Exploiting Syntactic Structure for Automatic Verification

Research Thesis

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Karen Yorav

Submitted to the senate of The Technion - Israel Institute of Technology

Sivan 5760 Haifa June 2000

The research thesis was done under the supervision of Prof. Orna Grumberg, in the Faculty of Computer Science, the Technion - Israel Institute of Technology.

The generous financial help of the Technion is gratefully acknowledged.

I would like to thank Orna for being the best advisor a student can hope for. Her help in learning the "secrets of the trade" was invaluable. I learned from her how to read (academic material), write (comprehensible articles), and think straight... Our long talks were often the height of my day. For this and everything else, I deeply thank her.

Many thanks are due to Sergio Campos, who worked with me on parts of this thesis. I thoroughly enjoyed his visits to Israel, and I hope that he did too.

Many thanks to Tamar Eilam and Gitit Rockenstein, for being the best room-mates, and to Tami Tamir. Good friends are what makes this more than just a learning experience.

A special thanks to Ran Shimshoni, who knows how to say the right thing at the right time.

To my parents, for always believing in me.

Last but not least, I wish to thank my husband Joe, the one and only. None of this would have been possible without you being there.

Contents

List of Figures

List of Tables

Abstract

As the complexity of software and hardware systems grows it becomes necessary to develop new methods for verification. We are mainly concerned with *temporal logic model checking* algorithms, which receive a model of the system and a specification formula and decide whether the model satisfies the specification. The main restriction on model checking algorithms is the size of the models they are applied to, and solving this problem is the subject of this work.

We suggest ways of using the syntactic description of a system in order to reduce space consumption of verification methods. For software systems the syntactic model is a *control-flow graph*, which includes a node for each command in the program and edges representing the order of execution. For hardware systems we examine the list of signals (flip-flops), and boolean functions that determine their values at each step.

Our results for software verification include a modular model checking algorithm and two space reduction methods. The modular algorithm is based on a partitioning of the program into sub-programs. It converts any CTL model checking algorithm so that it examines each sub-program separately. We have implemented it in a tool called SoftVer and got a substantial reduction both space and time consumption. Our space reduction methods are based on static analysis, which is the analysis of the control-flow graph of a program in order to extract information on its run-time behavior. The first reduction reduces according to control, while the second reduction reduces according to data. We tested the amount of reduction achieved by our methods on several examples. We find that in some cases they can be very productive.

The main method for verifying hardware designs is by simulating its behavior under a given sequence of inputs, called a test sequence, and examining the outputs. We present a *test-sequence generation* algorithm, which given a test sequence to a sub-circuit, creates a test sequence for the whole design. This can be used to translate a set of test sequences for the sub-circuit into a set of tests for the whole design.

Finally, we introduce *dynamic transition relations*, which are used to compute the set of predecessors of a given set of states more efficiently. The idea is that at each computation only the parts of the design relevant for this step are considered. Experimental results show impressive speedups and reduction in space consumptions, even up to two orders of magnitude.

Chapter 1

Introduction

As the complexity of software and hardware systems grows, the need for automatic verification tools becomes increasingly apparent. Traditional testing methods can no longer give a reasonable assurance for the quality of a product, and more rigorous and reliable methods are needed.

The term automatic verification is a general name for all methods of verifying the correctness of a system by means of an algorithm that is guaranteed to terminate and deliver accurate results. This, of course, can only be done on finite-state systems. Hardware systems are inherently finite since all signals are boolean and all memory components contain a specific number of bits. For software systems we must assume that all variables are over finite domains, e.g. boolean, enumerated, or integers of a specified finite range.

In this work we are mainly concerned with *model checking* algorithms. A model checking algorithm receives a model of the system and a specification formula and decides whether the model satisfies the specification. The most common model used to represent finite-state systems is a *Kripke structure* which is a state transition graph with labelings on the states (and not on the edges).

The choice of specification language is important since it determines both the usefulness of the model checking tool, and the complexity of the algorithm. It is important that the specification language will be strong enough to express the type of properties that the designer expects the system to have, but usually the stronger the specification language the more expensive the model checking algorithm becomes. The systems that we examine are mostly *reactive systems*, which means that they work continuously and react to their environment, rather then computing a result and then terminating. It is widely accepted that *temporal logic* specifications [51, 22], are convenient and effective for specifying reactive systems, and these are the specification languages that we refer to in this work. Temporal logic formulas are capable of describing the behavior of a system over time. They can express properties such as "whenever there is a request then some time in the future an acknowledge will be received" or "it is never possible to receive an acknowledge that is not in response to a request". There are several flavors of temporal logics, and each has a model checking

algorithm with a different complexity. The most general and most powerful temporal logic we examine is CTL*. We also use several sub-languages of CTL* because they poses special properties which we use.

The problem that model checking algorithms of all kinds face is that of space complexity. The size of the Kripke structure that represents a system is exponential in the number of (boolean) variables. When the system is composed of several processes running in parallel the cross product of the processes' structures also leads to an exponential blow-up. In order to perform model checking on any system the whole state space must be checked in one way or another, but this cannot be done if the representation of the structure does no fit into the available memory of the computer. This problem is known as the *state explosion problem* and it is the subject of significant parts of the research done in the area of automatic verification.

1.1 Model Checking Techniques

There are many types of model checking algorithms [14, 53, 41, 6, 24] and they differ in the specification language they use, and the way they represent the system. The first model checking algorithms developed were *explicit state* algorithms, which use an explicit representation of the structure of the system. These algorithms generally arrive at their conclusion by traversing the state-space of the system in some way or another. To overcome the state explosion problem, a new approach was developed, called *symbolic* model checking [44, 6]. Here the structure of the system is represented implicitly, by means of *Binary Decision Diagrams* [5] (BDDs). BDDs are data structures that represent boolean functions (functions whose operand is a boolean vector and the result a boolean value), and these can be used to represent sets of states and sets of transitions between states. Symbolic model checking revolutionized the field of automatic verification by making it possible to verify large systems of up to 300 boolean variables. However, there is still a long way to go before model checkers can be used on real designs in full.

There are many works that try to alleviate the state explosion problem. We consider here two prominent approaches. The first is introducing *modularity*. The idea is to be able to verify different parts of the system separately, since each part is much smaller then the combination of all the parts together. There are many ways of making model checking modular. We mention here some of the leading methods, although there are many more.

One way to create modularity is to separate a process from its environment. Using a method called *assume guarantee* [52, 34] a process is checked under certain assumptions on the environment in which it runs, and these assumptions are proved to hold on the actual environment (without the process itself). When several processes interact through limited interfaces, each process can be checked while the others are replaced by a small process that represents only the interface [16]. Modularity can also be achieved by use of fairness constraints. The environment is replaced by a constraint

that states that something good must happen infinitely often. Model checking is then performed, but only fair computations are considered [35].

An important step in mechanizing modular methods in the context of branching time temporal logics is the introduction of *simulation* relation [31] defined by [45]. Simulation relations define a preorder on models that preserves the satisfaction of temporal logic formulas. This means that if a formula is true in a model then it is also true in any model which is smaller in the preorder. To verify a single component of a system, we can find a formula ψ that represents the environment of the chosen component, build a model that represents ψ (a tableau) and verify the chosen component with this model (which is smaller in size) instead of the true environment. In order to prove that the real environment actually satisfies ψ we show that the true environment is smaller in the preorder than the model of ψ.

A different approach to the state explosion problem is that of *reduction*, where the goal is to create as small a structure as possible while maintaining the properties that are to be checked. One useful methodology of reduction is that of *partial order reductions* [50, 56, 30, 55]. These methods rely on the observation that the interleaving model of computation is very wasteful, since in many cases the specification is not influenced by different orderings of the same commands. Partial order reductions define models that exclude some of the possible interleavings between processes while maintaining the validity of the specification. Another type of reduction is *abstraction* [1, 15, 18, 19, 42]. An abstract model of a system is a model in which certain details have been omitted. Care must be taken in the way the abstract model is created so that it preserves the specification, i.e. if the abstract model satisfies the specification we can safely conclude that the concrete model (the original model of the system) also satisfies the specification. The simulation relation mentioned earlier is an important tool for abstraction. It allows us to prove that an abstract model we created is in fact a true abstraction, by showing that it is higher in the preorder than the concrete model. However, if the abstract model does not satisfy the specification it does not necessarily mean that the concrete model does not satisfy it. In this case a different abstraction should be used, one that will include more detail.

1.2 Hardware Verification

The process of hardware design consists of many stages, starting with a very abstract description of modules, and ending with a detailed plan of wires and transistors. Verification is done on all levels, where at each stage different properties are checked. We are concerned with logical behavior (as opposed to physical behavior for example), and this is normally checked at a level of description called *Register Transfer Level* (RTL). In this stage the design consists of memory components (flip-flops), and combinatorical components (logical gates) that determine the value of each flip-flop at every clock cycle.

The main method for verifying logical behavior in hardware design is simulation

(unrelated to the previously mentioned simulation relations), also called dynamic validation. The designs behavior is simulated using random or semi-random inputs, and the results are checked for correct behavior. This process can never cover all possibilities, and usually verifies only parts of the state-space. It is also not fully automatic since the designer must check the results. In spite of this, simulation is the main method of logical verification today. In contrast, model checking can provide an accurate and exhaustive result. When the circuit fails to satisfy the specification, model checking algorithms can also provide a counter-example which is invaluable to the designer in finding the cause for the error. The problem remains that most designs can not be dealt with in full, and only smaller parts can be model checked.

Since hardware designs are made up of many components that work in parallel (whether synchronously or not), modularity is often used to allow verification of larger designs. Most of the theoretical results in improving model checking techniques are targeted at synchronous systems, and are therefore most useful for synchronous circuits. The reason for this is that almost all industrial designs are in fact synchronous circuits.

1.3 Software Verification

The use of automatic verification for software systems is much more limited than for hardware systems, and so is the body of research on the subject. There are several reasons for this.

Most "real-life" programs involve integer variables, pointers, and dynamic allocation of memory. The use of these constructs creates a model of execution that has infinitely many states, and thus model checking is impossible. One way to solve this problem is to limit the programming language so that it allows only finite domains, thus dynamic allocation is prohibited and integers are over a specified finite domain. This solution limits the programmer, and is not applicable to many software systems that are used today. However, some classes of programs can be written in such a limited language, most importantly communication protocols and controllers. These programs are not only finite, but can be very complex and may indeed require verification.

A second way of solving the problem of infinite domains is abstraction. The main drawback of using abstraction is that the abstraction is usually created manually, and if the abstraction is not checked thoroughly errors may be introduced at this stage. More importantly, errors that existed in the original design may be accidently left out of the abstract design and will not be found. There are several works that propose ways of automatically creating abstract models [17, 20]. Although these methods may not produce the optimal abstraction, they are very important since they solve the problem of errors inserted/omitted at the abstraction stage.

1.4 Overview of this work

In this work we look for ways to use the high level description (program text) of a system in order to improve the model checking process, either by modularity or by reduction. The work can be divided into two parts - software and hardware verification.

The first thing to note about software systems is the difference between the *syntactic structure* and the *semantic structure*. When compiling a program (whether the goal is to execute it or to create a Kripke structure from it) the parser builds a *control-flow graph* that represents the syntax of the program. Each program counter location is represented by a node, and the edges express the possible ways in which control can pass from one location to another. From this control-flow graph the Kripke structure representing the program can then be created. The size of the control-flow graph is proportional to the size of the program text, which is very small compared to the size of the Kripke structure of the program. We set out to use this control-flow graph and exploit the special structure that can be found in it.

We first show how partitioning the program into sub-programs can help in creating a modular model checking algorithm. We introduce the notion of *partition graphs* which are a generalization of control-flow graphs in which a node can represent a sub-program rather than just one program counter location. Instead of creating a Kripke structure for the full program we create a structure for each sub-program and keep each one in a separate file (therefore, when not in use these structures are not in the immediate memory). We show that any model-checking algorithm for CTL can be adapted to perform model checking on a partition graph so that each sub-program is handled separately, never keeping the structures of two sub-programs in the immediate memory of the computer at the same time. This method can not only enable handling programs that would otherwise not fit in the memory of the computer, but since model checking is performed on smaller structures it can also be much faster than model checking on the full program.

We have implemented a symbolic model checker called SoftVer that performs our modular model checking algorithm. It uses a simple programming language that restricts variables to finite domains, and allows the user to determine the partitioning of the program into sub-programs. It also allows the use of other symbolic model-checking techniques such as variable reordering and local reachability (both are explained in detail in the appropriate chapter). We applied the tool on three examples, each with different partitionings and compared the space and time requirements needed for model checking with the space and time used when the program is unpartitioned.

