

Final thesis

**An optimising SMV to CLP(B)
compiler**

by
Mikael Asplund

LITH-IDA-EX-05/018-SE

2005-02-24

Final thesis

An optimising SMV to CLP(B) compiler

by **Mikael Asplund**

LITH-IDA-EX-05/018-SE

Supervisor: **Ulf Nilsson**

Department of Computer and Information
Science
at Linköpings universitet

Examiner: **Ulf Nilsson**

Department of Computer and Information
Science
at Linköpings universitet

Abstract

This thesis describes an optimising compiler for translating from SMV to CLP(B). The optimisation is aimed at reducing the number of required variables in order to decrease the size of the resulting BDDs. Also a partitioning of the transition relation is performed. The compiler uses an internal representation of a FSM that is built up from the SMV description. A number of rewrite steps are performed on the problem description such as encoding to a Boolean domain and performing the optimisations.

The variable reduction heuristic is based on finding sub-circuits that are suitable for reduction and a state space search is performed on those groups. An evaluation of the results shows that in some cases the compiler is able to greatly reduce the size of the resulting BDDs.

Keywords: SMV, CLP, BDD, FSM, CTL, compiler, optimisation, variable reduction, partitioning

Acknowledgements

First of all I would like to thank my supervisor Ulf Nilsson for helping me and answering all my questions. Vladislavs Jahundovics has also been of help with ideas and explanations. It has been very rewarding to discuss different matters of logic with Marcus Eriksson who will also be the opponent of this thesis.

When I have been stuck with the project it has been very nice to talk with my room mates at IDA. And of course my girlfriend Ulrika who has helped me get going and finishing this thesis instead of sitting and waiting for it to finish itself.

Contents

1	Introduction	1
1.1	Background	1
1.2	The need for a SMV to CLP compiler	2
1.3	Purpose	2
1.4	Structure of the report	3
2	Preliminaries	5
2.1	Logic concepts	5
2.2	Kripke structure	5
2.3	Computation Tree Logic CTL	7
2.4	An example	8
2.5	Symbolic Model Verifier SMV	9
2.6	Fairness	10
2.7	BDDs	12
2.8	BDD size	15
2.9	Constraint Logic Programming CLP	16
2.10	Synchronous and asynchronous	17
2.11	Partitioning the transition relation	19
	2.11.1 Disjunctive partitioning	19
	2.11.2 Conjunctive partitioning	20
3	Literature Survey	21
3.1	Compilation	21
3.2	State encoding	22

3.3	Variable removal	22
3.4	Clustering and ordering of partitions	23
3.5	Partial-Order Reduction	24
3.6	Variable ordering	24
3.7	Don't Care sets	24
4	Compiler basics	27
4.1	Architecture	27
4.2	Programming language	28
4.3	Intermediate Representation	28
4.4	SMV Language constructs	29
4.4.1	Modules	29
4.4.2	Declarations	30
4.4.3	Types	30
	Arrays	31
4.4.4	Running	31
4.5	Compiler stages	31
4.5.1	Simple reductions	32
4.5.2	Lift case expressions	33
4.5.3	Lift (reduce) set expressions	33
4.5.4	Solve finite domain constraints	34
4.5.5	Create Boolean encoding	34
4.5.6	Reduce INVAR	35
4.5.7	Optimisation	35
4.5.8	Handle running	36
4.5.9	Synchronise	36
4.6	Output	37
4.6.1	CLP	37
4.6.2	SMV	38
5	Optimisations	39
5.1	Clustering and ordering	39
5.2	State space reduction	40
5.2.1	Finding suitable reduction groups	42
5.2.2	Finding reachable states	43
5.2.3	Reencoding	44

6	Results	47
6.1	Metric	47
6.2	Comparisons	48
6.3	Interpretation	49
7	Discussion	51
7.1	The compiler	51
7.2	Performance	52
7.3	Applicability in real world cases	52
7.4	Future work	53
A	CLP syntax	55
B	Predicates	57
C	Unsupported SMV constructs	61

List of Figures

2.1	Example of a Kripke structure	7
2.2	The Kripke structure from Figure 2.1 converted to an incomplete infinite computation tree	8
2.3	CTL expressions	9
2.4	SMV example	11
2.5	(2.1) represented as a BDD	13
2.6	(2.1) represented as a ROBDD with ordering $a < b < c$	14
2.7	(2.1) represented as a ROBDD with order $a < c < b$	15
2.8	Two processes	17
2.9	Kripke structure for synchronous and asynchronous combination	18
2.10	Synchronised processes	18
4.1	Data flow	27
4.2	Translation of Boolean only case statement	33
4.3	Translation of case statement	33
4.4	Set encoding algorithm for comparison expressions	34
5.1	SMV example with one redundant variable	41
5.2	Algorithm for variable reduction	42
5.3	Algorithm for finding reduction groups	43
A.1	CLP-syntax on token level	56
A.2	CLP-syntax on string level	56

C.1 Example of parameters not handled by smv2clp 62
C.2 Equivalent of example in Figure C.1 handled by smv2clp . 62

List of Tables

4.1	Operator translation	32
5.1	No redundant variables	42
5.2	Three variables, with reachable state space of size three . .	44
5.3	A more efficient representation of the state space	44
6.1	Comparison between original problem and compiled problem	49
6.2	Effect of variable reduction	50

Chapter 1

Introduction

1.1 Background

In computer science and electrical engineering the concept of a discrete system or finite state machine (FSM) is of central importance. In essence it is a representation of a system that has a state and can move between different states depending on the input. This very simple concept can be used to describe very complex systems. All hardware construction is done using FSMs as basic building blocks. It is also widely used in communication protocols and software design.

It should not come as a surprise that there is a need to make certain that these systems do not contain any errors. Furthermore it is important to find these errors as early as possible. If an error is discovered late in the development process it will be very costly to fix. In some cases it is not only a question of development cost, but crucial that the system does not malfunction when in use. Typical areas include aviation, automobile systems and hospital equipment.

Traditionally testing has been the method used to verify discrete systems. Test cases that try to cover all possible cases are applied. However in most cases this is a daunting task and it is practically impossible to test all cases. Therefore it is often the case that errors go undetected through

the testing phase and emerge in production use. Intel's Pentium processor was discovered to be erroneous in its floating point division. This error cost Intel approximately 500 million US dollars [16]

To fully eliminate the possibility of undetected errors one must use some kind of formal verification method. That is, a formal proof that the system fulfils some specification. The systems that are to be verified are often very complex and it has been considered more or less impossible to formally verify such complex systems. However in recent years there has been a lot of improvement in this area and it is now possible to formally verify quite complex systems. As an example Intel used formal verification during the construction of the Pentium IV processor [4].

There are several systems that verify a given system description with some specification. SMV [20] is one of those systems and that is also the name of the input language.

1.2 The need for a SMV to CLP compiler

In Linköping there is a project investigating the possibilities of doing formal verification using methods from logic programming [25]. There is currently a system that takes a constraint logic program (CLP) as input and checking the CTL specification. The framework uses constructive negation and tabled resolution. A compiler that translates from SMV to CLP would make it possible to check the performance of the system for problems specified in SMV.

1.3 Purpose

The aim of the project is to develop an optimising compiler for translation of SMV code into a CLP program. The compiler is subsequently called `smv2clp`.

The syntax of the CLP-program is specified in Appendix A. The main task is to develop a compiler that translates objects and their states and state transition in SMV into a transition relation in CLP. Since the system is intended to be used as a testing environment it is important that

its behaviour is easily modified and that it is easy to extend with more functionality.

The main requirements on the project are:

- The code should be written in such a way that it is possible to replace the front-end of the compiler to support also other input languages.
- The input should support the full SMV syntax possibly with some exceptions.
- The output should be a limited CLP-language describing the transition relation.
- To allow for future experimentation it should be possible to specify, in a simple way, how much to partition the transition relation.
- The report should also contain a literature survey of what has been previously done to reduce the size of BDDs when compiling system descriptions (not necessarily expressed in SMV).

1.4 Structure of the report

The project is mainly concerned with constructing a compiler with certain properties. The report is a reflection of this; the concepts introduced are those that are necessary to know to understand the workings of the compiler.

Some prerequisites are naturally required. First of all a good understanding of logic is required and some basic mathematic notions. The concept of model checking and CTL is explained in Chapter 2.

The research in the area is summarily described in Chapter 3 with emphasis on the literature that is relevant to the project. Chapter 4 contains a description of the compiler with its architecture and description on how different language constructs in SMV are handled.

Chapter 5 is dedicated to the optimisations performed by the compiler. This includes one method which is described in the literature, and also a way of reducing the number of variables. Finally Chapter 6 and seven contain the results obtained with optimisation and a discussion along with ideas of future work.

Chapter 2

Preliminaries

This chapter is dedicated to explaining the theoretical concepts related to the compiler. It is not by any means exhaustive but should cover the most important aspects.

2.1 Logic concepts

Normal forms are very useful when dealing with propositional logic. Conjunctive normal form (CNF) and disjunctive normal form (DNF) are used throughout this thesis. Loosely formulated a formula in CNF is a conjunction of disjunctions, e.g. $(a \vee b \vee c) \wedge (\neg a \vee b)$. And a formula in DNF is a disjunction of conjunctions, e.g. $(a \wedge b \wedge c) \vee (\neg a \wedge b)$. Any logic formula can be converted into one of these normal forms using an algorithm which is polynomial in time and space.

2.2 Kripke structure

When doing model checking on a given system one must have some way to represent the specifications that the system should fulfil. This project is mainly concerned with systems whose properties can be expressed in

CTL which is described in Section 2.3. The compiler is also able to handle fairness constraints which are explained in Section 2.6.

CTL is not concerned with FSMs but rather with a very similar construct called Kripke structure.

A Kripke structure is formally defined as follows:

Definition 1 *A Kripke Structure is a four tuple $M = (S, s_0, R, L)$ where*

- *S is a finite set of states*
- *$s_0 \in S$ is a set of initial states*
- *$R \subseteq S \times S$ is a transition relation for which it holds that $\forall s \in S : \exists s' \in S : (s, s') \in R$.*
- *$L : S \rightarrow 2^{AP}$ is a labelling with the atomic propositions (AP) that hold in that state.*

An example of a Kripke structure with three states can be seen in Figure 2.1. As can be seen from the definition of the transition relation a Kripke structure is not allowed to have any deadlock states. A deadlock occurs when there is not transition going away from the current state. Unfortunately deadlock states often occur in system descriptions and most verification systems find them.

There are some differences between a Kripke structure and a (nondeterministic) FSM:

- In a Kripke structure there are no accept states as in a FSM. Or rather all states can be seen as accept states.
- A Kripke structure has no input alphabet. However input can be modeled using Kripke structures.

These are not major differences and it is possible to convert between the two representations.

The monolithic Kripke structure of a system tends to be very complex and it is seldom constructed directly. Instead a system is usually constructed by creating groups of automata that are combined in a synchronous or asynchronous manner.

