) = \mu_0(j) = 0$
1. (μ_0, pb, qa, p)
2. $(\mu_2, pc, qa, q), \mu_2(i) = 1, \mu_2(j) = 0$
3. $(\mu_2, pc, qc, 0)$

The initial memory state μ_0 assigns the initial values to the program variables. In this case, $\mu_0(i) = \mu_0(j) = 0$. The elements λ_p and λ_q will be pa and qa, respectively, and since the first statement executed was chosen from p the scheduling component κ will be p. Since the statement in p is of the form `while`, not modifying any variables, the next program state will be (μ_0, pb, qa, p). Remember that the change from pa to pb in λ_p required the existence of an element $\langle pa, cond, pb\rangle \in \text{trans}(p)$ such that $cond$ holds. Indeed, the element $\langle pa, j < 4, pb\rangle$ fulfills the requirement.

In the second step, the execution of pb changes the memory state by incrementing i. The requirement of an element in trans(p) for the transition from pb to pc is trivially met by the element $\langle pb, \top, pc\rangle \in \text{trans}(p)$.

In the last step, the component q is scheduled and the condition of the if statement is evaluated. Since $\mu(i) = 1$, the next state is reached by the transition $\langle qa, \neg(i > 1), qc\rangle \in \text{trans}(q)$.

3.6 SOUNDNESS AND COMPLETENESS

For the encoding of the SPINB programs to fulfill its purpose, on one hand, the models of the formula should correspond to executions in the operational semantics and, on the other hand, each execution should have a corresponding model. This section establishes these theorems of soundness and completeness of the encoding based on an inductive argument.

3.6.1 Soundness

Theorem 1 (Soundness) *Let Π be a parallel SPINB program and φ_k be its boolean encoding with the bound k. Then, if $\mathcal{M} \models \varphi_k$, $\mathcal{M}$ determines an execution sequence of Π of length k.*

Proof: The construction of the execution sequence is provided by a mapping g that maps in each state the boolean encoding of the variable values and queue contents to the memory state μ, takes the labels of the true p predicates (or the f predicate for a finished component) in each parallel component to be the control flow positions λ_i and the component of the true ac predicate to be the scheduling element κ. For the queues, the value for the last used queue slot is obtained from the qu predicate.

The proof aims to establish that g is well defined and the set of execution states obtained forms a valid execution sequence. Firstly, it should be noted that no unambiguity arises in the mapping of the control flow positions and the scheduling components. The formula φ_k is a conjunction and for each state it contains the frame axioms (Formulae 3.4 and 3.17) guaranteeing the uniqueness of the control flow positions within each component and the scheduled component, respectively.

Furthermore, in any model the atomic statements describing the bits of the variables have to have some truth value. Thus, the mapping g is well defined in the sense that it is possible from $\mathcal{M}$ to obtain a set of k consecutive execution states. The proof proceeds by induction on the set of states and shows that the changes between any consecutive pair of them agree with the requirements in the operational semantics.

1. Basic Case

   The basic case is that the initial state is correctly mapped. The initialization requirements of the boolean encoding demand that the initial values of the variables appear as conjuncts in φ_k. Thus they have to be in $\mathcal{M}$ and g maps the memory state in the first execution state correctly. Since φ_k also has the p predicates describing the first statements in each component as conjuncts they have to be true in $\mathcal{M}$. Thus, as required by the operational semantics, the control flow elements in the first execution state are the first statements in each component.

   As already stated, the active component in the first state is unique, and there has to exists one. Furthermore, it may not be a component starting with a queue receive statement, since the queues are initially empty. Otherwise, $\mathcal{M}$ would not be a model, since Formula (3.13) would be unsatisfied. Thus, the mapping is well defined and the result is a possible initial state.

2. Induction Hypothesis

   Assume that the set of execution states form a valid execution sequence up to the execution state t.

3. Induction Step

   By the induction hypothesis the execution state t has a unique active component (let it be p_l) and the corresponding component has an unambiguos control flow position (la) that are obtained from the ac and p predicates in $\mathcal{M}$. The predicates in the state t imply that certain atomic statements have to be in $\mathcal{M}$ in the state $t+1$ for it to be a model. It is shown that from these demands, the execution state $t+1$ given by the mapping g satisfies the conditions of the operational semantics.

   Firstly, it can be stated that by induction hypothesis the true p predicates in $\mathcal{M}$ in the state $t+1$ in the inactive components have to point to the same label as in the state t or otherwise, Formula (3.18) would be violated. In this sense g for the state $t+1$ is correct since this is also a requirement of the operational semantics.

   A second set of requirements depends on the statement type that is pointed to by the p predicate in the active component. The following is a case analysis of the possibilities.

   (a) Assignment

   If the statement to be executed is an assignment of the form `la: x = expr; lb: ...`, φ_k will contain Formulae (3.1 and 3.2). Now the preconditions for both of the implications evaluate to true in $\mathcal{M}$, so for it to be a model of φ_k, the right sides have to evaluate to true as well. Thus, the value of the variable x in $\mathcal{M}$ in the state $t+1$ has to be the result of $enc(x_{t+1} = expr)$. Assuming the enc function to be correct, the mapping of that value corresponds to a correct modification of the memory state for the execution state $t+1$.

   The second implication mandates $p(lb, t+1)$ modelling the advance in the control flow. As stated above, the cardinality constraints guarantee it to be unique and thus the control flow element for the active component for the execution state $t+1$ is obtained.

   For the step between the states t and $t+1$ to be correct the set trans(p_l) had to contain a triple consisting of two labels giving the control flow positions in the states t and $t+1$ and a boolean condition evaluating to true in the state t. For an assignment like the one above, it contains a triple of the form $\langle la, \top, lb\rangle$ that fulfills the requirement.

   (b) Conditional Execution

   If the statement chosen for the execution is an `if ...else` compound, Formulae (3.5, 3.6 and 3.7) are in φ_k. The precondition of the implication (3.5) evaluates to true, requiring the equivalence on the right side to do the same. Assuming the enc function to be correct, if $c1$ is true $p(lb, t+1)$ has to be in $\mathcal{M}$, otherwise it cannot. In the first case, both Formulae (3.6 and 3.7) evaluate to true and no further requirements on $\mathcal{M}$ are posed.

In the second case the precondition for Formula (3.6) is true and a similar discussion than above applies for the boolean condition $c2$. All the conditions in the structure are processed in the same way and in all cases, $\mathcal{M}$ has to contain the p predicate for the first true condition.