The results show that in some cases a substantial reduction is achieved in both space and time consumption. In one example the memory consumption using partitioning was only 13% of the memory consumption without using partitioning. Although our original goal was to reduce memory consumption, it turns out that modular model checking can also reduce running times, and in some cases it even reduces the running time more than the memory consumption. This is because the algorithm uses much smaller models, and for most of the operations performed running time is polynomial

in the size of the model. However, since the implementation is BDD-based we also get cases in which the partitioned version actually takes more space than the unpartitioned. This happens because of the use of symbolic model checking. If we were to use an explicit-state algorithm the space requirements could not grow as a result of partitioning. We review and discuss the results in the appropriate chapter.

We then move to consider reduction methods for programs. We show two types of reductions, both based on static analysis [48]. Static analysis is a general term for methods that analyze the control-flow graph of a program to reveal information on the run-time behavior of the program, without actually running it. In our case, performing static analysis means that we examine the control-flow graph of a program (the syntax) to extract information on the Kripke structure of the program, without actually creating this Kripke structure. The idea behind both of our reductions is to use static analysis in order to create a smaller structure for the program, which we call the *reduced* Kripke structure of the program. The reduced Kripke structure is built so that it is equivalent to the original structure of the program, i.e. a given specification is true in the original Kripke structure iff it is true in the reduced one. The specifications considered are formulas of the logic CTL*, and the logic CTL*-X which is similar to CTL*, but without the next-state operator.

Our algorithms for static analysis are based on syntactic manipulation of expressions, and therefore we allow variables with both finite and infinite domains. When the domains are finite, our methods can be used for automatic verification using an explicit representation of the transition system as well as for verification using a BDD representation. In either case, the verification algorithm itself is not changed, it just receives a smaller model to work on.

We create the reduced Kripke structure for a program directly out of the control-flow graph, and never build the full Kripke structure. We are therefore able to verify systems that would otherwise be too big to handle. The advantage of this approach is even more significant when the system is composed of several processes. In such a case, each process is reduced separately and only then they are composed to one Kripke structure. This solution thus serves to reduce the exponential blow-up that occurs when taking the cross product of the structures of the individual processes.

Another important advantage of using static analysis is that in order to implement our reductions, changes are made only to the compiler (which is relatively simple) and there is no need to change the verification tool or the verification algorithm. This enables integration with existing tools at a very low cost. It also means that the overhead of using our reductions is during the (very short) compilation stage and not in the verification process.

When developing a static analysis method for reducing program models we came up with two orthogonal approaches. We can examine the control of the program, which is the program counter, or we can examine the values that variables can have (the data). Reducing according to control means creating a model that performs fewer steps in order to achieve the same goal, whereas reducing according to data means creating a model that uses a smaller part of the variable domains.

We present and compare two methods that use static analysis to create reduced models for programs. The first method, called *path reduction*, reduces according to control, and the second, called *dead-variable reduction*, reduces according to data. Both methods automatically create a reduced Kripke structure directly out of the syntax of the program (the control-flow graph), thus avoiding the need to create the full Kripke structure. The reductions are independent of each other and can be used together on the same program. In such a case we perform both analyses on the program, and then create a single Kripke structure that is reduced according to both methods (there is no need to create intermediate models according to one method or the other).

We used Murphi [46] to test the amount of reduction achieved by our methods. Murphi is a tool that performs a DFS or BFS traversal of the reachable state space of a program. We chose several example programs and translated them into Murphi. We then constructed Murphi descriptions of the reduced systems created by our methods. We used Murphi's DFS search to compare the sizes of the original and reduced Kripke structures, and the time it takes to traverse them. The experiments with path reduction gave reduced models that were between 8% to 70% of the original model. This shows that for some programs path reduction can be very significant. The results for dead-variable reduction, however, show a much smaller reduction - the reduced models were between 65% to 100% of the original model. It is possible that these results are influenced by our choice of examples, and there may be examples where dead-variable reduction will be more productive.

We now turn to consider the structure of hardware designs. We start by presenting a *test-sequence generation* algorithm. It is often the case that a critical sub-circuit of the design must be checked thoroughly, and a set of test sequences is created for this purpose. Each test sequence is a series of inputs for the sub-circuit, and together they simulate all or most of the sub-circuit's important features. However, these test sequences cannot be used on the full circuit, since the inputs to the sub-circuit may be internal signals of the full circuit, and may not be accessible from the exterior of the design. For each test sequence of the small sub-circuit our algorithm creates a test sequence for the full design, which reproduces the test sequence of the small sub-circuit. This allows the creation of a set of test-sequences for the full design that can achieve good coverage of the sub-circuit.

To create an efficient algorithm one must again examine the structure of the system. We notice that in most designs a single flip-flop is influenced by only a small number of other flip-flops. The naive way of creating a test sequence would be a DFS traversal of the state-space from an initial state, searching for a path that reproduces the required test sequence. This solution is inefficient because every step the whole design would be involved in calculating the next state. Also, this search will traverse the full state-space in order to declare that there is no solution. We find that by starting with the (small number of) signals which are actually involved in the input test-sequence we can create a search that moves backwards, and involves only the parts of the circuit that are actually important. This observation is the basis for our algorithm.

We then move on to consider the operator $Pred$, which is widely used in model checking algorithms, and also in our test-generation algorithm. This operator receives a set of states (of the Kripke structure representing the design) and produces the set of predecessors of these states. To do this, it performs calculations on the transition relation of the Kripke structure. We show that this operator can be performed in a more efficient way by making it use *dynamic transition relations*. Instead of using the full transition relation, each time the operator is invoked a partial transition relation is used. The partial transition relation includes only parts of the design which are relevant for this step, and ignores other parts of the design. We show that the result is exactly the same set of states as the original operator, so the dynamic version can be used to improve model checking and test-generation.

Following the development of the dynamic $Pred$ operator, we developed a dynamic version of our test-generation algorithm. This version uses dynamic transition relations in the calculation of the $Pred$ operator, as well as for other operations the algorithm performs.

We implemented both the test-generation algorithm and the dynamic $Pred$ operator, and ran examples. Our experimentation shows that the dynamic operator can lead to significant reductions in time consumption of symbolic model checking, especially in cases where it is performed a small number of times in succession. We also found that it gives good results in our test-generation algorithm.

The work is organized as follows. Chapter 2 gives some preliminary definitions used throughout the work. Chapter 3 presents our results for software verification. It starts with a discussion of the structure of programs (Section 3.1) and then presents the modular model checking algorithm (Section 3.2) and the static analysis reductions (Section 3.3). Chapter 4 presents our results for hardware verification. Similarly to the software chapter it starts with a discussion of the structure of hardware designs (Section 4.1) and then moves on to present the test-generation algorithm, dynamic transition relations, and the dynamic version of the test-generation algorithm. Chapter 5 presents the SoftVer model checker. Finally, Chapter 6 concludes with a summary of the work, a discussion of works related to ours, and some directions for future research.

Chapter 2

Model Checking for Temporal Logics

2.1 Models of Systems

We use Kripke structures to model the behavior of a finite-state system. Basically, these are state machines with labelings on the states (and no labeling on the edges).

Definition 2.1: A *Kripke Structure* is a tuple $M = \langle S, R, I \rangle$ s.t. S is a set of states, $R \subseteq S \times S$ is a *transition relation* and $I \subseteq S$ is a set of *initial states*. A *computation path* (or simply a path) in M from a state s_0 is a sequence $\pi = s_0, s_1, \ldots$ s.t. $\forall i[s_i \in S$ and $(s_i, s_{i+1}) \in R]$. A *maximal* path in M is a path which is either infinite, or ends in a state with no outgoing transitions. Let π be a maximal path in M. We write $|\pi| = n$ if $\pi = s_0, s_1, \ldots, s_{n-1}$ and $|\pi| = \infty$ if π is infinite.

When we use a Kripke structure to represent a program, every state is a pair (l, σ) where l is a program location (a value for the program counter) and σ is a valuation to the program variables. When we represent a hardware system each state is simply a valuation σ to the set of signals in the design.

Kripke structures usually come with a set AP of *atomic propositions* and a labeling function $L : S \to 2^{AP}$ that associates each state in the structure with the set the of atomic propositions that hold in that state. This is used as a basis for specifications, because it gives different attributes to different states in the structure. In this work we use expressions over system variables as atomic propositions, and hence we do not need a specific labeling function. For each state (which includes values for system variables) we know whether the state satisfies a given expression or not.

Definition 2.2: For a Kripke structure $M = \langle S, R, I \rangle$ we define the set of *ending states* to be: $end(M) = \{s \in S \mid \neg\exists s'.(s, s') \in R\}$. We also use $init(M)$ to refer to the set I of initial states.

Most of the time we use boolean formulas to represent sets of states and sets of transitions. For example, if $\overline{V}$ is the vector of variables of a system then the set of states in the Kripke structure for this system is represented by a boolean formula

$S(\overline{V})$. For every valuation $\overline{v}$ to the variables in $\overline{V}$, $S(\overline{v}) = 1$ iff $\overline{v}$ represents a state of the system. To represent the transition relation we introduce a new set of variables $\overline{V'}$ which is a duplicate of $\overline{V}$, only primed. We use $\overline{V}$ to represent the state from which a transition exits and $\overline{V'}$ to represent the state into which it enters, so the transition relation is represented by a function $R(\overline{V}, \overline{V'})$. The set of ending states can then be represented as: $end(\overline{V}) = S(\overline{V}) \wedge \neg\exists\overline{V'}.R(\overline{V}, \overline{V'})$.

2.2 Specifications

We consider specifications in one of several specification languages globally referred to as *propositional temporal logics*. The feature that makes temporal logics appealing for automatic verification is that they can describe intricate behavior over time (as opposed to input-output specifications, for example, that only compare the starting and ending points of programs). This makes them most suitable for specifying reactive systems such as operating systems, communication protocols (whether implemented in software or hardware) and sequential hardware designs.

The most powerful specification language we consider is CTL* [23]. All other languages that we refer to are subsets of this language.

As mentioned before, we assume a set of *atomic propositions* AP, which serve as the base set for defining CTL* formulas. We assume that for every atomic proposition $p \in AP$ and every state s, it is known whether $s \models p$ or not. Actually, atomic propositions represent attributes of states. For example, p might represent the attribute $x > 0$. Since each state is or includes a value for all system variables, the set of states that satisfy p is the set of states in which the variable x has a value larger than 0.

Definition 2.3 gives the syntax of CTL* formulas, and definition 2.4 gives the semantics.

Definition 2.3: Given a set of atomic propositions AP, the set of *state formulas* includes:

- atomic propositions $p \in AP$, $true$ and $false$,
- $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, and $\neg\varphi_1$ for state formulas φ_1 and φ_2,
- $\mathbf{A}\psi$ and $\mathbf{E}\psi$ for a path formula ψ.

The set of *path formulas* includes:

- any state formula φ,
- $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, and $\neg\psi_1$ for path formulas ψ_1 and ψ_2,
- $\mathbf{X}\psi_1$, $\psi_1\mathbf{U}\psi_2$, $\mathbf{G}\psi_1$, and $\mathbf{F}\psi_1$ for path formulas ψ_1 and ψ_2.

The language CTL* is the set of state formulas.

Definition 2.4: Given a state s we define:

- $s \models true$ and $s \not\models false$ [1]

- $s \models \varphi_1 \vee \varphi_2$ iff ($s \models \varphi_1$ or $s \models \varphi_2$)
  $s \models \varphi_1 \wedge \varphi_2$ iff ($s \models \varphi_1$ and $s \models \varphi_2$)
  $s \models \neg\varphi_1$ iff $s \not\models \varphi_1$

- $s \models \mathbf{A}\psi$ iff for every path π that starts from s, $\pi \models \psi$
  $s \models \mathbf{E}\psi$ iff there exists a path π that starts from s and $\pi \models \psi$

Given a path $\pi = s_0, s_1, \ldots$, define $\pi^i = s_i, s_{i+1}, \ldots$ (the suffix of π starting at s_i). We define:

- for a state formula φ, $\pi \models \varphi$ iff $s_0 \models \varphi$,

- $\pi \models \psi_1 \vee \psi_2$ iff $(\pi \models \psi_1) \vee (\pi \models \psi_2)$
  $\pi \models \psi_1 \wedge \psi_2$ iff $(\pi \models \psi_1) \wedge (\pi \models \psi_2)$
  $\pi \models \neg\psi_1$ iff $\pi \not\models \psi_1$

- $\pi \models \mathbf{X}\psi$ iff $\pi^1 \models \psi$

- $\pi \models \psi_1 \mathbf{U} \psi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \psi_2$ and for every $0 \leq j < i$, $\pi^j \models \psi_1$

- $\pi \models \mathbf{F}\psi$ iff $\pi \models true\mathbf{U}\psi$

- $\pi \models \mathbf{G}\psi$ iff $\pi \models \neg\mathbf{F}\neg\psi$

We also use several sub-languages of CTL*:

Definition 2.5:

CTL*-X is the language we get from CTL* by excluding the **X** operator.