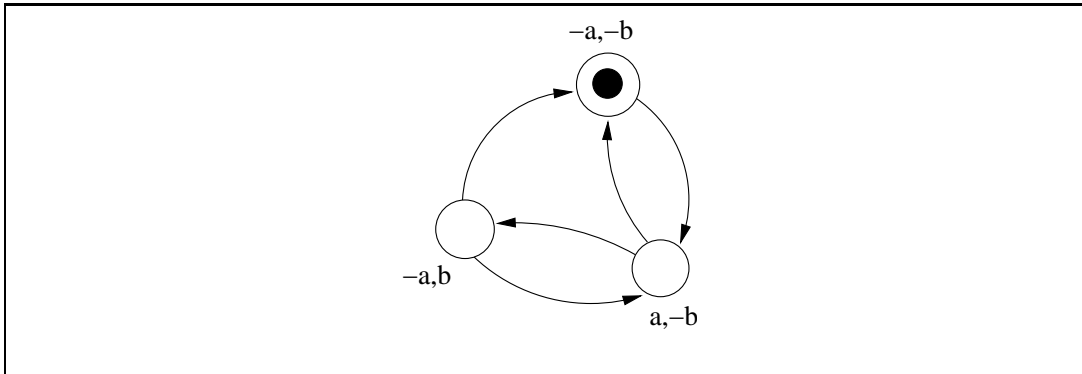


Figure 2.1: Example of a Kripke structure

2.3 Computation Tree Logic CTL

CTL is a form of temporal logic. Temporal logic is useful for specifying properties of systems whose state changes through time. Ordinary logic is not concerned with change and some kind of time concept is needed to make a formal logic system handle that a proposition can be true at time t but false at time $t+1$.

The time concept in CTL is that of steps in an infinite execution tree. Therefore the time concept is somehow implicit and hidden but it should be remembered that it is a form of temporal logic.

The CTL language is constructed of state formulas with the following syntax:

- A proposition $p \in AP$ is a state formula.
- If F_1 and F_2 are state formulas, then $F_1 \wedge F_2$, $F_1 \vee F_2$ and $\neg F_1$ are state formulas.
- If F_1 and F_2 are state formulas, then $ax(F_1)$, $ex(F_1)$, $ag(F_1)$, $eg(F_1)$, $au(F_1, F_2)$, and $eu(F_1, F_2)$ are state formulas.

In Figure 2.2 the Kripke structure from Figure 2.1 is converted to an incomplete infinite execution tree. Each path along the tree represents a possible path through the Kripke structure.

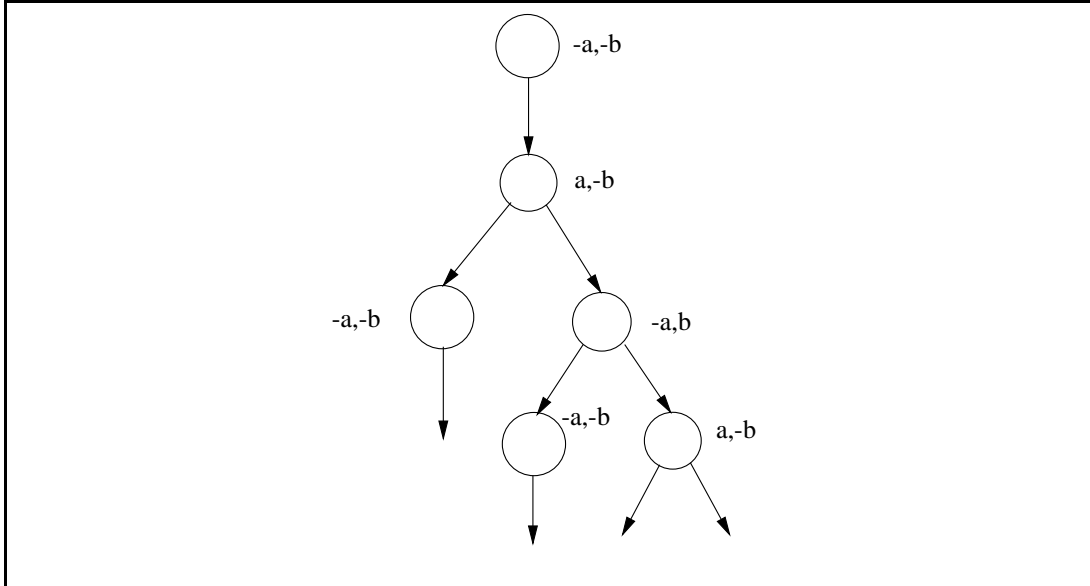


Figure 2.2: The Kripke structure from Figure 2.1 converted to an incomplete infinite computation tree

Model checking in CTL is the problem of deciding whether a Kripke structure, called *model* satisfies a CTL specification. The semantics of the CTL expressions are listed in Figure 2.3. $M, \sigma_0 \vdash F$ should be interpreted as that the CTL expression F is true for the Kripke structure M in state σ_0 .

2.4 An example

Consider a pedestrian crossing with traffic lights. It consists of the following processes:

Traffic lights for cars, states: red, yellow, green

Traffic lights for pedestrians: red, green

Pedestrians button states: active, inactive.

The pedestrians are only allowed to pass if the car lights are red. If the button is pressed then the car lights will go from green to yellow and from

$M, \sigma_0 \models p$	iff $p \in L(\sigma_0)$ where $p \in AP$
$M, \sigma_0 \models F_1 \wedge F_2$	iff $M, \sigma_0 \models F_1$ and $M, \sigma_0 \models F_2$
$M, \sigma_0 \models F_1 \vee F_2$	iff $M, \sigma_0 \models F_1$ or $M, \sigma_0 \models F_2$
$M, \sigma_0 \models \neg F$	iff $M, \sigma_0 \not\models F$
$M, \sigma_0 \models ex(F)$	iff there is a path $\sigma_0, \sigma_1 \dots \in M$ s.t. $M, \sigma_1 \models F$
$M, \sigma_0 \models eg(F)$	iff there is a path $\sigma_0, \sigma_1 \dots \in M$ s.t. $M, \sigma_i \models F, \forall i \geq 0$
$M, \sigma_0 \models eu(F_1, F_2)$	iff there is a path $\sigma_0, \sigma_1 \dots \in M$ and an $i \geq 0$ s.t. $M, \sigma_i \models F_2$ and $M, \sigma_j \models F_1 \forall j \in [0, i)$
$M, \sigma_0 \models ax(F)$	iff for all paths $\sigma_0, \sigma_1 \dots \in M$ it holds that $M, \sigma_1 \models F$
$M, \sigma_0 \models ag(F)$	iff for all paths $\sigma_0, \sigma_1 \dots \in M$ it holds that $M, \sigma_i \models F, \forall i \geq 0$
$M, \sigma_0 \models au(F_1, F_2)$	iff for all paths $\sigma_0, \sigma_1 \dots \in M$ it holds that $\exists i \geq 0$ s.t. $M, \sigma_i \models F_2$ and $M, \sigma_j \models F_1 \forall j \in [0, i)$

Figure 2.3: CTL expressions

yellow to red. Then the button will be inactivated and the pedestrians are allowed to pass.

In the next section this description will be presented in a more formal way.

2.5 Symbolic Model Verifier SMV

The SMV system was developed at Carnegie Mellon University by McMillan et al.[20]. The language SMV is used to describe the transition relation for a Kripke structure and a specification in CTL. The SMV system then takes this description and checks the supplied specification.

The example in Section 2.4 is expressed using SMV in Figure 2.4. The description is expressed using two modules, “main” and “button”. In the main module the state variables are declared along with the instantiation of the module “button”. The “process” keyword means that the assignments in the “button” module are performed asynchronously with the assignments in the main module.

In the “ASSIGN” section the initial values are set for “cl” and “pl”. Observe that “bp” is not assigned an initial value. The next state of each variable is decided by the following assignments.

2.6 Fairness

Consider the traffic light example and the following, quite reasonable, specification: $ag((pedlight = red) \rightarrow af(pedlight = green))$. This translates to “for all paths whenever the pedestrians have a red light then all paths will eventually lead to the pedestrians having a green light”. Or even simpler “the pedestrians always gets green light eventually”.

The problem with this specification is that it is false. Since there are always paths where the pedestrians button never gets active. To fix this we would like to say: The pedestrians button is pressed infinitely often but not always and not necessarily regularly.

There is no way of expressing this in CTL. Instead the concept of fairness is introduced. A fairness constraint is a constraint that should be fulfilled infinitely often. Formally:

Definition 2 *Let S be the set of states. A path π is called fair with respect to one fairness constraint $F^i \subseteq S$ iff some state in F^i occurs infinitely often on π .*

Definition 3 *Let F be a possibly empty list of fairness constraints $F = (F^1, \dots, F^m)$. A path π is fair with respect to F iff π is fair with respect to every F^i .*


```
MODULE main
VAR
  cl : {red, yellow, green};           --Car light
  pl : {red, green};                   --Pedestrian light
  pb : {active, inactive};             --Pedestrian button

  pedestrians_button : process button(pb);
ASSIGN
  --Car light initially green
  init(cl) := green;

  --Pedestrian light initially red
  init(pl) := red;

  --Determine the next state of the car light
  next(cl) := case (cl = green & pb = active) : yellow;
                (cl = green & pb = inactive) : green;
                (cl = yellow & pb = active) : red;
                (cl = yellow & pb = inactive) : green;
                (cl = red & pb = active) : red;
                (cl = red & pb = inactive) : green;
                esac;

  --Pedestrians light is only green if car light is red
  next(pl) := case (cl = red) : green;
                1 : red;
                esac;

  --Set pb inactive if the light is green
  next(pb) := case (pl = green) : inactive;
                1 : pb;

MODULE button(b)
ASSIGN
  next(b) = active;
```

Figure 2.4: SMV example

2.7 BDDs

The compiler is not directly concerned with BDDs. However they are relevant since the performance of the compiler is partly measured by the size of the BDD that is produced from its output. Therefore this section briefly introduces the concept and how it is relevant to this thesis.

Originally BDDs were introduced by Bryant [7]; however this introduction is based on [3].

Some concepts will be introduced and in order to demonstrate them the following Boolean expression will be used as illustration:

$$(a \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge \neg b) \quad (2.1)$$

First the logic formula should be in if-then-else normal form, which is defined by the if-then-else operator:

Definition 4 *let $x \rightarrow y_0, y_1$ be the if-then-else operator defined by: $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$*

Definition 5 *A Boolean expression is in INF (If-then-else Normal Form) if the expression consists entirely of Boolean variables, constants (0 and 1) and the if-then-else operator such that the tests are only performed on variables.*

Proposition 1 *Any Boolean expression is equivalent to an expression in INF.*

This simply means that we can rewrite any Boolean expression to INF. (2.1) can be written as:

$$a \rightarrow (c \rightarrow 1, (b \rightarrow 0, 1)), (b \rightarrow 0, 1) \quad (2.2)$$

Using this representation it is possible to create a binary graph. For each if-then-else operator a node is created. The name of the node is labelled with the if-variable. The then-expression is represented as the solid branch and the else-expression as the dotted branch. Figure 2.5 shows (2.1) represented as a BDD.

Definition 6 A Binary Decision Diagram (*BDD*) is a binary tree where all internal nodes are labelled with binary variables and all leafs are Boolean constants. The outgoing edges of a node u are given by the functions $low(u)$ and $high(u)$ and the variable by $var(u)$.