The structure requires the triples $\langle la, c1, lb\rangle$, $\langle la, \neg c1 \wedge c2, lc\rangle$, $\ldots$, $\langle la, \neg c1 \wedge \neg c2 \wedge \ldots, ln\rangle$ to be in trans(p_l). By the induction hypothesis, the memory state is correctly maintained up to the tth state. Thus, assuming that the function `enc` is correct, it will agree with the structure **S** and the taken transition will have the corresponding triple.

(c) Unconditional Jump

If the chosen statement is a `goto` statement is is encoded using Formula (3.8). Now, its precondition evaluates to true, so $\mathcal{M}$ has to contain the predicate $p(lb, t+1)$. The set trans(p_l) contains by definition the corresponding triple $\langle la, \top, lb\rangle$.

(d) Queue Send

If the statement is of the form `la: q!x; lb:  ...`, φ_k contains conjuncts similar to Formulae (3.8, 3.9 and 3.10). They are all implications, whose left sides evaluate to true. The first formula mandates that $\neg qu(q, qc, t)$ and $p(lb, t+1)$ are both true in $\mathcal{M}$. By the first, it cannot model a queue send statement to a full queue, a requirement met by hypothesis, and the second one provides the means for constructing the control flow element for the active component in the state $t+1$.

Formula (3.9) implements the message passing and the incrementing of the queue usage parameter. The right part of the implication is a conjunction of implications. By the induction hypothesis the queue usage is uniquely determined in the state t and points to the last used queue slot. Thus precisely one of the latter implications will have its left side true. Thus the right side of that implication has to evaluate to true in $\mathcal{M}$. Therefore, it has to contain the queue usage parameter incremented by one for the state $t+1$ and the first available queue slot has to contain the value of x. Furthermore, φ_k contains Formula (3.11) for all queues and all states. Therefore, for $\mathcal{M}$ to be a model, all the other qu predicates in the state $t+1$ have to be false.

Formula (3.10) is again an implication whose left side evaluates to true in $\mathcal{M}$. Thus the conjunction of implications in its right side has to do that as well. By the uniqueness of the qu predicate, in all cases but one, the left sides of the implications evaluate to true. Thus in all cases but one, the atomic statements encoding the queue slots have the same values as in the previous step. The one not having this requirement is the slot set to the value of the message x by the discussion above.

By the definition of the set trans(p_l), the queue send statement requires the triple $\langle la, q_l < q_{cap}, lb\rangle$ to be added. Its condition will indeed be true, since $q_l = q_{cap}$ would imply $qu(q, qc, t)$, not possible by the discussion above. Furthermore, the queue send statement is the only way for the value of the parameter q_l to increase. Therefore $q_l > q_{cap}$ is not possible.

(e) Queue Receive

Assuming a statement of the form `la: q?x; lb:  ...`, the encoding φ_k has as conjuncts Formulae (3.13, 3.14, 3.15 and 3.16). They all are implications whose left sides evaluate to true in $\mathcal{M}$.

The first one, analogously to the queue send statement disallows models reading from an empty queue. In addition, it requires $p(lb, t+1)$ to be true in $\mathcal{M}$. The second one requires that the binary encoding of x in the state $t+1$ will have the value of the first queue slot in the state t.

The right side of the third one (Formula (3.15)) is again a conjunction of implications and by the induction hypothesis in exactly one of them the left side is true. Analogously to the queue send statement, it will require $\mathcal{M}$ to contain a queue usage parameter in the state $t+1$ having a value one less than that in the state t.

Formula (3.16) requires that in the state $t+1$ in $\mathcal{M}$ the queue slots will have the same content as their successors in the queue in the state t. This models the fact that queue contents have to be shifted down after reading from it.

For a queue receive statement the triple $\langle la, q_l > 0, lb\rangle$ had to be added to the set trans(p_l). Its boolean condition is true by a similar argument than in the case of the queue send statement.

The case analysis above gives certain requirements for $\mathcal{M}$ in the state $t+1$. To show that these do not conflict with the rest of the conjuncts in φ_k, certain observations can be made. Firstly is the issue of implications encoding the effects of other possible statements in the state t. These are satisfied, because their form is an implication and their left sides evaluate to false in $\mathcal{M}$. Secondly, in all cases precisely one p predicate was required to be true in $\mathcal{M}$ in the state $t+1$. Therefore, there exists no conflict with Formula (3.4) since it can be satisfied by setting the rest to false. Finally, by a similar argument, Formula (3.11) is also satisfied in the state $t+1$.

Thus, the mapping g is well defined for the control flow positions in the state $t+1$. Furthermore, in the transition from the state t to the state $t+1$, the changes in the control flow do not conflict with the requirements of the operational semantics.

The mapping of the memory state μ was proved correct regarding the modified variable in assignment statements as well as regarding the variable pointing the last used queue slot. However, the operational semantics require all the variables not modified to retain their values.

It can be seen, that if $\mathcal{M} \models \varphi_k$, as supposed, this has to apply to it as well. Otherwise, Formula (3.3) would be violated in the state $t + 1$.

The remaining issue is the active component for the execution state $t + 1$. It cannot be a finished one, since then Formula (3.20) for the state $t + 1$ would be violated. Furthermore, it may not be a queue send statement to a full queue or a queue receive statement from an empty queue. Therefore the transition from the state t to $t + 1$ fully agrees with the operational semantics and the mapping gives a valid execution sequence. □

3.6.2 Completeness

Theorem 2 (Completeness) *Let* Π *be a parallel SPINB program and* $\mathbf{W}_\Pi$ *its execution sequence of length* k*. Furthermore, let* φ_k *be its boolean encoding with the bound* k*. Then,* $\mathbf{W}_\Pi$ *determines a model* $\mathcal{M}$ *such that* $\mathcal{M} \models \varphi_k$*.*

Proof: Let h be a mapping from $\mathbf{W}_\Pi$ to a model $\mathcal{M}$ of φ_k. The mapping is such that for each state it sets the ac predicate corresponding to the scheduling elements κ in the execution states as true and similarly the control flow positions in each parallel component in each execution state are mapped to true p predicates. If the component has reached the label denoting termination, its f predicate is set. The memory state μ in each execution state is mapped to the binary encoding of the variable values. The proof establishes that $\mathcal{M} \models \varphi_k$.