CTL is a language in which temporal operators (**X**,**U**,**G** and **F**) can only be used in combinations made out of a path quantifier applied on a temporal operator. Thus, besides the boolean operators $\vee$, $\wedge$ and $\neg$, the only operators allowed are **AX**, **EX**, **AU**, **EU**, **AG**, **EG**, **AF** and **EF**.

CTL-X is the language we get from CTL by excluding the **X** operator.

Definition 2.6: The *closure* of a formula ψ, denoted by $cl(\psi)$, is the set of all state sub-formulas of ψ (including itself). Note that this includes only sub-formulas which are state formulas.

[1] Recall that we assume we know whether or not $s \models p$ for $p \in AP$ so there is no need to define it.

2.3 Model Checking

We say that a Kripke structure is a model of (satisfies) a formula f if every state $s \in init(M)$ satisfies the formula. A *Model Checking* algorithm is an (automatic) algorithm that decides whether a given structure M is a model of a given formula f.

There are two types of model-checking algorithms. *Explicit-state* algorithms are those that use an explicit representation of the Kripke structure being examined. Usually, these algorithms use a *next-state function*, which is a function that given a state returns the set of successors of this state. Temporal logic model-checking can be performed using a DFS or BFS style traversal of the reachable state-space of the structure, using the next-state function. For such algorithms it is usually not necessary to build in advance the set of states or the transition relation of the structure being examined. However, if the model-checking algorithm needs to scan most or all of the state space in order to give an answer then it becomes necessary to hold the full state space of the structure in the immediate memory, in an explicit representation.

The second type of algorithms is *symbolic* model checking. In symbolic model checking the transition relation of the structure is represented using BDDs [5]. A BDD is a directed acyclic binary-decision graph in which each internal node is labeled with a BDD variable. A BDD is an efficient representation for boolean functions. Each state of the Kripke structure is encoded using BDD variables. A set of states is represented by the boolean function that gives true iff the input vector is the encoding of a state in the set. In a similar manner, transition relations are represented as sets of pairs of states.

The importance of BDDs is that in many cases (although not all) they give a polynomial size representation of the transition relation. Also, they allow the execution of operations on sets of states, instead of traversing the structure one state at a time. For these reasons BDDs and symbolic model-checking have proved to be extremely useful verification methods.

It is important to notice that CTL* specifications are often defined over Kripke structures with *total* transition relations, i.e. having no ending states. Most model-checking algorithms for CTL* (and its sub-languages) assume that the input structure has no ending states.

Chapter 3

Exploiting Structure in Software Verification

3.1 The Structure of Programs

3.1.1 Non-deterministic While Programs

We define a language that will encompass all the essential programming constructs so that it has similar power to "real" programming languages, except that it does not allow dynamic memory allocation. A program in this language is a parallel composition of sequential processes. When we want to discuss sequential programs specifically, we will refer to the set of programs that are built out of a single process that contains no communication commands.

Definition 3.1: A sequential process is defined by:

SeqProc $\rightarrow$ Proc ; terminate

Where Proc is defined by:

Proc $\rightarrow$

simple commands:

skip |
$x := expr$ |
$x := \{expr_1, \ldots, expr_n\}$ |
$a[expr_0] := expr_1$ |
$a[expr_0] := \{expr_1, \ldots, expr_n\}$ |

communication commands:

send($proc\_name$, $expr$) |
receive($proc\_name$, x) |

compound commands:

$Proc_1$; $Proc_2$ |

if B then $Proc_1$ else $Proc_2$ fi |
while B do $Proc_1$ od

where x is a program variable, $expr_i$ are expressions over program variables (including arrays), and B is a boolean condition.

The statement $x := \{expr_1, \ldots, expr_n\}$ is a non-deterministic assignment, after which x will contain the value of one of the expressions $expr_1, \ldots, expr_n$. This construct is added to allow the simulation of an input command. Each statement must have a label, identifying the program counter location associated with that statement.

As can be seen from the definition, at the end of each sequential process there is an extra command called "terminate". This command represents the point of termination of the process, and its program counter location is called the *end location*.

For the purpose of Model-Checking we require that the program will have a finite state space, in which case variable types must be finite: boolean, bounded integer, bounded arrays, enumerated types, etc. When the program is to be verified by other means, such as theorem proving, it is allowed to be infinite, and then we also allow types such as (unbounded) integers, and queues.

Definition 3.2: A non-deterministic while program is a parallel composition $P = [P_1 || \ldots || P_n]$ of sequential processes. Each process has its own set of local variables, and is not allowed to examine variables belonging to other processes or update them.

We use several terms to refer to sub-sets of the language of non-deterministic while programs. As mentioned before, a sequential process (or *full* sequential process) is any text which can be derived from SeqProc in definition 3.1. The term *full sequential program* is used to denote a sequential process that does not include any communication commands. We use the terms *sub-process* and *sub-program* to refer to any text which can be derived from Proc, and in the case of sub-programs does not include any communication commands. If not stated otherwise, when we refer to "a sequential program" we mean either a full-program or a sub-program.

3.1.2 Control-Flow Graphs

A control-flow graph is used for capturing the syntactic structure of a program. This graph is used as part of any translation process from program text to a semantic model (in our case - Kripke structures). The size of the graph is proportional to the number of lines of code, which is very small compared to the Kripke structure that represents the semantics of the program. Control-flow graphs are used by compilers as a data-structure for representing the syntax of programs. The majority of optimizations performed by compilers are done by examining this graph and extracting information about the execution of the program, without actually executing it.

Definition 3.3: Given a sequential process P, its control-flow graph is a graph $CF_P = \langle N, E \rangle$ where N is the set of nodes and E is the set of edges. Each node is labeled with a program counter location, and represents the command at that location: either

a simple command, a communication command, or a boolean condition (for "if"s and "while"s). The edges capture the flow of control in the program:

- A node representing a simple command, or a communication command, has a single out-going edge pointing to the node of the next program location (the next statement to be executed).
- A node representing a non-deterministic assignment $x := \{exp_1, \ldots, exp_n\}$ or $a[expr_0] := \{expr_1, \ldots, expr_n\}$ has n outgoing edges labeled with the expressions exp_1 through exp_n, all pointing to the node of the next program location.
- A node representing an "if" statement is also labeled with the boolean condition of the statement and has two out-going edges labeled "true" and "false", pointing to the program locations of the "then" and "else" statements respectively.
- A node representing a "while" command is also labeled with the boolean condition of the statement and has two out-going edges labeled "true" and "false", pointing to the program locations of the body of the while and the next statement after the while respectively.
- The control-flow graph node representing the end location (the "terminate" command) is the successor of the last statement of the program, and is the only node to have a self-loop.

Figure 3.1 gives an example of a sequential program and its control-flow graph.

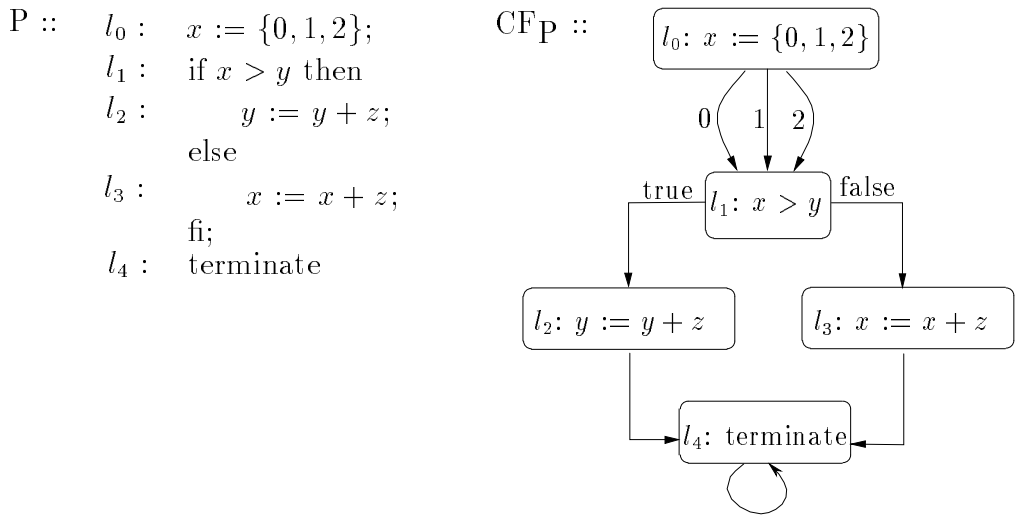


Figure 3.1: A sequential process and its control-flow graph

The above definition differs slightly from the regular definition of control-flow graphs in the definition of the edges leaving a non-deterministic assignment node.

Usually there is a single outgoing edge, similar to the case of regular assignments, whereas in our definition there are several edges - one for each expression possibly assigned.

To create a control-flow graph of a program which is composed of several processes, we create a separate control-flow graph for each process. Since this data-structure is meant to represent syntactic structure and not actual execution behavior, we do not account for communications in any way. The control-flow graph of each process may include one or more nodes labeled with communication commands. We do not (yet) worry about who may communicate with whom.

The control-flow graph of a sub-process (or sub-program) is created so that the set of graph nodes is the set of all program locations in the sub-process plus the location the program counter reaches when the sub-process is done. The extra node for the location in which the sub-process ends will have no out-going edges, even if it is the node of a "terminate" command. For example, for the program "P': l_1: if $x > y$ then l_2: $y := y + z$ else l_3: $x := x + z$ fi", which is a sub-process of the process P from figure 3.1, the partition graph $CF_{P'}$ will be as depicted in figure 3.2. The reason we add the node l_4 to this graph is that we will later define the semantics of control-flow graphs so that an edge exiting from a node represents the execution of the statement in that node. We include the edges outgoing from locations l_2 and l_3 so that the execution of these statements is included in the semantics associated with this control-flow graph. We do not include the self-loop of the terminate command because it is not part of this sub-process.

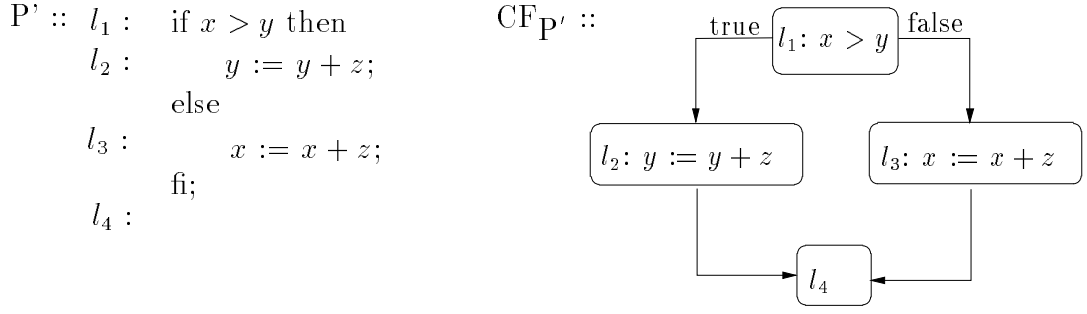


Figure 3.2: A sub-program and its control-flow graph

3.1.3 Semantics of Programs

As mentioned before, we define the semantics of programs by associating each program P with a Kripke structure $struct(P)$ that represents its behavior. We start by defining the Kripke structure of a sequential program, a single process (or sub-process) that contains no communication commands.

The structure that represents P is created from the control-flow graph CF_P of P, and may also be referred to as $struct(CF_P)$.

Throughout this chapter we use $\overline{x} = x_1, \ldots, x_n$ as the vector of program variables (not including the program counter) [1]. In some cases, where we want to describe a transition relation, we use $\overline{x}$ for the current state and $\overline{x}' = x'_1, \ldots, x'_n$ for the next state. We use pc and pc' to describe the current program counter location and the next program counter location respectively. We also use Σ as the set of all possible assignments to $\overline{x}$, and Loc as the set of all program locations (all the possible values that pc can have during the execution of P).

The set of states of $struct(P)$ is $S = Loc \times \Sigma$, i.e. the set of all pairs (l, σ) such that l is a program counter location and σ is an assignment to the program variables. The set of initial sets is $I = l_{start} \times \Sigma$, where l_{start} is the initial program counter location in the program. To define the transition relation R of $struct(P)$, we define a transition relation (a set of transitions) R_e for every edge e in the control-flow graph of P. The relation R_e represents the execution of the statement of the node from which e exits. The full transition relation of $struct(P)$ is then: $R = \bigcup_{e \in CF_P} R_e$. The transition relation R_e for the edge $e = n \rightarrow n'$ (n and n' are nodes in CF_P) is defined according to the command at n. We use $\sigma[x \leftarrow e]$ for the assignment that results from taking σ and assigning $\sigma(e)$ into x (the result of the assignment $x := e$).