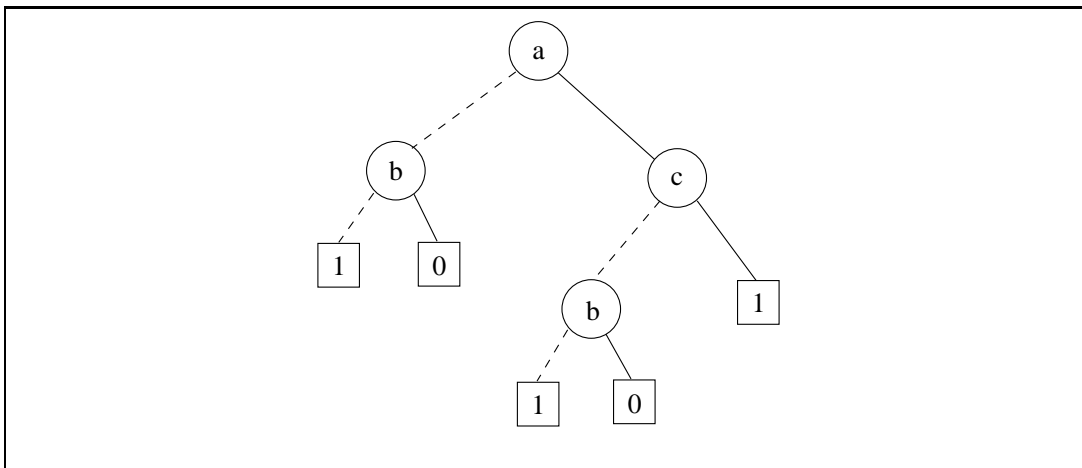


Figure 2.5: (2.1) represented as a BDD

The graph in Figure 2.5 has no specific ordering of the nodes. This can be changed. The nodes b and c can change places. In general a BDD can be transformed into a tree where all nodes are ordered. This is called an OBDD.

Definition 7 An Ordered Binary Decision Diagram is a BDD if for all nodes u given some ordering of the nodes: $u < low(u)$ and $u < high(u)$.

If the graph contains several equivalent sub-graphs (Like the two trees with b as root in Figure 2.5) the graph can be reduced to contain only one of them. If also all nodes with the same high and low successor are removed the result is called a ROBDD.

Definition 8 A Reduced Ordered Binary Decision Diagram is an OBDD where the following holds:

- *uniqueness: no two distinct nodes have the same variable name and low- and high-successor.*

$$\text{var}(u) = \text{var}(v) \wedge \text{low}(u) = \text{low}(v) \wedge \text{high}(u) = \text{high}(v) \Rightarrow u = v$$

- *non-redundancy: tests no variable node u has identical low- and high-successor:*

$$\text{low}(u) \neq \text{high}(u)$$

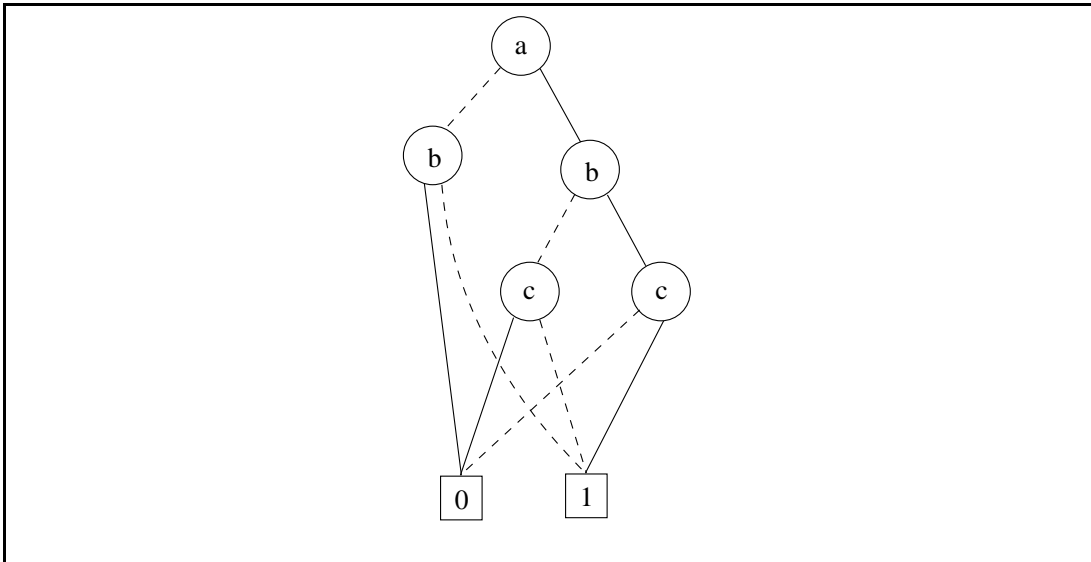


Figure 2.6: (2.1) represented as a ROBDD with ordering $a < b < c$

A ROBDD is a directed acyclic graph and it turns out that it is often a very compact representation of a Boolean function. And there is yet another advantage with this structure as stated in Proposition 2.

Let t^u represent the Boolean expression associated with a ROBDD, then let f^u be a function that maps $(b_1, b_2, \dots, b_n) \in \mathbb{B}^n$ to the truth value of t^u .

Proposition 2 (Canonicity Lemma) *For any function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ there is exactly one ROBDD u with variable ordering $x_1 < x_2 < \dots < x_n$ such that $f^u = f(x_1, \dots, x_n)$.*

In the rest of the report I will adhere to the convention to use the term BDD even though it is really ROBDD that is intended. The reason being that ROBDD is a bulky term.

BDDs are used in symbolic model checking because besides supplying a unique and concise representation of Boolean functions they allow for good performance in the required operations.

2.8 BDD size

Although there is a unique BDD for every Boolean function that is not to say that there is a unique BDD for every finite state machine. A Boolean function $f(x_1, \dots, x_n)$ supplies a specific ordering of the variables. If the ordering is changed, the resulting BDD will also change. In fact the number of nodes can change exponentially given different variable orderings. Figure 2.7 shows (2.1) as a BDD with ordering $a < c < b$. This requires three internal nodes as opposed to five in Figure 2.6.

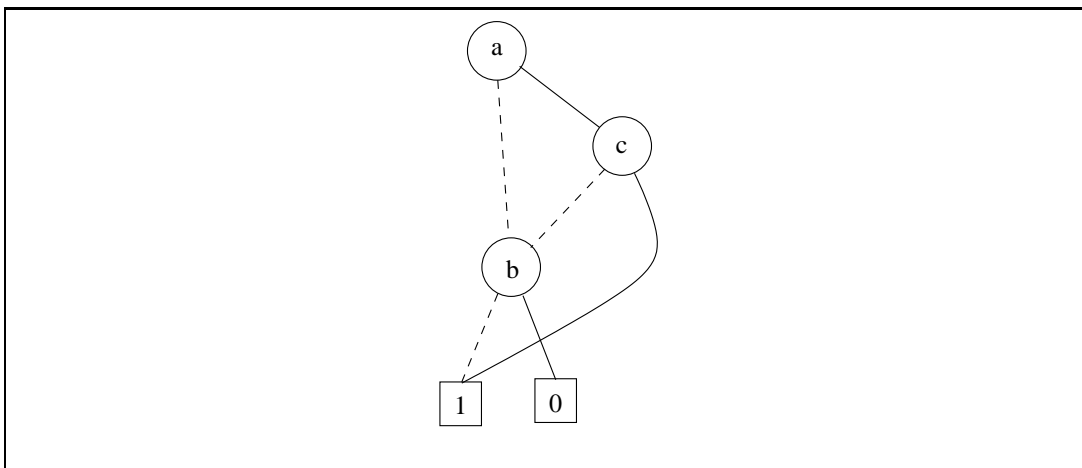


Figure 2.7: (2.1) represented as a ROBDD with order $a < c < b$

The state space of the expression can sometimes be changed without affecting the result of the model checking. The state space of the transition relation is the set of possible transitions. All transitions that go to or from unreachable states do not affect the result of the model checking. Therefore

one tries to change the state space in order to reduce the number of nodes in the BDD. How to change the state space is not obvious. The smallest BDD is created when the state space is maximal (BDD = true) or minimal (BDD = false).

When describing the transition relation using non Boolean variables the expressions must be encoded with a Boolean representation in order to create a BDD. The way this is done will also affect the size of the BDD.

2.9 Constraint Logic Programming CLP

The output of the compiler is a set of CLP(B) clauses. An exact definition of CLP can be found in [17]. This section will just briefly demonstrate the concept.

Each clause in a CLP(B) program will have the form:

$$A_0 : - C, L_1, \dots, L_n$$

where A_0 is an atomic formula, C a Boolean constraint and L_1, \dots, L_n literals; that is, an atomic formula or the negation of an atomic formula. For example a step relation can be expressed as:

$$\begin{aligned} \text{step}([S1, S2, T1, T2]) &: - \text{step}_0(S1, T1) \wedge \text{step}_1(S2, T2). \\ \text{step}_0([S1, T1]) &: - \text{sat}((S1 \wedge \neg T1) \vee (\neg S1)). \\ \text{step}_1([S2, T2]) &: - \text{sat}((S2 \wedge T2) \vee (\neg S2 \wedge T2)). \end{aligned}$$

where the $\text{sat}()$ predicate contains the Boolean constraints. This can for example be used to check if there is a transition between states (0, 1) and (1, 0). So $\text{step}([0, 1, 1, 0])$ is true if $\text{step}_0([0, 1])$ is true (which it is) and $\text{step}_1([1, 0])$ is true (which it is not). Therefore no such transition exists. This process can be carried out by a CLP-solver program such as SICStus Prolog.

The full syntax of the output is a subset of SICStus Prolog syntax and is defined in appendix A.

2.10 Synchronous and asynchronous

As per the definition of the Kripke structure given above it seems void to discuss whether a system is synchronous. However most systems are described using modules that interact with each other.

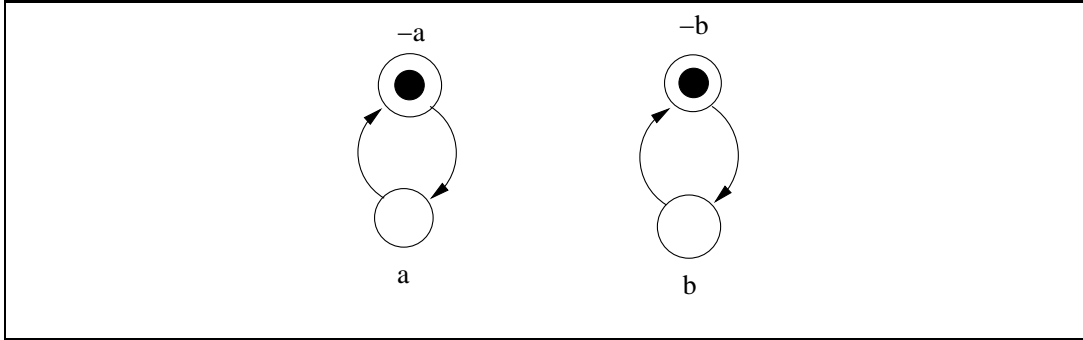


Figure 2.8: Two processes

Figure 2.1 shows two processes. The transitions are unconditional but this is often not the case. In order to express these transitions as a relation two new variables a' and b' are introduced. These represent the next state of the respective variable. Now each process can be described by a transition relation:

$$R_1(a, a') = ((\neg a \wedge a') \vee (a \wedge \neg a')) \quad (2.3)$$

$$R_2(b, b') = ((\neg b \wedge b') \vee (b \wedge \neg b')) \quad (2.4)$$

If we combine the two processes in a synchronous manner the final Kripke structure will have the following transition relation:

$$R_s(a, b, a', b') = R_1(a, a') \wedge R_2(b, b') \quad (2.5)$$

However if we combine them asynchronously the result is:

$$R_a(a, b, a', b') = ((R_1(a, a') \wedge b = b') \vee (R_2(b, b') \wedge a = a')) \quad (2.6)$$

Figure 2.9 shows the final Kripke structure for R_s and R_a respectively. In both cases the number of states is $m \cdot n$ where m and n are the states

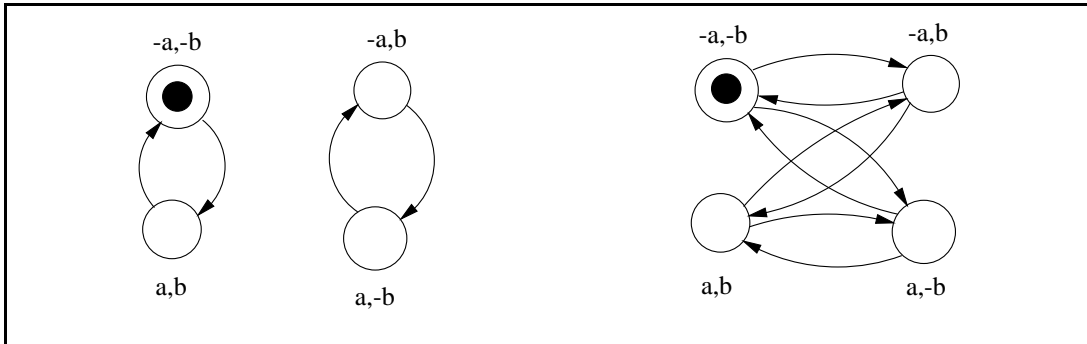


Figure 2.9: Kripke structure for synchronous and asynchronous combination

in each process (in this case two). However the asynchronous case results in a lot more transitions and also reachable states.