Firstly, it can be easily seen that $\mathcal{M}$ satisfies the frame conditions (Formula (3.4 and 3.17)) because in all states, precisely one p predicate in each component and precisely one ac predicate are set to true in each state. Furthermore, Formula (3.18) is also satisfied due the fact that the operational semantics requires the control flow to remain in the same position in the inactive components.

Regarding the rest of the conjuncts, an inductive argument is applied. The induction goes over the execution states of $\mathbf{W}_\Pi$. It is shown that for any execution state t, the true elements obtained by the mapping h satisfy the conjuncts relating to the state t in φ_k.

1. Basic Case

   The variables in the memory state μ of the first state in $\mathbf{W}_\Pi$ have their initial values. The control flow points to the first statements in each component. When these are mapped, the conjuncts corresponding to the initialization requirements in φ_k are satisfied.

2. Induction Hypothesis

   Assume that the conjuncts having the time argument up to the value t are satisfied by $\mathcal{M}$.

3. Induction Step

   Consider next the conjuncts having the time argument $t{+}1$. These can be divided to the frame conditions, satisfied by the discussion above, and the formulae encoding the effects of statements amenable to execution in the state t.

By the rules of the operational semantics, the transition from the execution state t to $t+1$ in $\mathbf{W}_\Pi$ corresponds to the execution of the statement pointed to by the label λ in the active component given by κ in the execution state t. By the rules of the boolean encoding, φ_k will contain boolean formulae encoding the effects the execution of that statement. The following is a case analysis of the different statement types and in each case it is proven that given the mapping h for the execution states t and $t+1$, these conjuncts are satisfied.

(a) Assignment

If the statement corresponding to the transition from the execution state t to $t{+}1$ is an assignment like `la: x = expr; lb: ...`, h will set $p(lb, t+1)$ to true and set the bits of x according to the value of `expr`.

The formula φ_k contains the implications given in Formulae (3.1 and 3.2) with their left side evaluating to true in $\mathcal{M}$. For the implications to be satisfied their right sides have to be evaluated to true as well. Assuming the correctness of the *enc* function, the binary encoding of x in the state $t+1$ agrees with that given by the mapping h and Formula (3.1) is satisfied. Furthermore, h gave $p(lb, t+1)$ and $\mathcal{M}$ satisfies Formula (3.2).

(b) Conditional Execution

If the statement corresponding to the transition is an `if ...else` structure the label in the execution state $t{+}1$ is determined by the first expression evaluating to true in $\mathbf{S}$. The boolean encoding φ_k contains the implications given by Formulae (3.5, 3.6 and 3.7).

The left side of Formula (3.5) evaluates to true in $\mathcal{M}$. By the definition of h and the correctness of the `enc` function, `enc(c1)` has a model if and only if $c1$ evaluates to true in $\mathbf{S}$. This is precisely the condition for the control flow to reach lb in the execution state $t+1$. Thus $\mathcal{M}$ satisfies Formula (3.5).

If $c1$ was indeed true, also Formulae (3.6 and 3.7) are satisfied, because their left sides evaluate to false. If this is not the case, the left side of Formula (3.6) evaluates to true and a similar argument is applied to $c2$. In this way, the argument can be extended to the entire structure.

(c) Unconditional Jump

If the transition corresponds to the execution of an unconditional jump, the encoding contains Formula (3.8) that is satisfied by the model obtained via the mapping h.

(d) Queue Send

The operational semantics allows a queue send statement of the form `la: q!x; lb: ...` to be chosen only if the queue is not full. The statement causes the control flow to proceed and modifies the memory state by copying the message sent to first available queue slot and increments the pointer q_l giving the last used slot.

For that statement, φ_k contains Formulae (3.8, 3.9 and 3.10). In the model giving by the mapping h, the left sides of all of them evaluate to true. In addition, the mapping h gives $\neg qu(q, qc, t)$ by rule regarding the queue send statement and $p(lb, t+1)$ by the advancement in the control flow. Thus, Formula (3.8) is satisfied.

In the right side of Formula (3.9), all but one implication in their conjunction are satisfied due to their left side being false. The last one is satisfied, because the transition in the operational semantics requires the first available queue slot to have the value of x in the execution state t and h maps this faithfully. Furthermore, in the execution state $t+1$ the queue usage parameter will be incremented. Thus, assuming i slots were used in the execution state t, h requires $qu(q, i+1, t+1)$ to be true in $\mathcal{M}$.

Notice, that h maps all the other queue slots than the one the message was put, to have the same values in both the states t and $t{+}1$. The one, whose value is changed is precisely the slot pointed to by the qu predicate in the state $t+1$. Thus, Formula (3.10) is also satisfied.

(e) Queue Receive

If the transition in $\mathbf{W}_\Pi$ is a queue reception statement of the form `la: q?x; lb:`, the queue in question has to be non-empty in the execution state t. Its encoding, φ_k contains the Formulae (3.13 – 3.16), all of whose left sides evaluate to true in $\mathcal{M}$.

As stated above, h maps $qu(q, 0, t)$ to false and since the control flow advances to the next statement $p(lb, t+1)$ to true. Hence, Formula (3.13) is satisfied.

Reading from the queue sets the value of x, the variable to which the message is read, equal to the first queue slot in the execution state $t+1$. Thus, the model $\mathcal{M}$ satisfies the equivalence in the right side of Formula (3.14) and therefore the entire implication.

Similarly than in the case of a queue send statement, the right side of Formula (3.15) contains set of implications of which, due to the uniqueness of the qu parameter, the left side of precisely one evaluates to true in $\mathcal{M}$. Since in the operational semantics, the variable pointing to the last used queue slot is decremented, the qu predicate given by the mapping h causes the right side of this implication to evaluate to true as well. Thus the entire implication is satisfied.

In the definition of a queue reception statement, in the state after its execution the contents of the queue had to be shifted done by one step. Thus, in $\mathcal{M}$, the right side of Formula (3.16) evaluates to true and the implication is satisfied.

The analysis above showed that were the active statement of any type, $\mathcal{M}$ satisfies the implications encoding the effects of it. However, φ_k contains also other implications that correspond to the effects of some other statement being executed in the state t. These may be active if some other component was scheduled in the state t or in the case of the statement being from the same component, in some other interleaving. They do not cause conflicts, however, since they are satisfied due to their left sides evaluating to false in $\mathcal{M}$.