Definition 3.4: Given an edge $e = n \longrightarrow n'$, where n is labeled with the location l and n' is labeled with the location l', we define the transition relation R_e according to the command at l, and the label on e (if there is one).

- skip:
  $R_e \stackrel{\triangle}{=} \{((l, \sigma), (l', \sigma)) \mid \sigma \in \Sigma\}$

- $x_i := expr$:
  $R_e \stackrel{\triangle}{=} \{((l, \sigma), (l', \sigma[x \leftarrow expr])) \mid \sigma \in \Sigma\}$

- $x_i := \{expr_1, \ldots, expr_n\}$, and e is labeled with $expr_k$:
  $R_e \stackrel{\triangle}{=} \{((l, \sigma), (l', \sigma[x \leftarrow expr_k])) \mid \sigma \in \Sigma\}$

- $a[expr_0] := expr_1$:
  $R_e \stackrel{\triangle}{=} \{((l, \sigma), (l', \sigma[a[\sigma(expr_0)] \leftarrow expr_1])) \mid \sigma \in \Sigma\}$

- $a[expr_0] := \{expr_1, \ldots, expr_n\}$, and e is labeled with $expr_i$:
  $R_e \stackrel{\triangle}{=} \{((l, \sigma), (l', \sigma[a[expr_0] \leftarrow expr_i])) \mid \sigma \in \Sigma\}$

[1] Each array of length t is represented by t variables, and these variables are included in the vector $\overline{x}$.

- A positive condition B (l is an "if" or "while" command and e is labeled with $true$):
  $R_e \stackrel{\triangle}{=} \{((l,\sigma),(l',\sigma)) \mid \sigma \models B\}$

- A negative condition B (l is an "if" or "while" command and e is labeled with $false$):
  $R_e \stackrel{\triangle}{=} \{((l,\sigma),(l',\sigma)) \mid \sigma \not\models B\}$

- terminate ($n = n'$ and $l = l'$):
  $R_e = \{((l,\sigma),(l,\sigma)) \mid \sigma \in \Sigma\}$

Notice that according to the above definition the transition relation of a full program is total. The transition relation of a sub-program, however, is not necessarily total, and the structure for a sub-program may include ending states.

Figure 3.3 gives an example of a simple process, and the Kripke structure that represents its semantics.

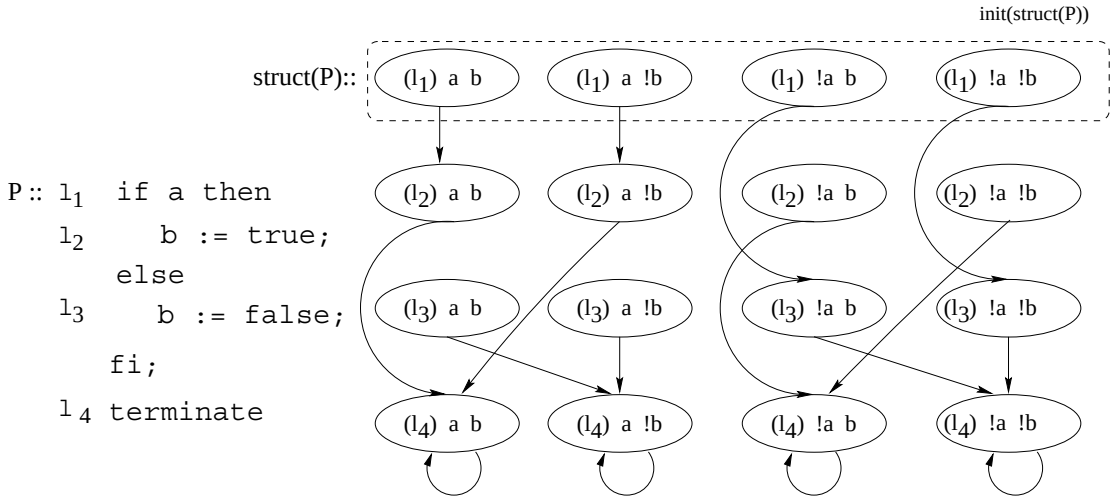


Figure 3.3: A sequential process and its Kripke structure.

When we want to create a Kripke structure for a parallel composition of processes $P = [P_1||\ldots||P_n]$ things are slightly more complicated, since we have to take into account communication commands and parallel execution of processes.

We first create the set of states S which is the cross product of the individual sets of states of the processes. The set of states S_i for a process P_i is defined as was defined above: $S_i = Loc_i \times \Sigma$, where Loc_i is the set of program counter locations

in P_i [2]. We make sure that the sets Loc_i are disjoint. The set of states for P is: $S = \{(s_1, \ldots, s_n) \mid s_1 \in S_1 \wedge \ldots \wedge s_n \in S_n\}$.

To build the transition relation for P, according to an interleaving model of execution, we need to differentiate between two types of transitions: those in which one process executes a single (internal) command and the others do not advance, and those in which a communication occurs. The transition relation R will be built of two separate sets of transitions. We partition each set of locations Loc_i into two parts, Loc_i^{comm} is the set of locations in which P_i executes a communication command and Loc_i^{int} is the set of locations in which P_i executes an internal command (obviously $Loc_i = Loc_i^{comm} \cup Loc_i^{int}$).

A pair of communication commands is called *matching* if they describe a possible communication between two different processes P_i, P_j $(i \neq j)$. This is a syntactic definition since the names of the processes appear in the commands. A matching pair of communication commands includes a "$l_i^1 : send(P_j, expr)\ l_i^2$" command in some process P_i and a "$l_j^1 : receive(P_i, x)\ l_j^2$" command in another process P_j ($l_i^1 \in Loc_i^{comm}$ and $l_j^1 \in Loc_j^{comm}$). We create a set of transitions including all possible situations in which this communication is executed: $\{((s_1, \ldots, s_n), (s'_1, \ldots, s'_n)) \mid (\forall k \neq i, k \neq j . s_k = s'_k)$ and if $s_i = (l_1^i, \sigma)$ and $s_j = (l_1^j, \tau)$ then $(s'_i = (l_2^i, \sigma) \wedge s'_j = (l_2^j, \tau[x \leftarrow \sigma(expr)]\}$. This set describes the passing of the value of $expr$ in P_i to the variable x in P_j. We build the set of transitions R_{comm} which is the union of the above for all possible matching pairs of communication commands.

The transitions in which one process executes an internal command and the others are idle are built according to the definition of the transition relation of a single process. We create a transition relation R_i for each process P_i which includes transitions created from commands in Loc_i^{int}.

We can now define the transition relation R of P to be:

$$R \triangleq R_{comm} \cup \bigcup_{i=1}^{n} \{((s_1, \ldots, s_n), (s'_1, \ldots, s'_n)) \mid (\forall k \neq i . s_k = s'_k) \wedge (s_i, s'_i) \in R_i\}$$

3.2 Modular Model Checking

This section describes a modular algorithm for CTL model checking of sequential non-deterministic while programs. We suggest a way of partitioning a program into components, following the program *text*. A given program may have several different partitions. A partition of the program is represented by a *partition graph*, whose nodes are sub-programs and whose edges represent the flow of control between sub-programs.

Once the program is partitioned, we wish to check each part separately, so that when working one part all the others are kept in secondary memory (files). However, verifying one component in isolation amounts to checking the specification formula on a model in which some of the paths are truncated, i.e. for certain states in the

[2] Note that here we discuss only full sequential processes, and not sub-processes. There is no point in defining the parallel execution of sub-processes.

component we do not know how the computation proceeds (since the continuation is in another component). Such states are called *ending states*. We notice, however, that the truth of a formula at a state inside a component can be determined solely by considering the state transition graph of this component, and the set of formulas which are true at the ending states. Moreover, the truth of a formula at an ending state depends only on the paths leaving it, and not on the paths leading to it. This observation is the basis for our algorithm.

We define a notion of *assumption function* that represents partial knowledge about the truth of formulas at ending states. Based on that, we define a *semantics under assumptions* that determines the truth of temporal formulas based on a given assumption function. Only minor changes are needed in order to adapt any standard CTL model checking algorithm so that it performs model checking under assumptions.

Given a procedure that performs model checking under assumptions, we develop a *modular model checking algorithm* that checks the program in parts. To illustrate how the algorithm works consider the program $P = P_1; P_2$. We notice that every path of P lies either entirely within P_1 or has a prefix in P_1 followed by a suffix in P_2. In order to check a formula ψ on P, we first model check ψ on P_2. The result does not depend on P_1 and therefore the algorithm can be applied to P_2 in isolation. We next want to model check P_1, but now the result does depend on P_2. In particular, ending states of P_1 have their continuations in P_2. However, each ending state of P_1 is an initial state of P_2 for which we have already the model checking result [3]. Using this result as an *assumption* for P_1, we can now model check P_1 in isolation. Handling loops in the program is more complicated but follows a similar intuition.

Before presenting our algorithm we give several definitions. The first is for assumption functions. We notice that a sub-program does not necessarily end with the "terminate" command, and therefore the Kripke structure representing it may have a non-empty set of ending states. Most model-checking algorithms are not designed to handle such structures. Assumption functions are introduced to solve this problem, because they hold information about the ending states. They tell us which formulas each ending state satisfies. This information is not available when examining the Kripke structure of a sub-program in isolation. We will later show how assumption functions are created and used.

Let ψ be a CTL formula, which we wish to check on a given program.

Definition 3.5: An *assumption function* for the Kripke structure $M = \langle S, R, I \rangle$ is a function $As : cl(\psi) \rightarrow (2^{S'} \cup \{\bot\})$ where S' is some subset of the set of states S. We require that $\forall \varphi \in cl(\psi)$, if $As(\varphi) \neq \bot$ then $\forall \varphi' \in cl(\varphi)$, $As(\varphi') \neq \bot$.

When $As(\varphi) = \bot$ it means that we have no knowledge regarding the satisfaction of φ in S'. It is used when we want to represent knowledge relating to other sub-formulas and ignore φ at this stage. If $As(\varphi) \neq \bot$ then $As(\varphi)$ represents the set of all states in S' for which we assume (or know) that φ holds. For every state $s \in S'$ s.t. $s \notin As(\varphi)$ we assume that $\neg\varphi$ holds. The significance of the set S' is that it is the set of states

[3] The result includes for each sub formula φ of ψ the set of states satisfying φ.

which we examine. The assumption function gives no information regarding states outside of S'.

We say that As is an assumption function over some set of states A if A is the set S' about which As gives information.

Satisfaction of a CTL formula φ in a state $s \in S$ under an assumption function As is denoted $M, s \models_{As} \varphi$ [4]. Satisfaction of formulas in M under an assumption As is defined only when the assumption As is defined over a set that includes $end(M)$. We define it so that it holds if either $M, s \models \varphi$ directly (by infinite paths only), or through the assumption function. For example, $M, s \models_{As} \mathbf{E}(f\mathbf{U}g)$ if there exists an infinite path from s satisfying f in all states until a state satisfying g is reached, but it is also true if there is a finite path from s in which the last state, say s', satisfies $s' \in As(\mathbf{E}(f\mathbf{U}g))$, and all states until s' satisfy f. Formally:

Definition 3.6: Let $M = \langle S, R, I \rangle$ be a Kripke structure and As an assumption function over a set S' such that $end(M) \subseteq S'$. For every $\varphi \in cl(\psi)$:
If $As(\varphi) = \perp$ then $s \models_{As} \varphi$ is not defined.
Otherwise, we differentiate between ending states and other states. If $s \in end(M)$ then $s \models_{As} \varphi$ iff $s \in As(\varphi)$. If $s \notin end(M)$ then $s \models_{As} \varphi$ is defined as follows:

- For every $p \in AP$, $s \models_{As} p$ iff $s \models p$.
- $s \models_{As} \varphi_1 \vee \varphi_2$ iff ($s \models_{As} \varphi_1$ or $s \models_{As} \varphi_2$).
- $s \models_{As} \neg\varphi_1$ iff $s \not\models_{As} \varphi_1$ [5].
- $s \models_{As} \mathbf{AX}\varphi_1$ iff $\forall s'.(s, s') \in R \Rightarrow s' \models_{As} \varphi_1$.
- $s \models_{As} \mathbf{EX}\varphi_1$ iff $\exists s'.(s, s') \in R \wedge s' \models_{As} \varphi_1$.
- $s \models_{As} \mathbf{A}(\varphi_1\mathbf{U}\varphi_2)$ iff for all maximal paths $\pi = s_0, s_1, \ldots$ from s there is a number $i < |\pi|$ such that:
  either $(s_i \models_{As} \varphi_2)$ or $(s_i \in end(M) \wedge s_i \models_{As} \mathbf{A}(\varphi_1\mathbf{U}\varphi_2))$, and $\forall 0 \leq j < i[s_j \models_{As} \varphi_1]$.
- $s \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$ iff there exist a maximal path $\pi = s_0, s_1, \ldots$ from s and a number $i < |\pi|$ such that:
  either $(s_i \models_{As} \varphi_2)$ or $(s_i \in end(M) \wedge s_i \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2))$, and $\forall 0 \leq j < i[s_j \models_{As} \varphi_1]$.