Given a set of asynchronous processes it is possible to transform them into a set of synchronous processes without having to construct the final Kripke structure. This is done by adding a transition to each state in the process that goes back to itself. Suppose that the processes in Figure 2.8 are asynchronous. The two synchronous processes can then be seen in Figure 2.10.

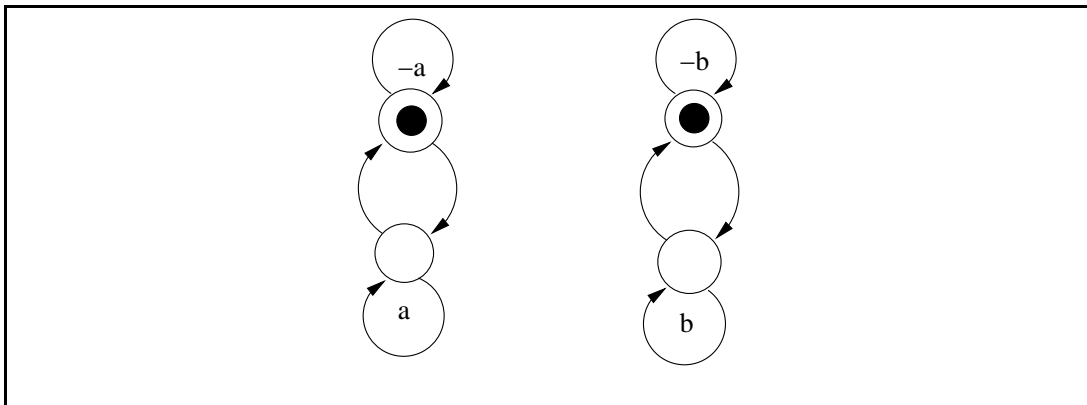


Figure 2.10: Synchronised processes

If these two processes are combined synchronously then the result will be the same as if the two original processes in Figure 2.8 were combined

asynchronously.

2.11 Partitioning the transition relation

The transition relation for the structure in Figure 2.1 can be written:

$$\begin{aligned}
 R(a, b, a', b') = & (\neg a \wedge b \wedge \neg a' \wedge \neg b') \vee \\
 & (\neg a \wedge b \wedge a' \wedge \neg b') \vee \\
 & (a \wedge \neg b \wedge \neg a' \wedge b') \vee \\
 & (a \wedge \neg b \wedge \neg a' \wedge \neg b') \vee \\
 & (\neg a \wedge \neg b \wedge a' \wedge \neg b')
 \end{aligned} \tag{2.7}$$

Where the primed variables represent the next state variables.

The relation in this case consists of a number of disjuncted relations. In general the transition relation can be written:

$$R(X, X') = \bigvee_{\forall i} R_i(X, X') \tag{2.8}$$

Where X is the set of variables in the current state and X' the set of variables in the next state. Also the transition can be partitioned conjunctively:

$$R(X, X') = \bigwedge_{\forall i} R_i(X, X') \tag{2.9}$$

Now let $S(X)$ be a so that it is true if X is in the current state set. Then the next state is:

$$S(X') = \exists X [S(X) \wedge R(X, X')] \tag{2.10}$$

2.11.1 Disjunctive partitioning

Using disjunctive partitioning this can be written:

$$S(X') = \bigvee \exists X [S(X) \wedge R_i(X, X')] \tag{2.11}$$

This is because the \exists operator distributes over disjunction. This means that the next state can be computed for each asynchronous process separately. The monolithic BDD for the transition relation does not need to be constructed and this saves a lot of memory.

2.11.2 Conjunctive partitioning

Unfortunately the above can not be used for conjunctive partitioning. However it is possible to split the computation up by using a technique introduced by Burch et al. [19]. They showed that by using partitioning it was possible to find a much larger reachable state space than previously achieved.

First the n partitions are ordered according to some heuristic. Then let D_i be the variables that the partition R_i is dependent on. And let:

$$E_i = D_i \setminus \bigcup_{k=i+1}^{n-1} D_k \quad (2.12)$$

Now $S(X')$ in Figure 2.10 can be calculated using the following iteration:

$$\begin{aligned} S_1(X, X') &= \exists_{x_j \in E_0} [S(X) \wedge R_0(X, X')] \\ S_2(X, X') &= \exists_{x_j \in E_1} [S_1(X, X') \wedge R_1(X, X')] \\ &\vdots \\ S(X') &= \exists_{x_j \in E_{n-1}} [S_{n-1}(X, X') \wedge R_{n-1}(X, X')] \end{aligned}$$

The only trick is to keep the BDD for $S_i(X, X')$ “small” during the whole iteration. More on ways to accomplish this can be found in Section 3.4

Chapter 3

Literature Survey

This chapter contains a survey of the literature written on the topic of optimising the transition relation with respect to the size of the resulting BDDs. Actually optimising is not a correct term since most of the methods described are heuristics that do not guarantee an optimal result.

Not much work has been done aimed specifically at compilation of discrete systems with the purpose of reducing the BDD size. However most of the methods used for reducing the size of the BDD are applicable at the compilation stage. Therefore some of them are described here.

Also there is of course quite some research done with the purpose of optimising certain properties of hardware synthesised from a given description. These will not be covered here because it is difficult to see how these methods could be applicable.

3.1 Compilation

This section introduces a couple of papers concerned with optimisation of BDDs during the compilation stage.

Aloul et al. [1] have made a compiler for CNF-clauses that changes the variable ordering with good results. They convert a CNF formula into a hyper-graph and reorder the variables using a cut minimisation method.

Finally the result is converted back to CNF. This method seems related to cut minimisation in BDDs but it is applied at an earlier stage.

Cheng and Brayton [10] have constructed a compiler for translating from a subset of Verilog into automata. Verilog is a hardware description language and therefore similar to SMV. However Cheng had to deal with a lot of issues relating to timing which is implicit in SMV. Also no optimisation or partitioning was performed.

3.2 State encoding

The source language SMV supports variables over finite domains other than the Boolean one. These variables must be translated to a set of Boolean variables. However there is no obvious way to do this and furthermore the encoding affects the size of the transition relation and hence also the BDD. There are heuristics [24, 12] that will try to find a good encoding or to change an existing encoding iteratively to reduce the size of the BDD.

The approach of Forth et al. [12] is to locate subtrees within the state transition graph and using them to create an encoding for the states. The drawback of this method is that the transition graph must be constructed.

Meinel and Theobald [24] used a local re-encoding approach. It is applied for two variables neighbouring at a time and the operation is a combination of level exchange and xor-replacement. They have shown how this effects the size of the BDD and it can be used to reduce the BDD size effectively. However since the compiler does not create the BDDs from the transition relation it can not apply this method.

Quer et al. [27] introduced methods for reencoding of a FSM with the purpose of checking equivalence with another FSM.

3.3 Variable removal

There has been some research on removal of redundant variables. Berthet et al. [5] proposed the first algorithm for state variable removal. The algorithm is based on a reencoding of the reachable state space so that redundant variables can be removed.

Sentovich et al. [29] proposed algorithms for variable removal aimed at synthesis. In [8] Eijk and Jess uses functional dependencies to remove variable during state space traversal.

There are many problem descriptions where there are time-invariant constraints (INVAR construct in SMV). These can be utilised to eliminate redundant variables. This is what Yang et al. have showed in [32].

3.4 Clustering and ordering of partitions

The idea to use conjunctive partitioning of the transition relation in symbolic model checking was introduced by Burch et al. [19]. They showed that it was possible to avoid constructing the BDD for the monolithic transition relation as demonstrated in Section 2.11.2.

However they gave no algorithm that could automate the partitioning. Geist and Beer [15] presented a heuristic for automatic ordering of the transition relation.

The heuristic is based on the notion of a unique variable; that is, a variable that does not exist in the other relations. So given a set of partitions first choose the partition with the largest number of unique variables. Remove the partition from the set and recalculate the number of unique variables for each partition in the set. Following this simple heuristic quite reasonable results were achieved.

The transition relation should not be partitioned as much as possible because that will make the implementation slow. The important aspect is to keep the BDDs so small that they are manageable but not smaller. Therefore the partitions should first be clustered into closely related partitions of suitable size and then the clusters should be ordered.

Ranjan Aziz and Brayton [28] supplied a way to both cluster the transition relation and to order them. This method has been very popular and the SMV systems utilises it. An explanation of the heuristic is supplied in Section 5.1

Cabodi et al. [9] modified the heuristic in [28]. The difference being not only using the number of support variables as size limit for clustering but also the actual BDD size. Furthermore the ordering is performed dynamically.

Meinel and Stangier have constructed some improved algorithms based on modularity. First in [22] they presented a heuristic that clusters partitions within a given module in the input language. They improved this in [21] producing a hierarchical partitioning of the relation. Also they supplied a heuristic [23] that does not require the modular information to be given but tries to construct its own modular groups and then clustering within them.

3.5 Partial-Order Reduction

Asynchronous systems are especially difficult to handle because the number of transitions explode when several processes are combined. The reason for this is that any possible interleaving of the transitions must be included.

However most transitions are “independent” of each other in the sense that the order in which they are activated does not affect the reachable states.

Partial order reduction formalises this into a method that can be used to decrease the BDD size quite drastically. Alur et al. [2] showed how this approach can be applied to symbolic model checking. It essentially includes a rewrite of the transition relation.

3.6 Variable ordering

The most important aspect of BDD size is the variable ordering. Heuristics to find good variable ordering have been developed. See [18] for an example of this and more references to relevant papers.

3.7 Don’t Care sets

Shiple et al. [30] investigated some different possibilities to heuristically minimise the size of BDDs using the concept of Don’t Cares. Let F be an incompletely specified function that can be contained by two completely specified functions c and f such that $F \Rightarrow f \vee \neg c$ and $(f \wedge c) \Rightarrow F$ the BDD for F . This is the same as to say that if c is true then we care and $F \equiv f$,

but if c is false then we do not care about the value of F . Since there is a set of functions that can be used to represent F the idea is to choose the function with the smallest BDD.