In addition, φ_k contains the frame conditions requiring that the the values of the variables or the queue parameters retain their values, if no statement modifying them is executed (Formulae (3.3 and 3.12)). The model obtained from the mapping h satisfies these as well, since if a statement does not modify a variable then its value (and its boolean encoding) is the same in the memory state of the next execution state. If a modifying statement is executed, then the respective p and ac predicates will be true and the latter part of the disjunction (3.3) satisfied. Similarly, the operational semantics requires the queue parameters to be maintained, if no send or receive statement is executed. If such a statement is executed, the latter disjunct of Formula (3.12) is satisfied by a similar argument.

The Formulae (3.20) pertaining to the components being finished, remain. In the operational semantics, a finished component can never be scheduled. Thus, the atomic statements $ac(p_l, t)$ and $f(p_l, t)$ can never occur in the same state in the model $\mathcal{M}$ and the formula is satisfied.

In the discussion above, all the conjuncts in φ_k with the time argument $t + 1$ were analyzed and found to be satisfied by the model $\mathcal{M}$ given by the mapping h. Therefore, $\mathcal{M} \models \varphi_k$. $\Box$.

4 REACHABILITY PROPERTIES

Temporal logic is a formal language that has its roots in the philosophical studies of the use of natural language. A. N. Prior wanted to investigate an enhancement to traditional propositional/predicate logic by allowing sentences to assume different truth values over time [30]. Already he characterized the developed system to be applicable in the formalization of the behavior of a digital computer.

The initial system was based on replacing the modal notion of necessity (often $\Box$) as “it will always be the case that”, and possibility ($\Diamond$) as “some time in the future it will be the case that”. These temporal operators have also been adopted to the systems used in verification (they are typically **G**, something holds globally, and **F**, something holds finally). In addition, the modern temporal logic systems for computer science may contain the operator **X** (something holds in the next state) and the binary operator **U** ($p\,\mathbf{U}\,q$, p holds until q holds). In general, a temporal logic system does not have to be restricted to operators only involving the future, but the past can be incorporated as well. For instance, $\mathbf{G}^{-1}\,p$ can be taken to mean that “always in the past, p holds”. Indeed, even though the most common temporal logic systems used in verification do not involve past operators, an important class of properties can be elegantly characterized using them.

To return to the example presented in Chapter 1, the property expressed in first-order logic can be expressed using the temporal formalism as:

$$pr1 \to \mathbf{F}arr1$$

The reason, why this formalism is well suited for verification is the fact that the formulae do not involve the explicit notion of time, but still allow the reasoning of sequences of events. Verification typically abstracts away details of the artifact being modelled and thus the properties pertaining to the exact duration of an operation are not very natural. Furthermore, the analysis typically involves severals iterations with more refined models and the properties to be verified are supposed to be preserved.

The first actual techniques for using temporal logic in the reasoning about computer programs were proposed by several researchers, including Burstall [8], Kröger [21] and Pnueli [29]. Their approaches were based on proving theorems that, as stated in Chapter 1, is a process hard to automate and often difficult to use in practise.

The foundations for model checking were laid, when Clarke and Emerson in the USA and Quelle and Sifakis in France discovered independently that concurrent computer programs could be translated to state – transition systems so that the infinite paths in the structure corresponded to the concurrent executions of the program. This made it possible to use the model theoretic approach of modal logic to determine the truth values of the properties the system should fulfill. Furthermore, the states satifying a temporal formula could be computed in a time linear in the size of the model and the length of the formula and the algorithm could be automated.

However, the logical language they used was different from the one informally presented above. In it, the temporal operators **G, F, X** and **U** were preceded by second-order quantifiers **E** and **A**, exists and for all. This temporal logic language is called CTL (Computational Tree Logic), due to the fact that the executions of the systems are treated as an infinite tree rather than a set of execution paths. The quantifiers are used to quantify over the paths starting from inner nodes of the tree. To clarify, one may state, for instance, $\mathbf{AXEF}p$ meaning that all the successors of the initial state (**AX**) are first states on some path on which in some state p holds (**EF** p).

The framework not allowing the quantifiers thus reasoning about the events occurring on a single path is called Linear Temporal Logic (LTL). The merits of the two different viewpoints spawned a discussion that has continued since the early 1980's [23, 13, 34]. The following central arguments have been presented:

- The frameworks are semantically incomparable [23, 24].
- The CTL model checking problem is easier than that for LTL [32].
- Verification engineers find the branching framework unintuitive [31, 34].

The first argument means that there are properties expressible in LTL, but not in CTL and vice versa. The second one refers to a fact first provided by Clarke and Sistla [32]. Assuming that the size of the system model is m and the length of the formula n, CTL model checking can be performed in time $\mathcal{O}(mn)$, whereas for LTL the problem is **PSPACE**-complete (taking the time $\mathcal{O}(m2^n)$ in the worst case). However, it is argued, the temporal properties are typically not very long, so the theoretical penalty for using LTL does not occur in real life. In addition, in some real-life verification problems, the linear framework is actually easier [34].

The third argument is based on industrial feedback of verification using the two frameworks. For instance, IBM researchers state that "nontrivial CTL equations are hard to understand and prone to error" [31]. In addition, Vardi claims that "a perusal of the literature reveals that the vast majority of the CTL formulas used in formal verification are actually equivalent to LTL formulas" [34]. Another problem, relating to a core argument for the benefits of model checking is that the counterexample a model checker generates in case of property failure, may for some CTL formulae not be linear, but rather a computational tree.

The properties whose encoding is presented in this report form a subset of the properties expressible in LTL, adhering to the views that verification engineers tend to think linearly. In addition, in the context of bounded model checking, the encoding of LTL is polynomial in the size of the formula [3].

4.1 PROPERTY TYPES

Verification using model checking is a task requiring some experience. An engineer aiming to verify a complex system is bound to reach the limitations of the available tools and hardware. It is the consensus of the community that classifying the properties into certain categories provides one way of getting the most out of the process. Not only does it help in constructing specifications with a better structure and fewer involuntary omissions, but it also helps in recognizing which techniques would be applicable for the particular problem at hand. Such a classification also provides the justification for the choice of the subset of LTL formulae whose encoding is presented in this work.

The verification goals are historically classified into the four following categories [6]:

- *Reachability properties* state that some particular state can be reached.
- *Safety properties* express that, under certain conditions, something never occurs.
- *Liveness properties* express that, under certain conditions, something will ultimately occur.
- *Fairness properties* express that, under certain conditions, something will (or will not) occur infinitely often.