Note that if the transition relation of M is total then the above definition is equivalent to the traditional definition of CTL semantics, because the assumption function is consulted only on states from which there are no outgoing transitions. Since the assumption function is consulted only for states in $end(M)$ it seems pointless at this

[4] When no confusion may occur we omit M.

[5] Since $As(\neg\varphi) \neq \perp$, which is why $s \models_{As} \neg\varphi$ is defined, we conclude that $As(\varphi) \neq \perp$. This means that the set of states that satisfy φ is defined, and $s \not\models_{As} \varphi$ iff s is not in this set.

point to define an assumption function over a set S' that includes more than just the ending states. However, later on we use assumption functions as (intermediate) results of model checking, and than we this possibility is used.

We write $M \models_{As} \psi$ iff $\forall s \in I, [M, s \models_{As} \psi]$. We can now define model-checking under assumptions.

Definition 3.7: Given a structure $M = \langle S, R, I \rangle$, and an assumption function As over a set that includes $end(M)$, we define a function $MC[M, As] : cl(\psi) \rightarrow (2^S \cup \{\perp\})$ so that for any $\varphi \in cl(\psi)$, if $As(\varphi) = \perp$ then $MC[M, As](\varphi) = \perp$. Otherwise, $MC[M, As](\varphi) = \{s \in S \mid M, s \models_{As} \varphi\}$.

Notice that $MC[M, As]$ is an assumption function over S; it is a function that given a formula φ produces the set of all states in M that satisfy φ under the assumption As. Given M and As, this function can be created using any known model checking algorithm for CTL [14, 53, 6], after adapting it to the semantics under assumptions. For example, $MC[M, As]$ can be calculated using BDDs since all the operations in its definition are efficient BDD operations.

3.2.1 Partition Graphs

A *Partition Graph* of a program P is a finite directed graph representing a decomposition of P into several sub-programs while maintaining the original flow of control. It is, in fact, a generalization of the control-flow graph, in the sense that a control-flow graph can be viewed as a partition graph, although there are many other possible partition graphs for the same program. The nodes of the graph are labeled with sub-programs of P or boolean conditions. A node labeled with a sub-program represents the execution of this program, and has one outgoing edge. A node labeled with a boolean condition represents the evaluation of this condition, and has two outgoing edges - one labeled with "true" and the other with "false". We also add dummy nodes with no labelings which are used to maintain structure, but have no semantic meaning (they do not represent execution of commands). There are three types of edges: null-edges, true-edges, and false-edges, denoted $n_1 \rightarrow n_2$, $n_1 \xrightarrow{true} n_2$ and $n_1 \xrightarrow{false} n_2$ respectively. True-edges and false-edges, also called *step edges*, represent the evaluation of a boolean condition, and will always exit a node labeled with a boolean condition B. Null edges are used only to maintain flow of control, and represent no execution step. A null-edge always exits a node labeled with a sub-program (or a dummy node) and represents sequential composition.

There are two differences between partition graphs and control-flow graphs. The first is that in control-flow graphs a node may include only a single command, while in partition graphs it may include a complicated sub-program, with "if"s and "while"s in it. The second difference is that even when the last (and perhaps the only) command in a node is a non-deterministic assignment there will only be one out-going edge from that node. This is just a technical difference, the semantics of the non-deterministic assignment does not change. We start by defining the set of all possible partition

graphs for a program, and then define the semantics of partition graphs by associating a Kripke structure with every partition graph.

Every partition graph has two designated nodes: the *entry node*, from which execution starts, and the *exit node*, at which it stops. The set $pg(P)$ contains all possible partition graphs of P, representing different ways of partitioning P into sub-programs. It is defined recursively, where at each step one may decide to break a given program according to its primary structure, or to leave it as a single node. Figure 3.4 shows the three different ways in which a program may be decomposed, according to the three structures by which programs are created. We use in_1 (in_2) for the entry node of G_1 (G_2) and out_1 (out_2) for the exit node. Every time a node n is partitioned into a graph G, all the edges that entered n will enter the initial node of G, and all the edges that exited n will exit from the exit node of G.

1. If $P = P_1; P_2$ we may decompose it into two parts, by creating (recursively) partition graphs $G_1 \in pg(P_1)$ and $G_2 \in pg(P_2)$, and connecting them with a null edge from out_1 to in_2. The entry node of the resulting graph would be in_1, and the exit node would be out_2 (Figure 3.4 A).

2. If $P =$ "if B then P_1 else P_2 fi", we again create the two graphs $G_1 \in pg(P_1)$ and $G_2 \in pg(P_2)$ but also create two new nodes, one labeled with B and the other a dummy node with no labeling The entry node is the B node, and the exit node is the dummy node. The edges connecting the different components are according to the semantics of the "if" command, i.e. a true-edge from the B to G_1, a false-edge from the B node to G_2 and null-edges from G_1 and G_2 to the dummy node. The edges entering G_1 and G_2 are pointed at in_1 and in_2 and the edges exiting G_1 and G_2 are from out_1 and out_2. (Figure 3.4 B).

3. If $P =$ "while B do P_1 od", we create a partition graph $G_1 \in pg(P_1)$ and again a node for B, which is the entry node, and a dummy node as the exit node. The edges represent the semantics of the "while" loop, i.e. a true-edge from the B node to G_1, a false-edge from the B node to the dummy node, and a null edge from G_1 to the B node. (Figure 3.4 C).

A partition graph is used to represent a decomposition of $struct(P)$ into several structures of sub-programs. We associate a Kripke structure with each element of the graph.

- For a node n labeled with a sub-program P' of P, the structure associated with n is the structure of the sub-program: $struct(n) = struct(P')$.

- A node n labeled with a boolean expression B represents the execution of an "if" or "while" command that evaluates this condition. Let l be the program location of this statement. The structure associated with n is: $struct(n) = \langle S, R, I \rangle$ such that $R = \emptyset$ and $S = I = \{(l, \sigma) \mid \sigma \in \Sigma\}$.

- The Kripke structure associated with the dummy node is empty because this node does not represent an actual execution step, but rather is added so that each sub-graph will have only one exit point. In the end there will be a null-edge from the dummy node to some real node n. Every edge that enters the dummy node is considered as if it entered n.

- A null edge $n_1 \rightarrow n_2$ does not reflect an execution step in itself, and therefore if $M_1 = struct(n_1)$ and $M_2 = struct(n_2)$ then $end(M_1) = init(M_2)$ (every ending state of M_1 is an initial state of M_2). For this reason, there is no structure associated with null-edges. Step-edges are the edges outgoing from a node labeled by a boolean expression B. Let l be the program location of the "if" or "while" statement that evaluates B, and l' be the program location of the first statement in n_2. Execution from a state in a node labeled B continues through the true-edge or the false-edge, depending on whether the expression evaluates to $true$ or $false$ in that state. These edges represent an actual step in the execution of P, and the structures associated with them capture this step. The structure associated with an edge $e = n_1 \stackrel{true}{\longrightarrow} n_2$ is defined by $struct(e) = \langle S, R, I \rangle$ where $S = \{l, l'\} \times \{\sigma \mid \sigma \models B\}$, $R = \{((l, \sigma), (l', \sigma)) \mid \sigma \models B\}$ and $I = \{(l, \sigma) \mid \sigma \models B\}$. For $e = n_1 \stackrel{false}{\longrightarrow} n_2$ the definition is similar, except that every $\sigma \models B$ becomes $\sigma \not\models B$.

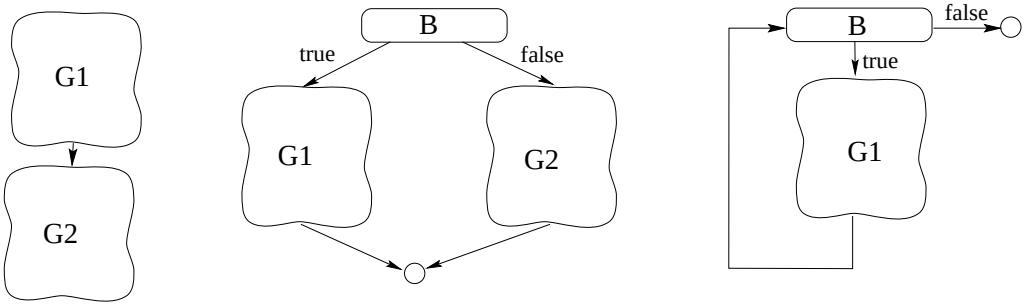
Given a partition graph $G \in pg(P)$, the structure that defines its semantics, denoted $struct(G)$ is constructed by taking the union of the structures associated with all its nodes and edges. The resulting structure is exactly $struct(P)$, the Kripke structure representing the program. This follows immediately from the definition of $struct(P)$ given in Section 3.1.3, and the above definition of the structures associated with nodes and edges. Notice that the connection between structures of sub-programs is through states that appear in more than one node. For example, given a partition graph of $P_1; P_2$ that has two nodes, one for P_1 and one for P_2, the structure for P_1 will include the set of states with the location of the beginning of P_2, because that location is a node in the control-flow graph of the sub-program P_1. The same states will also appear in the structure of P_2 because they are the initial states of this sub-program.

We define $init(G)$ to be the set of initial states in $struct(G)$ and $end(G)$ to be the set of ending states in $struct(G)$. Figure 3.5 gives an example of an actual partition graph, including the Kripke structures associated with the nodes.

3.2.2 Operations on Assumption Functions

Before we present our modular algorithm we define a few operations on assumption functions that are used in the algorithm.

The most basic operation is performing model checking under assumptions on a single node n labeled by a sub-program P'. For this we use MC, which was defined earlier. Given an assumption function As over the ending states of $struct(P')$ we can calculate the assumption function $As' = MC[struct(P'), As]$. The function As'



(A) P = P1;P2 (B) P = if B then P1 else P2 fi (C) P = while B do P1 od

Figure 3.4: Creation of partition graphs

gives us full knowledge about which states of M satisfy which formulas, under the assumption As.

Next we present a function called $CheckStepEdge$, which performs model checking under assumptions over a step edge. The input to this procedure is a step-edge $e = n_1 \xrightarrow{true} n_2$ or $e = n_1 \xrightarrow{false} n_2$ and an assumption function As over a set including the initial states of n_2 (which are the ending states of $struct(e)$) [6]. The output is the assumption function As' over $struct(e)$ calculated by: $As' = MC[struct(e), As]$. Notice that if e is a true-edge (false-edge) then As' is defined only on states that satisfy (do not satisfy) the condition labeling n_1. Actually, when calculating As' for a step-edge there is no need to use a full model-checking algorithm. The structure $struct(e)$ is defined so that each initial state has exactly one successor, and this successor is an ending state. Therefore, we can calculate As' more efficiently according to the following definition:

Definition 3.8: Let $e = n_1 \xrightarrow{true} n_2$ be a true-edge in a partition graph G, and let $As : cl(\psi) \rightarrow (2^{S_1} \cup \{\bot\})$ be an assumption function over S_1 such that $init(struct(n_2)) \subseteq S_1$. Define $As' = CheckStepEdge(e, As)$ s.t. $As' : cl(\psi) \rightarrow (2^{S_T} \cup \{\bot\})$. Let l be the program location of n_1 (an "if" or "while" statement), let l' be the program location of the beginning of n_2, and let $S_T = \{(l, \sigma) \mid \sigma \models B\}$ be the set of states of $struct(n_1)$ over which As' is defined.

If $As(\varphi) = \bot$ then $As'(\varphi) = \bot$. Otherwise, $As'(\varphi)$ is defined as follows [7] :

- For any $p \in AP$, $As'(p) = \{s \in S_T \mid s \models p\}$
- $As'(\neg\varphi) = S_T \setminus As'(\varphi)$
- $As'(\varphi_1 \vee \varphi_2) = As'(\varphi_1) \cup As'(\varphi_2)$
- $As'(\mathbf{AX}\varphi) = As'(\mathbf{EX}\varphi) = \{(l, \sigma) \in S_T \mid (l', \sigma) \in As(\varphi)\}$

[6] The assumption function As must be defined over a set S' that includes the initial states of n_2, but it may include other states as well.

[7] If $As(\varphi) \neq \bot$ then for all sub-formulas φ' of φ we are assured that $As(\varphi') \neq \bot$.