This technique can be used in symbolic model checking. One approach is to apply the heuristic dynamically when doing reachable state space search. This is done by Wang et al. in [31] where the next state is computed using don't cares. Unfortunately this idea is not directly applicable in the SMV compiler. As stated previously a static method is required in the compiler. One could try to minimise the transition relation given the non-reachable states as a "don't care"-function.

Chapter 4

Compiler basics

This chapter gives an overview on how the smv2clp compiler works and what kind of issues that have been dealt with.

4.1 Architecture

The first task that the compiler must perform is to parse the SMV description file into a data structure that allows for future manipulation. Since the construction of a parser for any language is so similar in structure several tools have been constructed to automate this process.

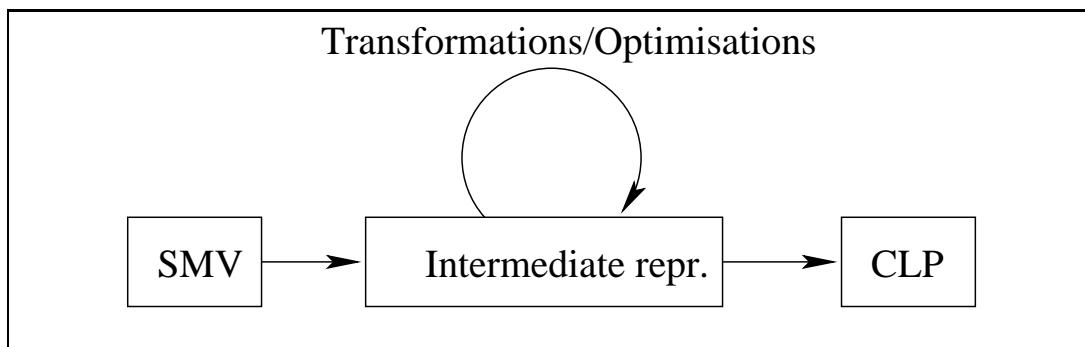


Figure 4.1: Data flow

In this project I have used the tools bison++ (which is bison [13] re-targeted to C++ by Alain Coetmeur) and flex [26]. These produce C++ code which can be integrated with the rest of the compiler.

One of the main objectives of the compiler is to allow for experimentation with different types of translations. A technique called visitor design pattern is used to accomplish this. For an explanation of design patterns see [14].

The main idea behind this technique is to keep the operations in a separate structure from the object structure. At first this seems to go against the usual object oriented philosophy. However this allows for adding new operations on the structure without changing the object class structure.

It is easy to see how this is useful in this particular case. Each optimisation technique is implemented as a visitor that inherits from a base visitor class. Even the step of producing CLP-code from the program structure is implemented as a visitor.

4.2 Programming language

The compiler is implemented in C++. The reasons for using this language are:

- It is object oriented and allows for extensive modularity
- There are parser generators available, e.g. flex and bison++
- It is robust and widely used
- There are good library packages such as STL and Boost

4.3 Intermediate Representation

One of the characteristics of the compiler is that it is possible to create another front end that takes an input language other than SMV. Therefore all optimisations should be performed on an intermediate representation.

This representation should be at least as powerful as the output language. Also it should be a suitable framework for partitioning and ordering the transition relation.

Modularity is an important aspect in many programming languages and this is also true when systems describing discrete systems. Complex systems can more easily be comprehended if divided into modules. Also modularity allows for reuse of code and makes the system less prone to error.

If these were the only reasons for modularity then there would be no reason to let the intermediate representation to be modular. The user will not have to deal directly with this representation. The advantage of keeping the modular structure is that it allows clustering based on the original modular structure such as in [21]. The idea is that the variables that occur in the same module are probably dependent on each other.

The FSM representation is constituted of the following entities:

Environments contains sub environments and constraints. The environment can be synchronous or asynchronous.

Constraints there are different types of constraints but all contain one expression that is either true or false.

Expression can be one of the following: constant, variable, unary, binary or ternary.

4.4 SMV Language constructs

The compiler is able to parse most SMV programs without any problem but there are some language constructs that are not supported. These are listed in Appendix C. This section explains how the compiler deals with the supported parts of the SMV language.

4.4.1 Modules

The SMV language allows for modularisation and a simple form of inheritance. The compiler creates an environment for each module contain-

ing constraints. A module can be instantiated as a synchronous or asynchronous process. The compiler creates an environment for each module. The environments will have the same hierarchy as the modules in the SMV program. An environment can be synchronous or asynchronous and contain a number of constraints.

4.4.2 Declarations

A module is constructed from a number of declarations:

- VAR: All variables must be declared here. Also modules are instantiated.
- SPEC: Contains specifications in CTL. These are represented in the compiler as a *SpecConstraint*.
- ASSIGN: Contains assignments to the initial state, the next state or the current state. These are directly translated into constraints INIT, TRANS and INVAR statements respectively.
- INIT: Contains an expression that must be true for all initial expressions. Represented in the compiler as an *InitConstraint*.
- TRANS: Contains a step relation. Represented in the Compiler as a *StepConstraint*.
- INVAR: Contains a constraint on variables that must be true in all states. Represented in the compiler as as *InvarConstraint*.
- DEFINE: Contains something very similar to macro expressions. These are expanded during translation to the internal representation.

4.4.3 Types

The SMV language supports the following variable types:

- Integer
- Boolean

- Set of atoms and integers
- Array of any type

Integer types have an upper and a lower bound. The Boolean type is really an integer type with 0 and 1 as elements. Sets can have integer and string members. Unfortunately this creates some inconvenience when translating to CLP as explained in Section 4.5.5.

Arrays

SMV supports array types in a very restricted way. A variable that is declared as an array cannot be referenced as is. The only way to set constraints on array variables is to use subscripts. Therefore it is natural to translate the declarations of an array to a set of declaration of variables.

4.4.4 Running

Asynchronous modules can be defined in SMV using the process keyword and the semantics is that given a set of modules the ASSIGN statements are “executed” interleaved in an arbitrary order.

This of course means that a process can be neglected for an infinite number of steps. The interleaving simply chooses some other process to execute at every step.

The solution to this is a variable that is associated with each asynchronous module called “running”. It is therefore possible to supply the following fairness constraint for every asynchronous module.

```
FAIRNESS
  running
```

4.5 Compiler stages

As explained in Section 4.1 the compiler visits the FSM representation a number of times until the CLP output can be produced. They are presented in the order which they are performed by the compiler with the exception of the simple reduction which are performed when needed.

4.5.1 Simple reductions

This section describes some simple reduction that are performed a number of times during the compilation.

Not reduction All expressions can be negated. This stage reduces expressions so that negation only occurs in front of a variable. This can be very useful when transforming expressions.

True and false reduction Many of the compiler stages leave expressions such as $true \wedge a$ and this stage simply reduces to an equivalent expression without true or false.

Operator reduction Once all expressions and variables are mapped into the Boolean domain there is only a small step left to be done before the relations can be written as CLP(B) sentences. SMV supports a wide range of operators but the target language only supports: and, or, xor, not and exists.

Some operators are eliminated during the state encoding (such as multiplication, division, set union, etc.). The rest are treated as in table 4.1

Operator	Translated
$a = b$	$\text{not}(\text{xor}(a,b))$
$a \Leftrightarrow b$	$\text{not}(\text{xor}(a,b))$
$a \neq b$	$\text{xor}(a,b)$
$a \leq b$	$\text{or}(\text{not}(a),b)$
$a \Rightarrow b$	$\text{or}(\text{not}(a),b)$
$a < b$	$\text{and}(\text{not}(a),b)$
$a > b$	$\text{and}(a, \text{not}(b))$
$a \geq b$	$\text{or}(a, \text{not}(b))$

Table 4.1: Operator translation

4.5.2 Lift case expressions

A case statement containing Boolean expressions can easily be converted to a Boolean expression as shown in Figure 4.2.

<pre> INVAR case a : b; c : d; esac; </pre>	$(a \wedge b) \vee (\neg a \wedge c \wedge d) \vee (\neg a \wedge \neg c)$
--	--

Figure 4.2: Translation of Boolean only case statement

In SMV however the case statement can be of any type. This makes it harder to do the encoding properly and therefore the case statements are “lifted” so that the case statements are always Boolean-valued. An example of this is shown in Figure 4.3.

<pre> VAR a : {1,2,3,4}; b : Boolean; c : Boolean; ASSIGN a := case b : 1; c : 2; 1 : {3,4}; esac; </pre>	<pre> INVAR case b : a = 1; c : a = 2; 1 : a = {3,4}; esac; </pre>
---	--

Figure 4.3: Translation of case statement

4.5.3 Lift (reduce) set expressions

Set expressions can easily be converted to a disjunction of equalities. For example:

$$\text{next}(a) \in \{1, 2, X\} \equiv (\text{next}(a) = 1) \vee (\text{next}(a) = 2) \vee (\text{next}(a) = X)$$

```

GETASSIGNMENTS(expr)
1  lhsDomain ← GETDOMAIN(expr.lhs)
2  rhsDomain ← GETDOMAIN(expr.rhs)
3  returnExpr ← false
4  for i ← lhsDomain.first to lhsDomain.last
5  do for j ← rhsDomain.first to rhsDomain.last
6     do if EVAL(i expr.operator j)
7         then returnExpr ←
8             (returnExpr ∨
9              ((expr.lhs = i) ∧ (expr.rhs = j)))
10 return returnExpr

```

Figure 4.4: Set encoding algorithm for comparison expressions

4.5.4 Solve finite domain constraints

It is possible to supply arithmetic expressions in SMV. These expressions can be converted directly to Boolean expressions but it is not a trivial task to do so. Instead the solution used by `smv2clp` is to “Solve” the constraints by first converting them to a number of assignments. This is done using the algorithm in Figure 4.4. The algorithm basically goes through all possible values for the expressions right hand side (*expr.rhs*) and the left hand side (*expr.lhs*). If the expression is true then this combination is added to the possible assignments.

4.5.5 Create Boolean encoding

In order to convert all expressions into Boolean expressions it is necessary to decide what encoding to use. In some cases the expression can not easily be converted into a Boolean expression without first rewriting the expression

Naturally we do not want to add more variables than absolutely necessary because of state explosion. A variable with a domain of size n results in at least $\lceil \log_2(n) \rceil$ Boolean variables.

SMV is not a strongly typed language and allows comparison and as-

signment between variables of different types. Also integer arithmetic is supported. The following section is OK in SMV

```
VAR
  a : {A,B};
  b : {B,C};
ASSIGN
  a := B;
  b := B;
```

It is easy to see that the two assignments must be converted using different encodings. Fortunately the stage described in Section 4.5.4 ensures that all non Boolean expressions have been reduced to these simple assignments.

When creating a Boolean encoding for a non-Boolean variable the number of new variables is $\lceil \log_2(n) \rceil$ where n is the size of the source domain. This results in a number of illegal values that must be considered. So for each variable that create these illegal values a constraint is constructed saying that the variables cannot take this value.

This constraint must be added in all modules where the variable is updated. The reason for this is that the description can be asynchronous.

4.5.6 Reduce INVAR

Usually a Kripke structure is specified by initial states and a transition relation. However one can given a specification reduce the Kripke structure by specifying constraints on the states. In SMV this is done using declarations in INVAR or by using the ASSIGN and assigning to current state.