From the classes above, the first two are usually the most crucial to system correctness. For instance, the authors of [6] state that "Such properties therefore deserve a more substantial investment in terms of time, priority, rigor, etc., on the part of the human verifier."

The present work also concentrates on reachability and safety properties by giving a boolean encoding for the first category and briefly discussing a technique that allows safety properties to be reduced to reachability ones.

4.1.1 Temporal Properties and BMC

The discussion above rises the question of how the temporal properties, analyzed over infinite paths, can be combined with the finite prefixes of execution sequences obtainable from the boolean translation. Obviously, the last state of such a prefix seems to have no successor. The gap can be surpassed by assuming the last state to form a self-loop. Thus, for a property to hold globally, it suffices that it holds on all states of the prefix.

This kind of analysis provides an approximation of the property. The approximation improves as the bound (the length of the prefix) in increased. Due to the LTL model checking problem being **PSPACE** complete and BMC being in **NP**, the existence of a polynomial bound k on the length of the prefix such that the two problems would be equivalent is unlikely. Some sufficient bounds with certain restrictions are presented in [3].

4.1.2 Reachability Properties

As stated above, reachability properties express that some particular situation can be reached, i.e. there exists an execution path on which the situation occurs. However, the linear time framework used in this report verifies properties occurring on *all* paths starting from an initial state (indeed, the LTL formulae can be conceived of starting with an implicit **A** quantifier). Therefore, using this framework, the reachability properties can only be expressed in a negative sense, i.e. something is not reachable. Reformulating this in a language more closely resembling the operators of temporal logic the claim becomes, on all paths, situation p cannot occur, i.e. its negation must hold in all states on all paths.

Assuming that the undesired situation is given by the formula p, the LTL formula naturally is $\mathbf{G}\neg p$. Its boolean encoding is then straight forward. Assume that the translation is conducted with the bound k and the property p in the state t is given by $p(t)$, then the formula is:

$$\neg p(0) \wedge \cdots \wedge \neg p(k) \tag{4.1}$$

However, in order to get the counterexample in case of a property violation, the property has to be negated. The negation of Formula (4.1) is a simple application of De Morgan's theorem:

$$p(0) \vee \cdots \vee p(k) \tag{4.2}$$

When Formula (4.2) is combined with the encoding φ_k of the program to be modelled to a conjunction, the resulting formula has a model if and only if there exists an execution of the program such that on some state $t', p(t')$ holds, i.e. the undesired state is reached.

The situations given by the formula p may involve statements about the control flow or variable values. One may state, e.g., that the component p_l is never active with the ac predicate, claim that a certain label is never reached (the p predicate) or demand that the value of the variable i is never less than 5. Naturally, these can be combined with the connectives of propositional logic.

4.1.3 Safety Properties

An alert reader may have noticed that the negative reachability properties discussed above are actually safety properties. However, the informal definition of a safety property actually included the phrase "under certain conditions" whereas the properties given above are unconditional. For instance, with only reachability property one could state a rather strange claim like "the car never starts" whereas with safety properties one is able to express the more likely "the car only starts, if the key was inserted in some earlier state".

The general problem of being able to tell from an arbitrary temporal logic formula whether or not it is a safety property is a rather tricky one. One elegant characterization is a syntactic one, safety properties are of the form $\mathbf{G}\,\phi^-$, where ϕ^- is a past temporal formula (in the loose sense of including the present), i.e. only using past temporal operators together with the boolean connectives [33]. For instance, the safety property given above may be expressed in the form:

$$\mathbf{G}(starts \rightarrow \mathbf{F}^{-1}key)$$

The justification for the type of formulae given above is that when a safety property is violated, one should be able to instantly notice it. To be able to argue something in the current state relies only on the accumulated knowledge of the past events and is independent of the future.

The discussion above rises the question of how the characterization above agrees with the fact that the temporal logic systems LTL and CTL do not contain past operators. Firstly, it should be said that their omission does not prevent the formalism from expressing the properties, just the elegant characterization is losed. In addition, in principle it is possible to translate any formula involving the past to a pure future formula [6]. The translation is a rather delicate one, though, and if the branching framework is used, the translation may not be a CTL formula.

In principle, the verification of any property of the type $\mathbf{G}\,\phi^{-1}$ can be reduced to handling a reachability property. This is achieved by the introduction of so called *history variables* to the transitions in the state - transition graph. These variables are set to true only if the past formula ϕ^{-1} holds in a particular state. For instance, for the formula $\mathbf{F}^{-1}key$ above, a history variable $h1$ could be introduced. It would initially be false, but if a transition landed on a state on which key would be true, $h1$ would be set. Other transitions would leave it unchanged. Using this technique, the property above translates to the formula $\mathbf{G}(start \rightarrow h1)$ amenable for encoding using Formula (4.2). The technique of using history variables can be extended to any past time formula, by introducing a new variable for any subformula having a past time temporal operator as its root [6].

4.1.4 Liveness and Fairness Properties

Having thus presented the ideas for encoding the two most crucial property types, it should be stated that in principle any LTL formula can be encoded [3]. The translation involves the idea of locating when a finite prefix of an infinite execution in fact forms a loop. However, it introduces a quadratic number of temporary variables leading to complex formulae. Improvements on the encoding remain the subject of further work.

5 ENHANCEMENTS

This chapter presents techniques for enhancing the capacity of the introduced concepts to verification. Firstly, a construct allowing nondeterministic choice is presented. Secondly, semantical models capable of abstracting away intermediate states not affecting the presence of errors are discussed.

5.1 NONDETERMINISM

The models used in verification are typically abstractions of the artifact being modelled. They have to contain all the behaviours of the item they model, but in addition, some executions not possible in reality may be present. It turns out that abstractions are naturally modelled using a specification language having the possibility of nondeterministic choice. In the encoding for SPINB presented so far, all the elements are completely deterministic. When a statement is reached, there is only one possibility from where the execution may continue.