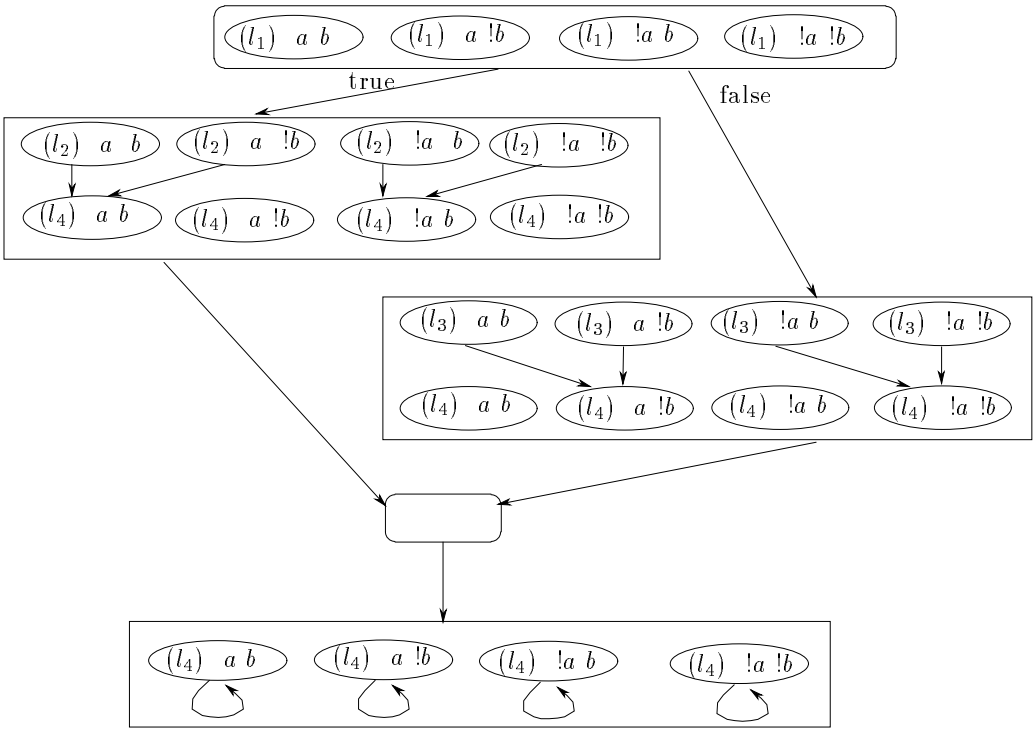


Figure 3.5: An example partition graph

This is a partition graph for the program P from figure 3.3. Instead of writing in each node the sub-program that it represents, we show the structure associated with that node.

- $As'(\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)) = As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)) =$
  $As'(\varphi_2) \cup (As'(\varphi_1) \cap \{(l,\sigma) \in S_T \mid (l',\sigma) \in As(\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2))\})$

For a false-edge $n_1 \xrightarrow{false} n_2$ the definition is the same, except that instead of S_T we use: $S_F = \{(l,\sigma) \mid \sigma \not\models B\}$.

Lemma 3.1: For any step edge e and assumption function As over S_1 such that $end(struct(e)) \subseteq S_1$,
$CheckStepEdge(e, As) = MC[struct(e), As]$.

This statement is obvious from the definitions of model-checking under assumptions and $CheckStepEdge$.

3.2.3 The Modular Algorithm

In this subsection we give an algorithm to check a formula ψ on a partition graph G of a sequential process P. The result is an assumption function over the set of initial states of P that gives, for every sub-formula φ of ψ, the set of all initial states of P satisfying φ. We start with an intuitive description of how the algorithm works. Variable names which are mentioned refer to variables in the algorithm.

The algorithm works on a partition graph G of a program P, and traverses it from the exit node upwards to the entry node. First the structure $struct(v)$ of a leaf node v of G is checked under an "empty" assumption for $cl(\psi)$, an assumption in which all values are $\emptyset$. Since v is a leaf it must end with the "terminate" command and therefore all paths in it are infinite, and the assumption function has no influence on the result. The result of the model checking algorithm is an assumption function As' over $init(struct(v))$ that associates with every sub-formula of ψ the set of all initial states of $struct(v)$ that satisfy that sub-formula. Once we have As' on v we can derive a similar function As over the ending states of any node u, preceding v in G (that is, any node u from which there is an edge into v). Next, we model check u under the assumption As. Proceeding in this way, each node in G can be checked in isolation, based on assumptions derived from its successor nodes. Special care should be taken when dealing with loops in the partition graph.

The algorithm is called CheckGraph. Given a procedure that properly computes $MC[M, As]$, CheckGraph takes a partition graph G and an assumption function As over $end(G)$ and performs model checking under assumptions resulting in an assumption As' over $init(G)$. CheckGraph is able to handle partially defined assumption functions, in which there are some $\perp$ values. For any sub-formula φ s.t. $As(\varphi) = \perp$ we get $As'(\varphi) = \perp$. CheckGraph is defined by induction on the structure of G. The base case handles a single node by using the given procedure MC. To model check a partition graph G of $P = P_1; P_2$, as in figure 3.4 (A), CheckGraph first checks G_2 under As, using a recursive call (see Figure 3.6). As_1 is the result of this call (As_1 is over the set $init(struct(G_2))$). It then uses As_1 as an assumption over the ending states of G_1 and checks G_1 w.r.t As_1 using another recursive call. The second call

returns for all $\varphi \in cl(\psi)$ such that $As(\varphi) \neq \perp$ the set of all initial states of G_1 (which are the initial states of P) that satisfy φ, which is the desired result.

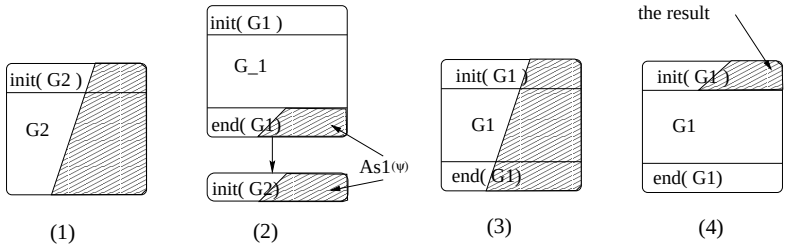


Figure 3.6: The operation of CheckGraph on sequential composition. The gray area is the set of states that satisfy ψ.

Let G be a partition graph of $P =$ "if B then P_1 else P_2", as in figure 3.4 (B). To check G we first check G_1 and G_2, and then compute 'backwards' over the step-edges (using $CheckStepEdge$) to get the result for the initial states of G.

The most complicated part of the algorithm is for the partition graph G of a program $P =$ "while B do P_1 od", as in figure 3.4 (C). We start from the dummy node, with the assumption As over the set of states of the dummy node (the set of states with the program counter location of the next command after the while). Walking backwards on the false-edge we use $CheckStepEdge$ to get an assumption $As_{\neg B}$ over the initial states of G that satisfy $\neg B$. We then use recursive calls over the body of the while to compute the assumption function As'.

In this part it is important that when calculating the set of states that satisfy a formula we have already finished all calculations for all of its sub-formulas. For this reason we order the formulas in $cl(\varphi)$ according to their length and operate on them one at a time. We start with an assumption function As' that has a $\perp$ value for all formulas except the shortest, and use a recursive call to evaluate this formula. The recursive call will disregard all formulas for which $As' = \perp$. When this is done, we move on to the next formula, changing the value of As' from $\perp$ to a set of states. We continue this process for each sub-formula in $cl(\varphi)$.

The value $As'(\varphi_k)$ is computed for each $\varphi_k \in cl(\varphi)$, according to the structure of the formula φ_k. The most complicated part here is the computation for the temporal operators **EU** and **AU**. We now demonstrate the computation of $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$. The algorithm handles the formulas in $cl(\varphi)$ one at a time, so that when reaching $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ it has already dealt with φ_1 and φ_2. This means that the assumption functions $As'(\varphi_1)$ and $As'(\varphi_2)$ are already calculated (over $init(G)$). Since the algorithm is recursive, every time we apply it to the body of the loop (G_1) it calculates values for all formulas which are not $\perp$ in the input assumption function. Specifically, this means that it will calculate correctly the sets of states in $struct(G_1)$ that satisfy φ_1 and φ_2. The goal is to mark all states that satisfy $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ (to create $As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$. Standard model checking algorithms would start by marking all states that satisfy φ_2, and then repeatedly move backwards on transitions and mark every state that has a transition

into a marked state, and satisfies φ_1 itself. We reconstruct this computation over the partition graph of P. For initial states of G that satisfy B we have no assumption regarding $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$, so we mark all those that satisfy B and φ_2 and keep them in $Init_B$. Together with $As_{\neg B}(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$ we have an initial estimate for $As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$ (kept in $As^0(\varphi_k)$). We now want to mark all the predecessors in G of these states. Notice that $init(G) = end(G_1)$, because both are defined as $l \times \Sigma$ where l is the program location in which B is evaluated. This means that the predecessors we are looking for are inside G_1. Hence we continue from $end(G_1)$ backwards inside G_1 until we arrive at $init(G_1)$, and keep the result in Tmp, which is an assumption function over $init(G_1)$. We notice that at this point, only the marks on states of $init(G_1)$ are needed to proceed, the marks on all other states of G_1 are not preserved. If and when we pass through G_1 again, some calculations may have to be repeated. We will later discuss how to solve this problem. Notice also that G_1 itself may consist of more than one node, and the creation of Tmp is done by a recursive call to CheckGraph. From Tmp we can calculate a new estimate for $As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$.

The whole process repeats itself since the body of a "while" loop can be executed more than once. It is essential that the initial states satisfying φ_1 and φ_2 be known before this process can be performed. Therefore, we use the $\perp$ value for $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ when working on the assumptions for φ_1 and φ_2. Only when calculations for all sub-formulas are completed we begin calculating the proper result for $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$.

This process stops when the assumption calculated reaches a fix-point ($As^i = As^{i-1}$). Obviously, no new information will be revealed by performing another cycle. The set of states in $init(G)$ that are marked increases with each cycle, until all states that satisfy the formula are marked, and the algorithm stops. This is formally proved later.

Following is the recursive definition of the algorithm. Given a partition graph $G \in pg(P)$ of a program P, and an assumption $As : cl(\psi) \rightarrow (2^{end(G)} \cup \{\perp\})$, CheckGraph$(G, As)$ returns an assumption $As' : cl(\psi) \rightarrow (2^{init(G)} \cup \{\perp\})$.

CheckGraph(G,As):

For a single node n (labeled by a sub-program P'), return As' s.t. $\forall \varphi \in cl(\psi)$, if $As(\varphi) = \perp$ then $As'(\varphi) = \perp$, otherwise
$As'(\varphi) = MC[struct(P'), As](\varphi) \cap init(struct(P'))$.

The three possible recursive cases are the ones depicted in Figure 3.4. We assume that in_1 (in_2) is the entry node of G_1 (G_2), n_B is the condition node (if present), and n_D is the dummy node (if present).

- For a sequential composition $P_1; P_2$ (Figure 3.4(A)) perform:
  1. $As_1 \leftarrow$CheckGraph(G_2, As).
  2. $As' \leftarrow$CheckGraph(G_1, As_1).
  3. Return As'

- For a graph of $P =$ "if B then P_1 else P_2 fi" (Figure 3.4(B)) perform:

  1. $As_1 \leftarrow$CheckGraph(G_1, As).
  2. $As_2 \leftarrow$CheckGraph(G_2, As).
  3. $As_B \leftarrow CheckStepEdge(n_B \xrightarrow{true} in_1, As_1)$
  4. $As_{\neg B} \leftarrow CheckStepEdge(n_B \xrightarrow{false} in_2, As_2)$
  5. For every formula $\varphi \in cl(\psi)$, if $As_B(\varphi) = \bot$ then define $As'(\varphi) = \bot$ [8]. Otherwise, $As'(\varphi) = As_B(\varphi) \cup As_{\neg B}(\varphi)$
  6. Return As'

- For a graph of $P =$ "while B do P_1 od" (Figure 3.4(C)) perform:

  1. $As_{\neg B} \leftarrow CheckStepEdge(n_B \xrightarrow{false} n_D, As)$
  2. Find an ordering $\varphi_1, \varphi_2, \ldots, \varphi_n$ of the formulas in $cl(\psi)$ such that each formula appears after all of its sub-formulas. Set $As'(\varphi_i) = \bot$ for all i. For $k = 1, \ldots, n$ perform step 3 to define $As'(\varphi_k)$ [9].
  3. To define $As'(\varphi_k)$ perform one of the following, according to the form of φ_k:
     $\underline{\varphi_k \in AP}$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid s \models \varphi_k\}$
     $\underline{\varphi_k = \neg\varphi_l}$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid s \notin As'(\varphi_l)\}$.
     $\underline{\varphi_k = \varphi_l \vee \varphi_m}$: $As'(\varphi_k) \leftarrow As'(\varphi_l) \cup As'(\varphi_m)$.
     $\underline{\varphi_k \in \{\mathbf{AX}\varphi_l, \mathbf{EX}\varphi_l\}}$:
     (a) $Tmp \leftarrow$CheckGraph(G_1, As')
     (b) Let l be the location of the "while" and l' the first location in the body. $As'(\varphi_k) \leftarrow As_{\neg B}(\varphi_k) \cup \{(l, \sigma) \mid \sigma \models B \wedge (l', \sigma) \in Tmp(\varphi_l)\}$ [10].
     $\underline{\varphi_k \in \{\mathbf{A}(\varphi_l \mathbf{U} \varphi_m), \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)\}}$:
     (a) $Init_B \leftarrow As'(\varphi_m) \cap \{s \in init(G) \mid s \models B\}$.
     The initial assumption function is $As^0 \leftarrow As'$.
     Initialize the value for φ_k: $As^0(\varphi_k) \leftarrow As_{\neg B}(\varphi_k) \cup Init_B$.
     Set $i \leftarrow 0$.
     (b) do:
     - $i \leftarrow i + 1$
     - $Tmp =$CheckGraph(G_1, As^{i-1})
     - $TmpB \leftarrow CheckStepEdge(n_B \xrightarrow{true} in_1, Tmp)$

[8] Notice that since As_B and $As_{\neg B}$ both originate from the same assumption function As, it holds that $As_B(\varphi) = \bot$ iff $As_{\neg B}(\varphi) = \bot$. Also, the images of As_B and $As_{\neg B}$ are disjoint.