The compiler replaces all INVAR constraints with one INIT and one TRANS constraint. If the constraint $R_i(x_1, x_2, \dots, x_n)$ is an INVAR constraint it is replaced by $R(next(x_1), next(x_2), \dots, next(x_n))$ as a TRANS constraint and $R_i(x_1, x_2, \dots, x_n)$ as an INIT constraint.

4.5.7 Optimisation

There are two optimisation stages. One that tries to reduce the variables involved in the transition relation. (See Section 5.2.)

The other optimisation stage is aimed at partitioning the transition relation and ordering the partitions according to a given heuristic. (See Section 5.1.)

4.5.8 Handle running

As described in Section 4.4.4 each asynchronous process is associated with one running variable. This step ensures that this is included in the FSM representation. In every asynchronous environment a running variable is created. All running variables are false initially. Then in every environment i the following constraint is added:

$$\forall \text{next}(\text{running}_i) \rightarrow \left(\bigwedge_{\forall j \neq i} \neg \text{running}_j \right)$$

This means that only one running variable can be true at any given step. Also if one running variable is true then all the constraints in the associated environment must also be true.

This step is also responsible for adding non-change constraints to the environment. In SMV the step relation in an asynchronous module can only be described using the “ASSIGN” construct. This means that only assigned variables are changed. This must be reflected in the CLP program and so the following constraint is added to each asynchronous environment i :

$$\bigwedge_{\forall \text{var}_j \notin NS(i)} \text{var}_j = \text{next}(\text{var}_j)$$

where $NS(i)$ are all variables that are constrained in the next state in environment i .

4.5.9 Synchronise

This is an optional step which will transform all asynchronous processes into synchronous. This is again done using the “running” variable. Replace each step constraint R_j in the asynchronous environment i with:

$$R_j \vee \neg \text{running}_i$$

One must also ensure that one of the processes is always running. This is achieved by adding the following transition relation:

$$\bigvee_{\forall i} next(running_i)$$

4.6 Output

The final step is to produce the output. Currently the compiler supports two types of output: CLP(B) and SMV.

4.6.1 CLP

The CLP output consists of a number of clauses. For a resolution engine to successfully check the model a number of predicates are also needed.

This section introduces some of the predicates needed to solve the CLP programs that the compiler produces. Moreover it contains a description of the predicates that are produced.

The Kripke structure is expressed using the *step* and *sat* predicates. An example of this can be seen in Section 2.9. The *sat* relation will only contain Boolean constraints over state variables.

The CTL specifications are expressed using the *holds* predicate. Each state variable is considered a property. The name of this property is the same as the name of the Boolean variable in SMV. Finally a query is produced that asks if the specifications are true for the initial states.

$$\mathit{holds}(var_1) : - \mathit{sat}(S1).$$

$$\mathit{holds}(var_2) : - \mathit{sat}(S2).$$

$$\mathit{holds}(\mathit{init}, [S1, S2]) : - \mathit{sat}(\mathit{and}(S1, S2))$$

$$\mathit{spec}_0([S1, S2]) : - \mathit{holds}(\mathit{ag}(var_1 \wedge var_2), [S1, S2]).$$

$$\mathit{query}([S1, S2]) : - \mathit{holds}(\mathit{init}, [S1, S2]), \mathit{spec}_0([S1, S2]).$$

The CLP(B) syntax is described in appendix A and the predicates necessary are listed in appendix B.

4.6.2 SMV

Producing SMV output is only possible for synchronous systems. This is due to SMV not being able to handle the TRANS construct in asynchronous modules. SMV requires the transition relation to be specified using ASSIGN statements and the translation required is not implemented. Therefore the compiler always transforms the model into a synchronous equivalent before creating the SMV representation.

Note that the resulting output is not nearly as readable as the input. This is partly because the output is always expressed using Boolean variables but also because the variable names are automatically generated rather than chosen for readability.

Chapter 5

Optimisations

The compiler uses two optimisation techniques: clustering and ordering of conjunctive partitions and state space reduction. The clustering and ordering is implemented according to the heuristic supplied by [28] which has proved to be quite successful and is utilised by NuSMV [11] and VIS [6]. The variable reduction is in essence a rewrite of the problem into a simpler one which should allow for faster verification.

5.1 Clustering and ordering

The partitioning technique used by the compiler is based on [28]. The algorithm maintains two sets of partitions, one with the already ordered partitions and one with those not yet ordered. In each step all partitions in the unordered set are evaluated using a heuristic function which takes into account the number of quantifiable variables, next state variables and the variable ordering. The partition with the best value is moved to the ordered set.

Then the partitions in the ordered set are merged until a given limit of the BDD representing them is reached. Since the `smv2clp` compiler does not keep a BDD representation of the relations the limit is not set by BDD size but by the number of variables. This is a crude substitute since the

BDD size can vary significantly using the same number of variables but it keeps the BDDs bounded.

5.2 State space reduction

This optimisation is aimed at reducing the Kripke structure so that the number of variables needed is reduced. Section 3.3 mentions some articles written on the subject of removing variables by finding redundant variables. The approach taken here is somewhat different even though the results can be very similar.

- Let $M = (S, s_0, R, L)$ be a Kripke structure and AP_M the set of atomic propositions that is associated with it.
- Then let $AP_{M'} \subset AP_M$ and let $H : S \rightarrow 2^{AP_{M'}}$ be such that $H(s) = L(s) \cap AP_{M'}$.
- Create a set $S' = \{s' : s' \in 2^S \wedge (s_i, s_j \in s' \Leftrightarrow H(s_i) = H(s_j))\}$ and let $L' : S' \rightarrow 2^{AP_{M'}}$ be such that $L'(s') = H(s_i)$ for $s_i \in s'$.
- Then create the Kripke structure $M' = (S', s'_0, R', L')$ where $s'_0 \in S'$ contains S_0 and $R' \subseteq S' \times S'$ is defined as $((s_i \in s'_i) \wedge (s_j \in s'_j) \wedge R(s_i, s_j)) \Rightarrow (R'(s'_i, s'_j))$.
- The structure M' will be called a sub-circuit of M .

Intuitively a sub-circuit is constructed by taking a subset of the variables and all the transition relations that update these variables. The idea of the heuristic is to locate a small sub-circuit that can be minimised and reencoded and thereby saving variables.

To exemplify the simple but unrealistic system in Figure 5.1 is used. Each combination of variables and updating transition relations constitutes a sub-circuit. It should not take long to realise that in the example the variables a and b always have the same value, $a = b$ in all states. Therefore we could remove one of the variables by substituting all occurrences of it with the other variable. Thus we save one variable. If the example given is

```
MODULE cproc
VAR
  c : Boolean;
ASSIGN
  init(c) := 0;
  next(c) := !c;

MODULE main
VAR
  a : Boolean;
  b : Boolean;
  d : process cproc;
ASSIGN
  init(a) := 0;
  init(b) := 0;
  next(a) := case a&b : 0;
    !a & !b : 1;
  esac;
  next(b) := case a&b : 0;
    !a & !b : 1;
  esac;
```

Figure 5.1: SMV example with one redundant variable

part of a bigger system then that saved variable can have a very significant effect on the final result.

The variable reduction performed by the compiler is not just aimed at redundant variables as described in Section 3.3. The simple example in Table 5.1 illustrates this. None of the variables are redundant but still there is one variable more than needed.

The algorithm used for the variable reduction is shown in Figure 5.2. The partitions are collected in groups such that for each variable that is updated there is a group containing all partitions that are active in that update. For the example in Figure 5.1 there is one relation for each variable.

var_1	var_2	var_3
0	0	1
0	1	0
1	0	0
0	0	0

Table 5.1: No redundant variables

But in some cases there are several constraints for the same variable in the next state and then the reachable states cannot be calculated without including all of these constraints. In the following subsections the rest of the algorithm is described in more detail.

```

REDUCEVARIABLES()
1   $P \leftarrow \{ \text{The set of step constraints} \}$ 
2   $V \leftarrow \{ \text{The set of variables} \}$ 
3   $S \leftarrow \{s \subseteq P : \exists v \in V[(p \in s) \Leftrightarrow v \in \text{Updates}(p)]\}$ 
4   $R \leftarrow \text{FINDPOSSIBLEREDUCTIONGROUPS}(S)$ 
5  for  $i \leftarrow R.begin$  to  $R.end$ 
6  do  $States \leftarrow \text{FINDREACHABLESTATES}(\text{InitConstraints}, i)$ 
7      $\text{SavedVars}[i] = \text{GETSAVEDVARS}(States)$ 
8      $\text{ReEncoding}[i] = \text{GETREENCODING}(States)$ 
9   $\text{ReducedVars} \leftarrow \{\}$ 
10 while  $|\text{SavedVars}| \neq 0$ 
11 do  $x \leftarrow i : \text{MAX}(|\text{SavedVars}[i]|)$ 
12      $\text{REENCODE}(\text{ReEncoding}[x])$ 
13      $\text{ReducedVars} \leftarrow \text{ReducedVars} \cup \text{GETVARIABLES}(x)$ 
14      $\text{SavedVars} \leftarrow \text{SavedVars} \setminus \{\text{SavedVars}[x]\}$ 

```

Figure 5.2: Algorithm for variable reduction

5.2.1 Finding suitable reduction groups

The goal of this step is to find sub-circuits where the number of reachable states is less than half of the total number of states. This is done by merging

partitions into clusters. The overall algorithm is shown in Figure 5.3.

In each step the two clusters with the highest pairing heuristic value are merged. The heuristic that determines the value of merging two partitions simply measures how the number of non-updated variables decreases. There is also a check so that the partitions do not grow too large.

For every non-updated variable in the cluster it must be assumed that it can take any value at any time during the state space search. Therefore if the non-updated variables are as few as possible the chance of finding a small state space increases.

```

FINDPOSSIBLEREDUCTIONGROUPS( $S$ )
1   $R \leftarrow \emptyset$ 
2   $R_{size} \leftarrow 0$ 
3  for  $i \leftarrow S.begin$  to  $S.end$ 
4  do if  $CheckLimits(i)$ 
5      then  $R \leftarrow R \cup \{i\}$ ;
6  while  $|R| \neq R_{size}$ 
7  do  $R_{size} \leftarrow |R|$ 
8      for  $i \leftarrow R.begin$  to  $R.end$ 
9      do for  $j \leftarrow R.begin$  to  $R.end$ 
10         do if  $i \neq j$ 
11             then  $Pairing(i, j) \leftarrow HEURISTIC(i, j)$ 
12          $[a, b] \leftarrow [i, j] : MAX(Pairing(i, j))$ 
13          $R \leftarrow R \cup MERGE(a, b)$ 
14          $R \leftarrow R \setminus \{a, b\}$ 
15 return  $R$ 

```

Figure 5.3: Algorithm for finding reduction groups

5.2.2 Finding reachable states

First of all constraints are converted into DNF. Starting with the init constraints each DNF term can be translated into a set of states fulfilling that particular term. This is done for all DNF terms describing the initial states.

Then the step relations are translated into a set of step rules each containing just one state in its pre-image and one state in the image. These are applied to the current set of reachable states until no more reachable states can be found.