This section presents a structure allowing nondeterministic choice and its encoding. The syntax is taken from PROMELA. The possible paths to be taken are guarded by boolean conditions and any path having a true condition may be taken. If all the conditions are false, the entire structure is skipped.

```
la:
    if
    :: (c1) -> lb: ...
    :: (c2) -> lc: ...
    fi
ld:
```

Figure 5.1: A simple nondeterministic choice

For simplicity, let the structure have two boolean conditions, `c1` and `c2` as given in Figure (5.1), more complex structures following the same scheme. Nondeterminism can be encoded with the following demands. Firstly, if the statement is reached, the execution has to continue from one of the labels `lb, lc` or `ld`. This can be formalized with following formula (together with the cardinality constraint given in Formula (3.4)):

$$p(la,t) \wedge ac(p,t) \rightarrow (p(lb,t+1) \vee p(lc,t+1) \vee p(ld,t+1)) \qquad (5.1)$$

The nondeterministic choice between the labels following the true conditions can be encoded as follows:

$$p(la,t) \wedge ac(p,t) \rightarrow (p(lb,t+1) \rightarrow enc(c1)) \qquad (5.2)$$
$$p(la,t) \wedge ac(p,t) \rightarrow (p(lc,t+1) \rightarrow enc(c2)) \qquad (5.3)$$
$$p(la,t) \wedge ac(p,t) \rightarrow (p(ld,t+1) \rightarrow \neg enc(c1) \wedge \neg enc(c2)) \qquad (5.4)$$

Since the statement has been reached, the left sides of all the implications evaluate to true. Firstly, it should be noted that a path defined by a false condition (let, e.g. $c1$ be false) may not be taken, since then $enc(c1)$ would be false and if $p(lb, t+1)$ would be set, the implication between them would be false and Formula (5.2) unsatisfied. Consider next the case when both the conditions would evaluate to true. Then, both the implications $p(lb, t+1) \rightarrow enc(c1)$ and $p(lc, t+1) \rightarrow enc(c2)$ would be true, independent of their left sides. However, $p(ld, t+1)$ cannot be taken. Therefore, together with the cardinality constraint (Formula (3.4)) guaranteeing the uniqueness of the control flow, either of the paths can be taken.

5.2 OTHER SEMANTICAL MODELS

Concurrent systems consist of independent components whose actions are causally and temporally unrelated unless otherwise specified by some means of synchronization. When the behaviors of such systems are analyzed using the *interleaving* model, a single behavior is a totally ordered set of atomic actions. Concurrently executed actions appear arbitrarily ordered with respect to each other and the consideration of every such sequence may result in an extremely large state space [11]. The encoding presented for SPINB adheres also to the interleaving model.

This section presents two semantical models aiming to reduce the search space of the SAT solver by abstracting away states, whose evaluation is irrelevant with respect to the property being verified. The purpose is to obtain an encoding that could potentially find property violations that would remain hidden using the interleaving translation with the same bound. This section discusses briefly (without formal proofs), how the SPINB encoding could be modified to accomodate these semantical models. The set of properties is also restricted to only contain claims pertaining to the memory state.

5.2.1 Independent Actions and Step Semantics

The goal of formal verification is to exhaustively check all the executions of a concurrent system with respect to a property. An execution consists of an interleaving of execution steps for the processes [14]. It is often the case that such a sequence contains consecutive steps that are independent, i.e., their execution in any order would lead to the same state, and furthermore, the execution order does not affect the presence or absence of errors. Hence, in order to reduce the state space, it would suffice to check just one representative instead of all the orderings as in the interleaving case.

This is the reasoning behind an optimization technique known as *model checking using representatives*. Furthermore, the sequence could be shortened by abstracting away states within it. This is possible in a semantical model known as *step semantics* and it is achieved by allowing any number of independent actions simultaneously. Since the number could also be one, step executions contain all the interleavings.

It should be noted that the task of finding out a good independence relation (and thus its complement, the pairs of dependent actions) in a set of actions is a non-trivial problem. It is possible to statically infer one from the program text, but that approach would lead to unnecessary dependencies. The presentation in this and the following section assumes the relations to be given and discusses the modifications to the SPINB encoding. In addition, the adjustments are presented assuming that in any time step, only a single statement is executed within one component, or using the terminology above, all the actions within the same component are dependent.

To modify the encoding to allow several independent statements from different components to be executed in the same step is relatively easy. Formula (3.17) has to be modified to allow several components to be active in the same state:

$$\bigwedge_{0 \le t \le k} \exists p \, ac(p,t) \tag{5.5}$$

The existential quantification translates to a disjunction of the ac predicates.

$$p(la,t) \wedge ac(p,t) \leftrightarrow ex(la,t) \tag{5.6}$$

This definition above is merely a shorthand having the intuitive meaning that a statement is executed. That happens exactly when the control flow has reached it and the component it is in is active. The shorthand is used to prevent two dependent actions from occurring in the same time step. This can naturally be achieved with the following formula (assuming that the statements pointed to by la and lb are dependent):

$$\neg(ex(la,t) \wedge ex(lb,t)) \tag{5.7}$$

5.2.2 Trace Theory

Allowing any number of independent actions simultaneously, as is the case in step executions, effectively leads to a situation where several executions represent the same "concurrent behaviour". This can introduce search space adversely effecting the running time of the solver used [15]. This section sketches, how the SPINB encoding could be modified to produce formulae that only have satisfying truth assignments corresponding to "truly" different executions. For that purpose, a brief discussion of the concept of trace theory is required.

The description of sequential processes is traditionally based on the notion of a finite automaton, seen as a restricted type of a Turing machine. Their strength derives from the "simple elegance" of the underlying model. Powerful mathematical tools may be used to analyze both the structural properties, due to the graph-like description, and the behaviors of the system based on the notion of its language. The language theory has provided a valuable link to the theory of free monoids [12].

In the description of concurrent processes, the mathematical analysis of the used formalisms may sometimes be very complicated. In the 1970's, Mazurkiewicz formulated a theory that tried to apply the techniques of language theory to the analysis of concurrent systems based on the notion of concurrency understood in the same way as in the theory of Petri nets. The abstract description of a concurrent process is called a *trace*, being defined as a congruence class of word modulo identities of the form $ab = ba$ for some pairs of letters [12]. The developed theory handles well some important phenomena of concurrency and it is closely related to the theory of free monoids describing sequential processes. The next section defines more formally the concepts of trace theory necessary for the present discussion. The material has been adopted from [12].

5.2.3 Free Partially Commutative Monoids

If Σ is a finite alphabet, let Σ^* denote the set of all words over Σ. This set, together with the concatenation operator forms the free monoid with the set of generators Σ. The empty word, denoted by 1 acts as the unit element.

Let $I \subseteq \Sigma \times \Sigma$ be a symmetric and irreflexive relation over the alphabet Σ. This relation is called the *indepence* or commutation relation. It denotes the pairs of actions, whose mutual execution order does not influence the reached state. Thus, for the pair $(a, b) \in I, ab = ba$ and they may be performed in any order or even simultaneously.