[9] Notice that when working on φ_k we have already calculated As' for all of its sub-formulas, so they are not $\bot$.

[10] The definition is the same for $\mathbf{AX}\varphi_l$ and $\mathbf{EX}\varphi_l$ because each state $(l, \sigma) \in init(G)$ has exactly one successor state $(l', \sigma) \in init(G_1)$.

- Define As^i so that for all $j < k$, $As^i(\varphi_j) \leftarrow As^0(\varphi_j)$ and $As^i(\varphi_k) \leftarrow TmpB(\varphi_k) \cup As_{\neg B}(\varphi_k)$

Until $As^i = As^{i-1}$.

(c) $As'(\varphi_k) \leftarrow As^i(\varphi_k)$

4. Return As'.

Theorem 3.2: The above algorithm computes model checking under assumptions correctly for any partition graph G of any program P. Formally, for any assumption function As: $CheckGraph(G, As) = MC[struct(G), As]$.

The proof of this theorem is quite long, and it is deffered to subsection 3.2.5. The consequence of the above theorem is the following:

Theorem 3.3: For any sequential process P, CTL formula ψ, partition graph $G \in pg(P)$ and empty assumption function $As : cl(\psi) \rightarrow \{\emptyset\}$, if $As' = \text{CheckGraph}(G, As)$ then for every $\varphi \in cl(\psi)$ and $s \in init(G)$, $s \in As'(\varphi) \Leftrightarrow s \models \varphi$.

This theorem states that if we run the algorithm on a sequential process, with an empty assumption function, the resulting function will give us full knowledge about which formulas in $cl(\psi)$ hold in the initial states of the program according to the standard semantics of CTL.

As promised, we now show how to make the algorithm more efficient by saving on recalculations done in recursive calls. When there is a recursive call to $CheckGraph$ on a smaller graph, all sub-formulas for which the input assumption function is not $\perp$ are calculated, even if they were already calculated in a previous call. To avoid this calculation, the results of calculations during recursive calls can be kept in files. Of course, the result for each node is kept in a separate file. When there is a second call to the same sub-graph, because this sub-graph is part of the body of a while loop, the input assumption function for sub-formulas that have been calculated before will not change (because we are now working on a larger formula), and so the resulting assumption function for these sub-formulas is identical. When this case is identified, the previous results can be read from the appropriate file instead of making the same calculations again. This scheme may improve the performance of the algorithm substantially, especially in systems where accessing the file is not too big a problem.

Another way of making the algorithm for a while loop more efficient is to work on sub-formulas of the same size together, instead of one at time. Any combination that guarantees that when working on a formula, all of its sub-formulas have already been dealt with, will suffice. The simplest way is to choose to work on all formulas of length 1, then all formulas of length 2, etc.

The space requirements of our modular algorithm will usually be better than that of algorithms that need to have the full model in the direct memory. Our algorithm holds in the direct memory at any particular moment only the model for the subprogram under consideration at that time. In addition, it keeps an assumption function, which at its largest holds the results of performing model checking on this subprogram. This of course is equivalent to any model checking algorithm that must keep its own results.

The time requirements depend on the model checking algorithm used for a single node and on the partition graph. However, experimental results show a saving in time as well as space requirements, as described in the next section.

3.2.4 Results

The modular model checking algorithm presented here can be considered as a framework into which any model checking algorithm for Kripke structures can be integrated. Since our method uses a given model checking algorithm as a procedure, whenever a better algorithm is developed it can immediately be plugged into ours.

An important new notion suggested here is that of partition graphs. These are used to partition the model checking task into several sub-tasks. They also maintain the flow of information (by means of assumption functions) between the sub-tasks. Choosing the right partition graph is crucial to the effectiveness of our method. As presented here, the algorithm is given a specific partition graph, but it may be possible to develop some heuristics that will allow automatic creation of the partition graph.

We implemented the modular model checking algorithm in a prototype tool called *SoftVer* (the tool itself is described in more detail in chapter 5). The tool is based on a BDD representation of models and on symbolic model checking.

In order to evaluate the effectiveness of partitioning on memory and time requirements, we applied the tool to a few small examples. Each example program was checked with two different partitionings. The moderate partitioning divided the program into a few components, while the extensive partitioning further divided it into smaller components. For comparison we also checked the unpartitioned full program.

The largest overhead occurs when applying our algorithm to a program in which the body of a while loop is partitioned. Therefore, all our examples include while loops which are divided by both partitionings. Table 3.1 (on page 64) gives the space and time used by each partitioning. Times are given in hours, minutes, and seconds, memory consumption is given Kilobytes. The "Min/Max module size" column gives the minimal and maximal sizes of the files that keep the structures of single partition graph nodes. These are also measured in Kilobytes. The examples were run on a machine with 400M RAM.

Besides the partitioning of the program, the influence of two other parameters was checked. The first is the use of *variable reordering* (the "reo" option in the table). This is an option provided by the BDD package. When the BDD sizes go over a certain threshold a variable reordering algorithm is performed. This algorithm attempts to find a different ordering of the BDD variables, so that the BDDs will be smaller. This option has proved to be very useful for symbolic model checking. The BDD library we used enabled us to use different variable orderings for different partition-graph nodes (which reside in separate files). The second parameter is *local reachability* (the "lr" option in the table). This option does not perform full reachability analysis, but does eliminate from the state space a certain amount of the unreachable states. This option is explained in more detail in Chapter 5.

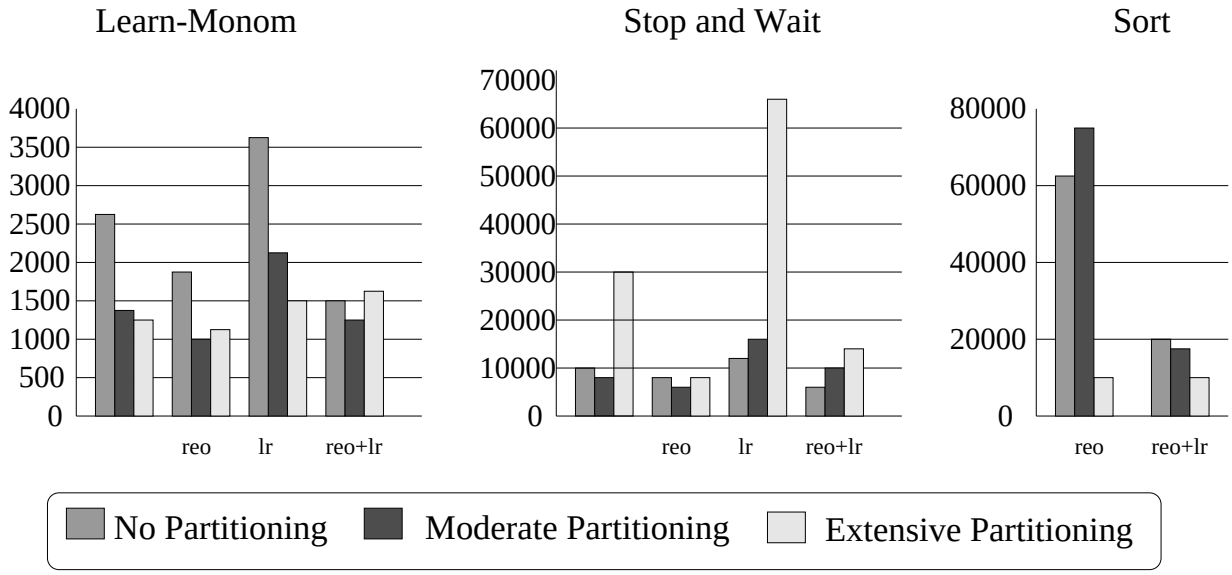
Table 3.1 gives results for three examples. The **Learn Monom** example is a learning algorithm that learns a single term by examples. The moderate partitioning separates the code into several components so that different tasks are in different components. The extensive partitioning breaks down the components even further. The specification used requires that the algorithm will never give a false negative result, i.e. the algorithm does not make errors on inputs for which the term evaluates to *true*.

The **Stop and Wait** example is a simulation of the "stop and wait" communication protocol. It consists of a large while loop whose body has two major parts, one for the sender and one for the receiver. The moderate partitioning separates the receiver and the sender into two components. The extensive partitioning further separates different tasks in each of the components. The specification used requires that the sender does not move to the next message before the current message has arrived at the receiver.

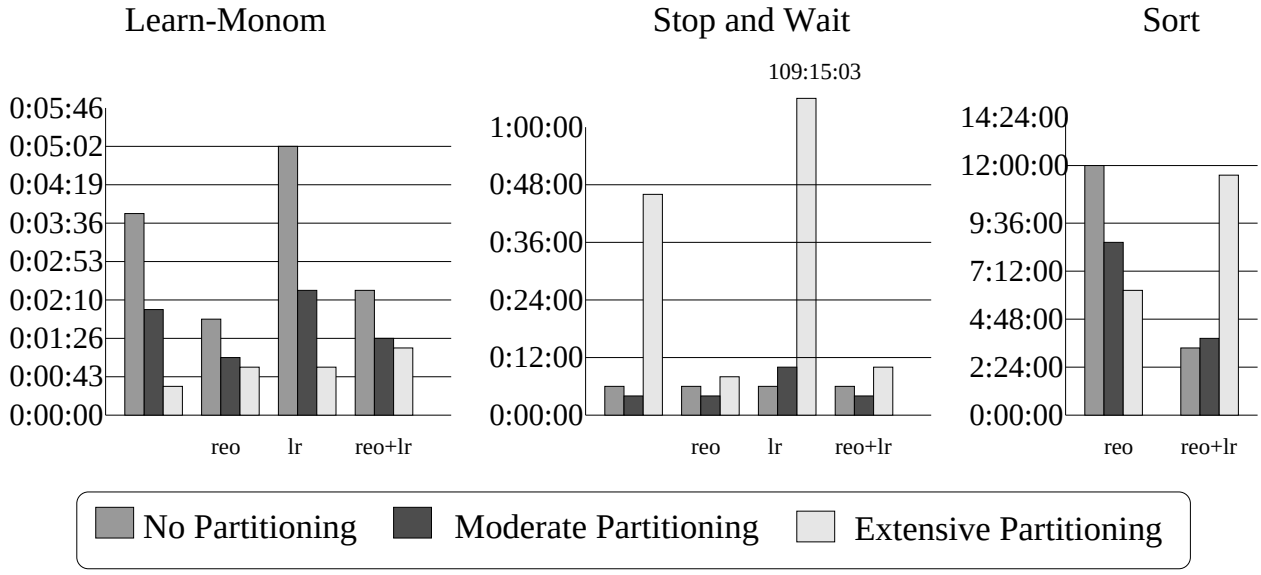
The **Sort** example is the shortest of the three, and it performs bubble sort on an array of 5 elements. This program consists of two nested while loops, with a single "if" statement in the body of the inner loop (that compares two adjacent elements). The moderate partitioning breaks the outer loop and the extensive partitioning breaks the inner loop as well. The specification states that the algorithm will terminate, and at that point the array will be sorted. All three examples work with 8-bit integers. In the first two examples integers are used mainly as pointers into buffers or array indices, but in the Sort example the array which is sorted is an array of integers. For this reason, even though it is the shortest example (least number of program counter locations) it is not the smallest in terms of the size of its state-space. This example could not complete in the available 400M when run without variable reordering, so there are no entries in the table for this example without the reordering option.

To make the results more easy to analyze, we summarized them in TWO graphs. Figure 3.7(a) shows the effect of partitioning on memory consumption and figure 3.7(b) shows the effect of partitioning on time consumption. Each graph contains three sub-graphs, one for each example.