5.2.3 Reencoding

Once the reachable state set for a given set of constraints has been found the question arises how to take advantage of this. Consider the simple example in Table 5.2. Since we have only three states we should be satisfied with only two variables. Again we cannot simply remove one of them since none of them is constant for all reachable states.

var_1	var_2	var_3
0	0	1
0	1	0
1	0	1

Table 5.2: Three variables, with reachable state space of size three

Instead we create a new encoding of the state space using variables $nvar_1$ and $nvar_2$ as in Table 5.3.

var_1	var_2	var_3	$nvar_1$	$nvar_2$
0	0	1	0	0
0	1	0	0	1
1	0	1	1	0

Table 5.3: A more efficient representation of the state space

Now it is possible to say that for all reachable states:

$$var_1 \equiv (nvar_1 \wedge \neg nvar_2)$$

$$var_2 \equiv (\neg nvar_1 \wedge nvar_2)$$

$$var_3 \equiv (\neg nvar_1 \wedge \neg nvar_2) \vee (nvar_1 \wedge \neg nvar_2)$$

So for each variable we get a DNF expression that we can replace it with in the contexts where it is referenced. Also the step-relations that were analysed must of course be replaced. Hopefully several of the reduction clusters result in reencodings of the variables. If this is the case then there can be an overlap of variables that are to be reduced in the different reencodings. Therefore the reencoding which saves the most variables is applied first and then the next encoding is chosen so that no variable is removed twice.

Chapter 6

Results

This chapter contains some comparisons of BDD size and execution time when verifying the output of the compiler.

6.1 Metric

Although the compiler is aimed at producing efficient CLP(B) output there are currently no systems that are suitable for using as benchmarking utilities. Therefore the compiler is also able to produce SMV output.

The SMV output is very similar to the CLP(B) output. All variables are Boolean. There are no INVAR expressions and the modular structure is similar to the way the CLP description is built up from clauses. Moreover all problems are converted to synchronous equivalents.

In the report the focus has been on minimising the size of the BDDs produced during the fix-point evaluation. The reason of course being that if the BDDs grow too large the computer will run out of memory and model checking will not be possible.

However it is also interesting to look at the execution time needed to perform the model checking. This is apparent when doing conjunctive clustering where there is a tradeoff between memory usage and execution speed.

When doing variable reduction on a given system the problem is converted into an equivalent but simpler description. This will take some time to do for the compiler. Now if the compiler takes a longer time to convert the problem than for SMV to solve it one could argue that the optimisation is void. However the compiler is not designed for speed and the variable reduction can be done more efficiently. The interesting part is to see whether it is possible to perform these reductions without too much effort.

6.2 Comparisons

The results obtained when the compiler does not perform any optimisations at all can be seen in Table 6.1. Or at least no optimisations with the purpose to affect the size of the BDDs constructed. The reason for this comparison is to see how the results differ from when the compiler must transform the problem considerably to be able to convert it to CLP(B).

Table 6.2 shows the results from SMV¹ when verifying the unoptimised as well as the optimised output from the compiler. The “-” indicates that the execution timed out after an hour.

Unfortunately there was no easy way to get SMV to accept the partition produced with the clustering and ordering heuristic described in Section 5.1. Therefore there are no results from that optimisation step. However this is a well known heuristic and the results should be similar. Most of the problems are the examples that come bundled with the NuSMV package.

A short description of each problem follows:

abp4 Alternating Bit Protocol, four agents

abp_ctl Alternating Bit Protocol, sender and receiver

counter A synchronous three bit counter.

dme1 A synchronous version of a distributed mutual exclusion algorithm with three cells

¹SMV was used for all problems except dme2 where NuSMV was used because of incompatibility

dme2 A asynchronous version of a distributed mutual exclusion algorithm with three cells

mutex A simple mutual exclusion algorithm.

mutex1 Another simple mutual exclusion algorithm

p-queue A priority queue example.

ring An asynchronous three bit counter.

semaphore A model of two processes synchronized using a semaphore.

syncarb5 A model of a synchronous arbiter with 5 elements.

Problem	Original		Compiled	
	BDD size (nodes)	time (s)	BDD size (nodes)	time (s)
abp4	20420	0.84	216108	2.42
abp_ctl	2348	0	1689	0
counter	830	0	968	0
dme1	478575	4.51	-	-
dme2	269472	0.68	421051	1.11
mutex	619	0	766	0
mutex1	2582	0	3662	0
p-queue	1054613	32.79	97524	0.4
ring	268	0	680	0
semaphore	685	0	2112	0
syncarb5	1830	0	2982	0

Table 6.1: Comparison between original problem and compiled problem

6.3 Interpretation

From Table 6.1 one can see that there is no obvious way on how the compilation affects the size of the problem. No specific variable ordering has been applied in any of the cases. Since a new set of variables are created

Problem	No optimisation		Reduced	
	BDD size (nodes)	time (s)	BDD size (nodes)	time (s)
abp4	216108	2.42	110062	12.71
abp_ctl	1689	0	1689	0
counter	968	0	968	0
dme1	-	-	292254	4.34
dme2	421051	1.11	170848	0.473
mutex	766	0	822	0.1
mutex1	3662	0	3654	0
p-queue	97524	0.4	97524	0.4
ring	680	0	680	0
semaphore	2112	0	2328	0
syncarb5	2982	0	2982	0

Table 6.2: Effect of variable reduction

during the compilation this is probably one of the main reasons for the different results.

The “dme1” example shows that in some cases the compiler makes the situation much worse than in the original case. On the other hand one sees that the “p-queue” case is drastically improved by the compiler. Both these cases could be the result of different variable ordering. In the “p-queue” case it is also possible that the finite domain solver has helped boost performance since there are some arithmetic constraints involved.

The effects of the variable reductions that are seen in Table 6.2 also varies for different problems. Here it seems as if the greatest savings are achieved for asynchronous systems however some saving can be achieved for synchronous systems too. Also bigger problems tend to be easier to optimise than smaller ones. For many problems the heuristic fails to find suitable reduction groups and thus the problem remains unchanged. Whether this is due to an inefficient heuristic or that the problem contains no such groups is hard to say.

If the uncompiled problems (first two columns of Table 6.1) are compared with the compiled and optimised ones (the last two columns of Table 6.2) the compiler seems to do well enough.

Chapter 7

Discussion

This chapter contains a discussion on the compiler and the results presented in Chapter 6. It also contains a presentation what kind of improvements that can be done to the compiler.

7.1 The compiler

The SMV language is not very complex but is very good for describing discrete systems. Unfortunately there is no complete description of the language since the manual lacks some constructs that are supported by the SMV system. The architecture used has turned out to work very well. Different rewrites can be applied independently of each other and new optimisations can be introduced without having to change the intermediate representation.

The lack of strong typing is a problem when translating into Boolean expressions. The solution used by `smv2clp` to solve all finite domain constraints works well for small problems but could be a problem for complex arithmetic constraints. Complex arithmetic constraints should probably be avoided anyway because of the blowup in BDD size for integer multiplication.

Although SMV is modular in its appearance there is no encapsulation

and it would probably have been easier to flatten the structure entirely before translating it. The advantage of the current implementation is that a partitioning strategy based on modularity could very easily be added.

7.2 Performance

As seen in Table 6.1 the execution time and BDD size can be greatly affected by the translation done by the compiler. There are several possible reasons for this.

Encoding NuSMV utilises a different encoding scheme than this compiler and therefore there will be some differences.

Integer arithmetic The finite domain solver translates into equalities and Boolean connectives which should improve performance.

Variable ordering The translated description uses a different set of variables and therefore the ordering probably differs.

INVAR reduction The compiler converts these into init and step constraints and the SMV system could handle them differently.

In general it seems that the compiler in some cases makes the problem harder and in some cases easier. This is expected and though it would be better if the complexity was not altered at all it can be considered acceptable.

Fortunately it seems that there is little risk in applying the variable reduction optimisation. The reduction will if successful in most cases make the problem easier to verify.

7.3 Applicability in real world cases

The compiler has no difficulty translating complex problems into CLP. The heuristic however is not always able to find partitions so that the number of variables can be reduced. The compiler has a limit set on the size of the possible reduction groups. This is to ensure that the state space search does not take too much resources in time and space.

The heuristic does not suffer much from the global size of the problem but rather the size of the smallest partitions. If the partitions themselves are greater in size than the heuristic allows then obviously no reduction groups will be found. So the limitation is primarily due to the structure of the problem rather than the total size.

7.4 Future work

When doing the variable reduction optimisation the state space is explored using explicit state space traversal. One of the main bottlenecks of the compiler seems to be to the DNF representation. But obviously the state space search itself is also a major limitation.

A symbolic approach would allow for reduction of larger groups and thus probably would allow for more variables to be reduced. However since the compiler is not intended to take over the job of the fix point engine but merely to rewrite the problem description into a less complex one.

Most of the techniques described in Chapter 3 could be applied in this compiler. The partial-order reduction seems especially suited for the compiler since it is possible to apply it using rewrites of the logic formulae.

It would be of interest to see if the heuristic for finding reduction groups finds all the possible reduction groups below the specified size limit.

Appendix A

CLP syntax

This section describes a slightly modified subset of Prolog. The constraints are Boolean constraints and the special predicate `sat/1` should be used to express them. The `sat` should take as argument a constraint expression which can be either a variable, the constants 0 and 1 or one of the predicates `not/1`, `or/2`, `and/2`, `xor/2`, `exists/2`.

Figure A.1 describes the syntax on token level. That is whitespace and newlines are ignored.

Figure A.2 describes the syntax on string level. The character types referenced are specified as:

small-letter character codes 97..122 (a-z)

capital-letter character codes 65..90 (A-Z)

digit character codes 48-57 (0-9)

underline character code 95 (`_`)

$\langle \text{clause} \rangle$	\rightarrow	$\langle \text{non-unit-clause} \rangle \mid \langle \text{unit-clause} \rangle$
$\langle \text{non-unit-clause} \rangle$	\rightarrow	$\langle \text{head} \rangle \text{' :- ' } \langle \text{body} \rangle \text{' . '}$
$\langle \text{unit-clause} \rangle$	\rightarrow	$\langle \text{head} \rangle \text{' . '}$
$\langle \text{head} \rangle$	\rightarrow	$\langle \text{atom} \rangle$
$\langle \text{literal} \rangle$	\rightarrow	$\langle \text{atom} \rangle \mid \text{' \+ ' } \langle \text{atom} \rangle$
$\langle \text{body} \rangle$	\rightarrow	$\langle \text{literal} \rangle$
		$\mid \langle \text{literal} \rangle \text{' , ' } \langle \text{body} \rangle$
$\langle \text{atom} \rangle$	\rightarrow	$\langle \text{name} \rangle \text{' (' } \langle \text{arguments} \rangle \text{') '}$
$\langle \text{term} \rangle$	\rightarrow	$\langle \text{atom} \rangle$
		$\mid \langle \text{list} \rangle$
		$\mid \langle \text{constant} \rangle$
		$\mid \langle \text{variable} \rangle$
$\langle \text{arguments} \rangle$	\rightarrow	$\langle \text{term} \rangle$
		$\mid \langle \text{term} \rangle \text{' , ' } \langle \text{arguments} \rangle$
$\langle \text{list} \rangle$	\rightarrow	' [' '] '
		$\mid \text{' [' } \langle \text{listexpr} \rangle \text{'] '}$
$\langle \text{listexpr} \rangle$	\rightarrow	$\langle \text{term} \rangle$
		$\mid \langle \text{term} \rangle \text{' , ' } \langle \text{listexpr} \rangle$
		$\mid \langle \text{term} \rangle \text{' ' } \langle \text{term} \rangle$
$\langle \text{constant} \rangle$	\rightarrow	$\langle \text{name} \rangle \mid \langle \text{number} \rangle$
$\langle \text{number} \rangle$	\rightarrow	$\langle \text{unsigned-number} \rangle$
$\langle \text{unsigned-number} \rangle$	\rightarrow	$\langle \text{natural-number} \rangle$