The relation I induces an equivalence relation $\sim_I$ over Σ^*. Two words are equivalent, denoted for the words x and y as $x \sim_I y$, if there exists a sequence $z_1, z_2, \ldots, z_k$, such that $x = z_1, y = z_k$, and for all i, $1 \leq i < k$ there exist words z_i' and z_i'', and letters a_i, b_i satisfying:

$$z_i = z_i' a_i b_i z_i'', z_{i+1} = z_i' b_i a_i z_i'', \text{and } (a_i, b_i) \in I.$$

The set of words equivalent to x is denoted $[x]_I$. It can be verified that $\sim_I$ it the least congruence over Σ^* such that $ab \sim_I ba$ for all pairs $(a, b) \in I$. The *free partially commutative monoid* is the quotient of Σ^* by the congruence $\sim_I$, denoted $\mathbb{M}(\Sigma, I)$. In the special cases of I being the empty relation (no two letters commute) or the full relation (any two letters commute), $\mathbb{M}(\Sigma, I)$ is the free monoid Σ^* and the free commutative monoid denoted $\mathbb{N}^\Sigma$, respectively.

Example 3 Let $\Sigma = \{a, b, c, d\}$ and $I = \{(a, b), (b, a), (c, d), (d, c)\}$. Let $x = abbcbdc$. Then

$$[x]_I = \{abbcbdc, babcbdc, bbacbdc, abbcbcd, babcbcd, bbacbcd\}$$

5.2.4 Normal Forms

In determining which representative among the set $[x]_I$ to choose, the notion of a normal form is central. In this section two normal forms, the lexicographic and the Foata normal form, are presented. The presentation assumes the alphabet Σ to be totally ordered. Then the set Σ^* has a corresponding lexicographic ordering.

If X is a set of words, the unique minimal element of X with respect to the lexicographic ordering is denoted by $\text{Min}(X)$. A word x is said to be in a lexicographic normal form if it is minimal among the set of words that are equivalent to x, i.e.

$$x = \text{Min}([x])$$

The *Foata normal form* of a trace is defined using the definition above as follows. A word x of Σ^* is in the Foata normal form, if it is the empty word or if there exist an integer $n > 0$ and non-empty words x_i, $(1 \leq i \leq n)$ such that

1. $x = x_1 \cdots x_n$
2. for each i, the word x_i is a product of pairwise independent letters and x_i is minimal with respect to the lexicographic ordering,
3. for each $1 \leq i < n$ and for each letter a of x_{i+1} there exists a letter b of x_i such that $(a, b) \notin I$.

Each of the x_i is called a step. In the following, modifications to the SPINB encoding that restrict the models of the formula to correspond to executions in Foata normal form are discussed.

The requirements that were presented in the step semantics, still apply. Several active components (Formula (5.5)) are allowed, but executing dependent actions in the same step prohibited (Formula (5.7)). To restrict the models further, an additional formula is needed. It requires that if a statement is possible and all the statements from the other parallel components that it depends on are not executed, then it will be executed. This is formalized as follows:

$$p(la, t) \wedge \neg ex(lb, t) \wedge \ldots \neg ex(ln, t) \rightarrow ac(p, t) \tag{5.8}$$

In the formula above, the statement under consideration is labelled la and the dependent statements have the labels from lb to ln. Furthermore, p is the component the statement is in. Notice that if a statement is completely independent the formula reduces to $p(la, t) \rightarrow ac(p, t)$.

For the statements manipulating the queues, the necessary precondition about the queue status has to be taken into account. For instance, for the queue send statement, the queue may not be full and the formula becomes:

$$p(la, t) \wedge \neg qu(q, qc, t) \wedge \neg ex(lb, t) \wedge \ldots \neg ex(ln, t) \rightarrow ac(p, t) \tag{5.9}$$

Lemma 2 (Soundness) *Any model of the modified encoding corresponds to an execution in Foata normal form.*

Proof. In order for the execution to be of the required form, the three criteria presented above have to be considered. Firstly, the division of the sequence of actions into steps follows the division into states in the model. The lexicographic ordering is irrelevant since the atomic actions in the states are executed simultaneously. Each step is a product of independent letters, since Formula (3.4) restricts the possible statements from within a component to one, and Formula (5.7) disallows simultaneous execution of dependent statements.

The third condition is the hardest one. Assume that among the executed statements in the state $n+1$ there would exist an atomic action x such that it would be independent of all the actions occurring at time step n. Firstly, this would mean that in the state n, no statement would be executed from the component x is in. This implies that x would have to be possible also in the state n, since the control flow cannot advance, if no statement is executed. In addition, no dependent statement from the other components would have been executed. However, if this was the case, Formula (5.8) would be violated. Thus, no such x can exist $\Box$.

The proof above establishes certain aspects of the models of the formulae obtained from the modified encoding. It is not a real proof of soundness with respect to the operational semantics. However, from a model it is still possible to construct a unique execution sequence. The process is to linearize the atomic actions in each step. The bound that would be given to the length of the counterexamples is now a lower bound for the execution sequence, in most cases it would be longer.

The proof of completeness would rely on a theorem stating that every trace has a unique normal form [12]. The interleaving executions would be converted to that form and then the model candidate would be constructed with the same mapping h used in the proof of completeness of the interleaving semantics. As stated above, the discussion is based on a given dependence relation and to statically compute one that is not trivial (the full relation leading to interleaving model) is a hard task. The analysis of that problem and possibly a dynamic approach remain subjects for future work.

6 CONCLUSIONS

The aim of this report is to develop techniques to apply bounded model checking to the verification of programs written in a small, parallel programming language. The syntax of the language together with its operational semantics is described. Based on that description, the translation of syntactically valid programs to a boolean formula is devised, further divided to the analysis of the arithmetical expressions and the statement types. The encoding is proven sound and complete with respect to the operational semantics.

The use of temporal logic in verification is discussed with a justification of choosing the linear time framework. Different property types are presented and the encoding one of them, reachability, is given. Efficient translation of the entire temporal logic LTL is a topic for future work.

The encoding presented corresponds to programs executed according to the interleaving model. Chapter 5 presents the use of semantical models that executed several atomic actions simultaneously, however, not losing any erratical behaviour. The motivation and the modifications to the encoding to accomodate the semantical models are briefly discussed, a more thorough analysis being the subject of future work.

An initial implementation of the encoding presented exists. In the future, the purpose is to see how well existing SAT solvers can handle the boolean formulae resulting from the encoding when real-life model are used. During the work, it is noticed that the different conjuncts had similar elements. Therefore, the structure sharing provided by the use of boolean circuits is an interesting possibility.

Another course of actions is to integrate the BMC method with techniques capable of reasoning of potentially infinite state spaces. That is, upon noticing that a certain property holds up till a certain bound examine, whether it were possible to use, e.g., an inductive argument to prove that it holds for any bound.

A third step would be to see how data abstraction could be used together with SPINB. It is a technique aiming to reduce the state space traversed by replacing the entire range of values for a data type with a smaller set of representatives. The modification may not, however, effect the detection of error. With these steps outlined above, the goal is to assess the scalability of the method and compare is with other verification techniques.

ACKNOWLEDGEMENTS

The financial support of Helsinki Graduate School of Computer Science and Engineering (HeCSE), the Academy of Finland (project 53695) and Nokia Foundation is gratefully acknowledged.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] M. Baldamus and J. Schröder-Babo. p2b: A translation utility for linking PROMELA and symbolic model checking (tool paper). In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, pages 183–191. Springer Verlag, 2001.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1579 in Lecture Notes in Computer Science, pages 193–207, 1999.

[4] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Formal Methods in Computer Aided Design*, number 1633 in lecture Notes in Computer Science, pages 60–71, November 2000.

[5] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer-Aided Verification*, number 2102 in Lecture Notes in Computer Science, pages 454–464, 2001.

[6] B. Bérard, Bidoit M., Finkel A, Laroussinie F., Petit A., Petrucci L., Ph. Schnoebelen, and McKenzie P. *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer, 2001.

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

[8] R. M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress 74*, pages 308–312. North Holland, 1974.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Proceeding of International Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 495–499. Springer Verlag, 1999.

[10] E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[11] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[12] V. Diekert and G. Rozenberg, editors. *Book of Traces*. World Scientific, 1994.

[13] E. Emerson and J. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[14] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *Proceedings of the 1996 International Symposium of Software Testing and Analysis*, volume 21, pages 496–507, May 1996.

[15] Keijo Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory*, pages 218–232, August 2001.

[16] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[17] D. Jackson. Alloy: A lightweight object modelling notation. Technical report, Massachusetts Institute of Technology, 2000.

[18] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium of Software Testing and Analysis 2000*, pages 14–25, 2000.

[19] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th Conference on Artificial Intelligence*, pages 359–363, 1992.

[20] Brian Kernighan and Dennis Ritchie. *The C programming language*. AT & T Bell Laboratories, 2nd edition, 1988.

[21] Fred Kröger. LAR: A logic of algorithmic reasoning. *Acta Informatica*, 8(3):243–266, 1977.

[22] Fred Kröger. *Temporal Logic of Programs*. Springer Verlag, 1987.

[23] L. Lamport. "Sometime" is sometimes "not never" – on the temporal logic of programs. In *Proceedings of the 7th annual ACM Symposium on Principles of Programming Languages*, pages 174–185. ACM, January 1980.

[24] Monika Maidl. The common fragment of CTL and LTL. In *Proceedings of the 41st Conference on the Foundations of Computer Science*, pages 643–652, 2000.

[25] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

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

[27] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[28] Frank Pagan. *Formal Specification of Programming Languages.* Prentice-Hall, 1981.

[29] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science,* pages 46–57. IEEE Computer Society Press, 1977.

[30] A. N. Prior. *Past, Present and Future.* Clarendon Press, 1967.

[31] T. Schlipf, T. Buechner, T. Fritz, M. Helms, and J. Koehl. Formal verification made easy. *IBM Journal of Research and Development,* pages 567–576, 1997.

[32] A. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM,* 32(3):733–749, July 1985.

[33] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *Proceedings of the 4th ACM Symposium on Principles of Disributed Computing,* pages 39–48, 1985.

[34] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems,* number 2031 in Lecture Notes in Computer Science, pages 1–22, 2001.

[35] Poul F. Williams. *Formal Verification Based on Boolean Expression Diagrams.* PhD thesis, Technical University of Denmark, Lyngby, August 2000.

A SPINB SYNTAX

Table A.1: The syntax of SPINB

```
program := units;
units    := unit | units unit;
unit     := g_decl | proc | spec | init | ϵ;
g_decl   := type var_list ’;’ ;
type     := ’int’ | ’short’ ;
var_list := svar | var_list ’,’ svar ;
svar     := var | var ’=’ expr | var ’=’ ch_init ;
var      := name | name ’[’pconst’]’;
name     := [a-zA-Z][a-zA-Z0-9]* ;
pconst   := [1-9][0-9]* ;
expr     := expr ’+’ expr | expr ’-’ expr | expr ’*’ expr | expr ’/’ expr |
            expr ’%’ expr | expr ’»’ expr | expr ’«’ expr |
            expr ’>’ expr | expr ’<’ expr | expr ’>=’ expr | expr ’<=’ expr |
            expr ’==’ expr | expr ’!=’ expr | ’!’ expr |
            ’(’ expr ’)’ | ’-’ expr | const | name ;
const    := [0-9]* ;
ch_init := ’[’ const ’] of {’ typ_list ’}’ ;
typ_list := type | typ_list ’,’ type ;
proc    := type name ’(’ a_decl ’)’ body;
a_decl  := args | ϵ ;
args    := arg | args ’,’ arg ;
arg     := type name ;
body    := l_decls stmts ;
l_decls := l_decls l_decl | ϵ ;
l_decl  := g_decl ;
stmts   := stmt | stmts stmt ;
stmt    := name ’=’ expr ’;’ | name ’?’ margs ’;’ name ’!’ margs ’;’ |
           name ’:’ | ’goto’ name ’;’ | ’while (’ expr ’) {’ stmts ’}’
           ’if (’ expr ’) {’ stmts ’}’ elses ;
margs   := expr | margs ’,’ expr ;
elses   := ’else if (’ expr ’) {’ stmts ’}’ elses | ’else {’ stmts ’}’ | ϵ;
spec    := ’spec := G’ expr ’;’;
init    := ’init {’ run ’}’ ;
run     := ’run’ name ’(’ earg ’) ;’ ;
earg    := rarg | ϵ ;
rarg    := expr | rarg ’,’ expr;
```

ISBN 951-22-6063-8
ISSN 1457-7615