Figure A.1: CLP-syntax on token level

$\langle \text{name} \rangle$	\rightarrow	$\langle \text{word} \rangle$
$\langle \text{word} \rangle$	\rightarrow	$\langle \text{small-letter} \rangle \text{ ? } \langle \text{alpha} \rangle \dots$
$\langle \text{natural-number} \rangle$	\rightarrow	$\langle \text{digit} \rangle \dots$
$\langle \text{variable} \rangle$	\rightarrow	$\langle \text{capital-letter} \rangle \text{ ? } \langle \text{alpha} \rangle \dots$
$\langle \text{alpha} \rangle$	\rightarrow	$\langle \text{capital-letter} \rangle$
		$\mid \langle \text{small-letter} \rangle$
		$\mid \langle \text{digit} \rangle$
		$\mid \langle \text{underline} \rangle$

Figure A.2: CLP-syntax on string level

Appendix B

Predicates

$holds(and(F1, F2), S) \leftarrow holds(F1, S), holds(F2, S).$

$holds(or(F1, F2), S) \leftarrow holds(F1, S).$

$holds(or(F1, F2), S) \leftarrow holds(F2, S).$

$holds(not(F1), S) \leftarrow \neg holds(F1, S).$

$holds(xor(F1, F2), S) \leftarrow holds(F1, S), \neg holds(F2, S).$

$holds(xor(F1, F2), S) \leftarrow holds(F2, S), \neg holds(F1, S).$

$holds(eg(F), S) \leftarrow holds(not(af(not(F))), S).$

$holds(ex(F), S1) \leftarrow step(S1, S2), holds(F, S2).$

$holds(eu(F, G), S) \leftarrow holds(G, S).$

$holds(eu(F, G), S) \leftarrow holds(F, S), holds(ex(eu(F, G)), S).$

$holds(ax(F), S) \leftarrow holds(not(ex(not(F))), S).$

$holds(ef(F), S) \leftarrow holds(eu(true, F), S).$

$holds(af(F), S) \leftarrow holds(F, S).$

$holds(af(F), S) \leftarrow holds(ax(F), S).$

$holds(ag(F), S) \leftarrow holds(not(ef(not(F))), S).$

$holds(au(F1, F2), S) \leftarrow holds(and(not(eu(not(F2), and(not(F1), not(F2)))), not(eg(not(F2)))), S).$

$holds(ecg(F), S)$	\leftarrow	$fairness(L), holds(ecg(F, L), S).$
$holds(ecx(F), S)$	\leftarrow	$fairness(L), holds(ecx(F, L), S).$
$holds(ecf(F), S)$	\leftarrow	$fairness(L), holds(ecf(F, L), S).$
$holds(ecu(F1, F2), S)$	\leftarrow	$fairness(L), holds(ecu(F1, F2, L), S).$
$holds(acg(F), S)$	\leftarrow	$fairness(L), holds(acg(F, L), S).$
$holds(acx(F), S)$	\leftarrow	$fairness(L), holds(acx(F, L), S).$
$holds(acf(F), S)$	\leftarrow	$fairness(L), holds(acf(F, L), S).$
$holds(acu(F1, F2), S)$	\leftarrow	$fairness(L), holds(ecu(F1, F2, L), S).$
$holds(ecg(F, []), S)$	\leftarrow	$holds(eg(F), S).$
$holds(ecg(F, [H T]), S)$	\leftarrow	$path(F, S1, S2), holds(H, S2), step(S2, S3),$ $path(F, S3, S2), holds(ecg(F, T), S).$
$holds(true, S).$		
$equal([], []).$		
$equal([H1 T1], [H2 T2])$	\leftarrow	$sat(not(xor(H1, H2))), equal(T1, T2).$
$path(F, S1, S2)$	\leftarrow	$equal(S1, S2), holds(F, S1).$
$path(F, S1, S3)$	\leftarrow	$holds(F, S1), step(S1, S2), path(F, S2, S3).$
$holds(fair(L), S)$	\leftarrow	$holds(ecg(true, L), S).$
$holds(ecx(F1, L), S)$	\leftarrow	$holds(ex(and(F1, fair(L))), S).$
$holds(ecu(F1, F2, L), S)$	\leftarrow	$holds(eu(F1, and(F2, fair(L))), S).$
$holds(acx(F, L), S)$	\leftarrow	$holds(not(ecx(not(F), L)), S).$
$holds(ecf(F, L), S)$	\leftarrow	$holds(ecu(true, F, L), S).$
$holds(acf(F, L), S)$	\leftarrow	$holds(not(ecg(not(F), L)), S).$
$holds(acg(F, L), S)$	\leftarrow	$holds(not(ecf(not(F), L)), S).$
$holds(acu(F1, F2, L), S)$	\leftarrow	$holds(and(not(ecu(not(F2), and(not(F1),$ $not(F2))), L), not(ecg(not(F2), L))), S).$

Appendix C

Unsupported SMV constructs

This appendix lists the SMV language constructs that are not supported by the `smv2clp` compiler. Most of these are not mentioned in the SMV manual but nevertheless seems supported by the SMV system.

- Linar Time Temporal Logic (LTL) specifications
- Real time CTL specification (ABU, COMPUTE, etc)
- PRINT declaration

Moreover the compiler is not able to handle undefined out parameters when instantiating modules. The example in Figure C.1 shows this. The reason is that the “outpar” symbol is never declared in the main module. The same problem can be expressed as in Figure C.2

```
MODULE m(a)
VAR
  b : boolean;
DEFINE
  a := !b;

MODULE main
VAR
  testmod : m(outpar);
SPEC
  EF(outpar)&EF(!outpar)
```

Figure C.1: Example of parameters not handled by smv2clp

```
MODULE m
VAR
  b : boolean;
DEFINE
  a := !b;

MODULE main
VAR
  testmod : m;
SPEC
  EF(testmod.a)&EF(!testmod.a)
```

Figure C.2: Equivalent of example in Figure C.1 handled by smv2clp


Bibliography

- [1] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Faster sat and smaller bdds via common function structure. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 443–448. IEEE Press, 2001.
- [2] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 340–351. Springer-Verlag, 1997.
- [3] Henrik Reif Andersen. An introduction to binary decision diagrams, 1997.
- [4] Bob Bentley. Validating the intel pentium 4 microprocessor. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 244–248. ACM Press, 2001.
- [5] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulation of finite state machines. In *ICCD*, pages 224–227, October 1990.
- [6] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple,

- Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In *CAV*, pages 428–432, 1996.
- [7] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [8] C. A. J. van Eijk and J. A. G. Jess. Exploiting functional dependencies in finite state machine verification. In *Proceedings of The European Design and Test Conference (ED & TC)*, pages 9–14, Paris, France, 1996. IEEE Computer Society Press (Los Alamitos, California).
- [9] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Dynamic scheduling and clustering in symbolic image computation.
- [10] S. Cheng. Compiling verilog into automata, 1994.
- [11] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
- [12] Riccardo Forth and Paul Molitor. An efficient heuristic for state encoding minimizing the bdd representations of the transition relations of finite state machines. In *Proceedings of the 2000 conference on Asia South Pacific design automation*, pages 61–66. ACM Press, 2000.
- [13] Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. *Bison*.
- [14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [15] Daniel Geist and Ilan Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 299–310. Springer-Verlag, 1994.

-
- [16] John Harrison. Formal verification at intel. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 45. IEEE Computer Society, 2003.
- [17] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.
- [18] Jawahar Jain, William Adams, and Masahiro Fujita. Sampling schemes for computing obdd variable orderings. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 631–638. ACM Press, 1998.
- [19] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.
- [20] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.
- [21] C. Meinel and C. Stangier. Hierarchical image computation with dynamic conjunction scheduling, 2001.
- [22] Christoph Meinel and Christian Stangier. Speeding up image computation by using rtl information. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 443–454. Springer-Verlag, 2000.
- [23] Christoph Meinel and Christian Stangier. A new partitioning scheme for improvement of image computation. In *Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 97–102. ACM Press, 2001.
- [24] Christoph Meinel and Thorsten Theobald. Local encoding transformations for optimizing obdd-representations of finite state machines. *Form. Methods Syst. Des.*, 18(3):285–301, 2001.

-
- [25] Ulf Nilsson and Johan Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proceedings of the First International Conference on Computational Logic*, pages 384–398. Springer-Verlag, 2000.
- [26] Vern Paxson. *Flex*. Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
- [27] Stefano Quer, Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, Ellen Sentovich, and Robert K. Brayton. Verification of similar fsms by mixing incremental re-encoding, reachability analysis, and combinational checks. *Formal Methods in System Design*, 17(2):107–134, 2000.
- [28] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification, 1995.
- [29] Ellen Sentovich, Horia Toma, and Gérard Berry. Latch optimization in circuits generated from high-level descriptions. In *ICCAD*, pages 428–435, 1996.
- [30] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Heuristic minimization of BDDs using don't cares. In *Design Automation Conference*, pages 225–231, 1994.
- [31] Chao Wang, Gary D. Hachtel, and Fabio Somenzi. The compositional far side of image computation. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 334. IEEE Computer Society, 2003.
- [32] B. Yang, R. Simmons, R. E. Bryant, and D. R. O'Hallaron. Optimizing symbolic model checking for constraint-rich models. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 328–340. Springer-Verlag, Berlin, 1999. LNCS 1633.

 <p>LINKÖPINGS UNIVERSITET</p>	<p>Avdelning, Institution Division, Department</p> <p>Institutionen för datavetenskap 581 83 LINKÖPING</p>	<p>Datum Date 2005-02-24</p>
--	---	---

<p>Språk Language</p> <p>Svenska/Swedish X Engelska/English</p>	<p>Rapporttyp Report category</p> <p>Licentiatavhandling X Examensarbete C-uppsats D-uppsats Övrig rapport =====</p>	<p>ISBN</p> <p>ISRN LITH-IDA-EX--05/018--SE</p> <p>Serietitel och serienummer ISSN Title of series, numbering _____</p>
--	---	---

URL för elektronisk version
<http://www.ep.liu.se/exjobb/ida/2005/dd-d/018/>

Titel	En optimerande kompilator för SMV till CLP(B)
Title	An optimising SMV to CLP(B) compiler
Författare	Mikael Asplund
Author	

Sammanfattning
Abstract

This thesis describes an optimising compiler for translating from SMV to CLP(B). The optimisation is aimed at reducing the number of required variables in order to decrease the size of the resulting BDDs. Also a partitioning of the transition relation is performed. The compiler uses an internal representation of a FSM that is built up from the SMV description. A number of rewrite steps are performed on the problem description such as encoding to a Boolean domain and performing the optimisations.

The variable reduction heuristic is based on finding sub-circuits that are suitable for reduction and a state space search is performed on those groups. An evaluation of the results shows that in some cases the compiler is able to greatly reduce the size of the resulting BDDs.

Nyckelord
Keyword

SMV, CLP, BDD, FSM, CTL, compiler, optimisation, variable reduction, partitioning

Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om *Linköping University Electronic Press* se förlagets hemsida <http://www.ep.liu.se/>

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the *Linköping University Electronic Press* and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>

© Mikael Asplund