From figure 3.7(a) we can see that partitioning a program can in fact reduce space consumption considerably. The sort example is the most notable since it is a type of application which usually does not work well with BDDs. However in some cases partitioning the program can also enlarge the amount of space required. This happens only because we used BDDs to represent structures. When implementing the algorithm using an explicit-state representation this phenomenon can not happen. The moderate partitioning gives good results in 7 out of 10 cases, whereas the extensive partitioning gives good results only in 5 cases (out of which only in 4 cases the extensive partitioning gives better results than the moderate partitioning). On the other hand, the best result overall was achieved by the extensive partitioning in the Sort example. These results support our claim that the choice of partitioning is crucial. More research needs to be done in order to classify the types of programs in which partitioning is expected to improve performance, and the way in which such programs should be partitioned. When choosing a partition one must balance the overhead of partitioning with the



(a) Graph of memory consumption



(b) Graph of time consumption

Figure 3.7: Result graphs for Modular Model Checking

reduction in the sizes of the modules.

By comparing figures 3.7(a) and 3.7(b) we see that there is no direct link between the space and time consumption. There are 2 cases in which although the extensive partitioning leads to a greater space consumption than the moderate partitioning, the extensive partitioning requires less run-time than the moderate one, and one case where the opposite happens. This can be explained by the fact that in the extensive partitioning there is one module for whom the model checking at one point exploded and required more space, but since all other modules were small the model checking for all the other modules took much less time. The space consumption listed in the table is the *maximum* space used by the process, but in most cases this amount is used only for a short period of time.

Another point to notice is the ill effect of local reachability and the favorable effect of variable reordering. These will be further investigated and explained in Chapter 5.

To summarize our conclusions from the examples, we believe that the use of our modular model checking algorithm can be instrumental in verifying some programs, but not in every case. We would advise the user to use it when it seems that regular model checking is too expensive, and to choose the partitioning so that the different tasks that the program performs reside in different modules.

3.2.5 Proof of main theorem

This subsection is devoted to proving theorem 3.2 which stated that:
The modular model checking algorithm computes model checking under assumptions correctly for any partition graph G of any program P. Formally, for any assumption function As over $end(G)$: $CheckGraph(G, As) = MC[struct(G), As]$.

The proof of this theorem requires several stages. We start with the following lemma.

Lemma 3.4: Let $M = \langle S, R, I \rangle$ be the structure of some program P. Let $M' = \langle S', R', I' \rangle$ be the structure of a sub-program P' of P, or of an edge e in the control-flow graph of P. Let As be an arbitrary assumption function over $end(M)$ and As' an assumption function over $end(M')$ such that for every $s \in end(M')$ and $\varphi \in cl(\psi)$ it holds that $s \in As'(\varphi) \Leftrightarrow M, s \models_{As} \varphi$ [11]. Then for every $s \in M'$ and every $\varphi \in cl(\psi)$: $M', s \models_{As'} \varphi \Leftrightarrow M, s \models_{As} \varphi$.

Proof: The proof is by induction on the top-most operator of φ. For the base case, $\varphi \in AP$, and the induction steps that do not involve temporal operators the proof is trivial since the definition of $M, s \models_{As'} \varphi$ does not depend on the assumption function As' (only on the induction hypothesis for sub-formulas of φ). The interesting cases are the ones that involve temporal operators. We show only the proofs for $\mathbf{EX}\varphi_1$ and $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$, the proofs for $\mathbf{AX}\varphi_1$ and $\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ are dual. The induction hypothesis is that for φ_1 and φ_2 the claim holds, i.e. for every $s \in S'$: $M', s \models_{As'} \varphi_i \Leftrightarrow M, s \models_{As} \varphi_i$

[11] The theorem will also hold if As is over a set that includes $end(M)$ instead of being defined exactly over $end(M)$ (and the same for As' and $end(M')$).

for $i = 1, 2$.

- Let $\varphi = \mathbf{EX}\varphi_1$. For every state $s \in S'$ we have:
  $M', s \models_{As'} \varphi \Leftrightarrow$
  $\exists s' \in S'.R'(s, s') \wedge M', s' \models_{As'} \varphi_1$ or $s \in end(M') \cap As'(\varphi)$
  $\Leftrightarrow$ (since $S' \subseteq S$, $R' \subseteq R$, and since $R(s, s') \wedge s \notin end(M')$ implies $s' \in S'$)
  $\exists s' \in S.R(s, s') \wedge M, s' \models_{As} \varphi_1$ or $s \in end(M') \cap As'(\varphi)$
  $\Leftrightarrow$ (from our assumption about $As'(\varphi)$)
  $M, s \models_{As} \varphi$.

- Let $\varphi = \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$. We first prove that for every state $s \in S'$: $M', s \models_{As'} \varphi \Rightarrow M, s \models_{As} \varphi$:
  $M', s \models_{As'} \varphi$
  $\Rightarrow$ (by definition)
  There exists a path $\pi = s_0, \ldots, s_i$ in M' $(s = s_0)$ such that either $s_i \in end(M') \cap As'(\varphi)$ or $M', s_i \models_{As'} \varphi_2$, and for every $j < i$: $M', s_j \models_{As'} \varphi_1$
  $\Rightarrow$ (from the induction hypothesis about φ_1 and φ_2 and what we know of As')
  There exists a path $\pi = s_0, \ldots, s_i$ in M $(s = s_0)$ such that for every s_i, either $M, s_i \models_{As} \varphi$ or $M, s_i \models_{As} \varphi_2$, and for every $j < i$: $M, s_j \models_{As} \varphi_1$
  $\Rightarrow$ (by definition)
  $M, s \models_{As} \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2 \vee \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$
  $\Rightarrow M, s \models_{As} \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$.

  Next, we prove that for every state $s \in S'$, $M, s \models_{As} \varphi \Rightarrow M', s \models_{As'} \varphi$. If $M, s \models_{As} \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ then there exists a path $\pi = s_0, \ldots, s_i$ in M $(s = s_0)$ such that $M, s_i \models \varphi_2$ and for every $j < i$: $M, s_j \models_{As} \varphi_1$. Here there are two possibilities:

  1. For every $k \leq i$, $s_k \in S' \setminus end(M')$. In this case, from the induction hypothesis about φ_1 and φ_2 we have:
     $M', s_i \models_{As'} \varphi_2$ and for every $j < i$: $M', s_j \models_{As'} \varphi_1$
     $\Rightarrow M', s \models_{As'} \varphi$
  2. There exists a state from $end(M')$ along π. Let s_k be the first such state (so for every $j < k$ we have $s_j \in S' \setminus end(M')$). Then we have:
     $M, s_k \models_{As} \varphi$ and for every $j < k$: $M, s_j \models_{As} \varphi_1$
     $\Rightarrow$ (from our assumption about As' and the induction hypothesis)
     $s_k \in As'(\varphi)$ and for every $j < k$ $M', s_j \models_{As'} \varphi_1$
     $\Rightarrow$ (by definition)
     $M', s \models_{As'} \varphi$

□

The proof of theorem 3.2 is by induction on the size of the partition graph G. The most complicated part to prove is the operation of the algorithm on a while loop. This

part is proven by induction on the sub-formulas φ_k. The most complicate part of this inner induction is, naturally, the proof for $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ or $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$. In the following we prove several lemmas about this part, and then use them in the proof for the whole algorithm. These lemmas are going to be used in the inner-most induction, so the induction hypotheses may be needed as premises.

The first premise is the induction hypothesis of the outermost induction on the size of the partition graph.

Premise 1:

Every recursive call of CheckGraph on the partition graph of the body of the while (G_1) with an (arbitrary) assumption function As^{in} over $end(G_1)$, results in an assumption function As^{out} over $init(G_1)$ such that, for every $\varphi \in cl(\psi)$ s.t. $As^{in}(\varphi) \neq \bot$ and every state $s \in init(G_1)$ we have $s \in As^{out}(\varphi) \Leftrightarrow struct(G_1), s \models_{As^{in}} \varphi$.

The second premise is the induction hypothesis for the induction on the sub-formulas φ_k.

Premise 2:

After running the algorithm for the formulas $\varphi_1, \ldots, \varphi_{k-1}$ the algorithm built the sets $As'(\varphi_1), \ldots, As'(\varphi_{k-1})$ correctly, i.e. for every $s \in init(G)$ and $j < k$ we have $s \in As'(\varphi_j) \Leftrightarrow struct(G), s \models_{As} \varphi_j$.

To reason about the algorithm for the while loop we define the following three structures. The structure of the partition graph G of the whole while-loop is $M_G = \langle S_G, R_G, I_G \rangle$. The structure of the partition graph G_1 of the body of the while loop is $M_1 = \langle S_1, R_1, I_1 \rangle$. We define a third structure $M = \langle S, R, I \rangle$ which is the structure M_G without the transitions of the false edge $e_{\neg B} = n_B \xrightarrow{false} n_D$. This means that $S = S_G \setminus end(G)$ ($end(G)$ is the set of states that have the program location following the while loop, the states you end up at when the loop is done), and $R = R_G \setminus R_{e_{\neg B}}$. The following observations are at the base of our proof:

- $end(M_1) = init(M_G) = init(M)$
  The set of ending states of the body of the loop is the set of initial state of the whole loop. These are the states in which the condition of the while is evaluated. Their successors are either in $init(M_1)$, if the condition is true, or in $end(M_G)$, if it is false.

- $end(M) \subseteq init(M)$
  The ending states of M are the states in which the condition B is evaluated and found not to hold. The transitions outgoing from these states are included in M_G, but not in M.

- For checking satisfaction under assumptions on M_G the assumption function must be defined over $end(M_G)$, which is the set of states with the location following the while loop. For checking satisfaction under assumptions on M the assumption function must be defined over $end(M)$. Any assumption function defined over $init(M)$ is also defined over $end(M)$ and can be used for satisfaction under assumption of states in M.

We start by connecting satisfaction of formulas in M_G under the assumption As with satisfaction in M under the assumption $As_{\neg B}$. Recall that As is over $end(M_G)$ and $As_{\neg B}$ is over $end(M)$.

Lemma 3.5: For every $s \in init(G)$ and $\varphi \in cl(\psi)$: $M_G, s \models_{As} \varphi \Leftrightarrow M, s \models_{As_{\neg B}} \varphi$.

Proof: The assumption function $As_{\neg B}$ is created by a call to CheckStepEdge on the false edge, with As. Therefore, for every $s \in init(G)$ such that $s \not\models B$ and $\varphi \in cl(\psi)$ we have $s \in As_{\neg B}(\varphi) \Leftrightarrow M_G, s \models_{As} \varphi$. We have already noted that the set of states $s \in init(G)$ such that $s \not\models B$ is exactly $end(M)$, so we can use lemma 3.4 to conclude that for every $s \in init(M)$: $M, s \models_{As_{\neg B}} \varphi \Leftrightarrow M_G, s \models_{As} \varphi$. □

From here on, we prove correctness with respect to M and $As_{\neg B}$, and this will imply correctness with respect to M_G and As.

The next lemma we prove makes two claims. One is that for every $s \in As^i(\varphi_k)$: $M, s \models_{As_{\neg B}} \varphi_k$. This means that every state that is marked as satisfying φ_k, actually does satisfy φ_k. The second claim is that the series of sets $As^i(\varphi_k)$ is a monotonically growing series of sets, i.e. each new set includes all the states of the previous set. This means that we do not "lose" states that have already been found to satisfy φ_k. Notice that when calculating the series of assumption functions As^i, the sets $As^i(\varphi_j)$ for $j < k$ are always the same, because they are copied from As^0. The goal of this part of the algorithm is only to compute the set $As'(\varphi_k)$. We also notice here that the assumption functions As^i are defined over $init(M)$, which includes $end(M)$. During the calculation of these functions we create the assumption function Tmp over $init(G_1)$ and $TmpB$ over the set of states in $init(M)$ that satisfy B.

Lemma 3.6: Assume that premises 1 and 2 hold. Let As^i be one of the assumption functions calculated by the algorithm when working on a formula φ_k of the form $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ or $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ on a graph of a while loop. Then for every state $s \in As^i(\varphi_k)$, $M, s \models_{As_{\neg B}} \varphi_k$, and if $i > 0$ then $As^{i-1}(\varphi_k) \subseteq As^i(\varphi_k)$.

Proof: We prove both claims of the lemma together, by induction on i.

To show that $s \in As^0(\varphi_k)$ implies $M, s \models_{As_{\neg B}} \varphi_k$ we examine part (a) of this portion of the algorithm, where As^0 is created. If $s \in Init_B$ then s is an initial state that satisfies B and, according to the induction hypothesis for φ_m, also satisfies φ_m under the assumption $As_{\neg B}$. Therefore, $M, s \models_{As_{\neg B}} \varphi_k$. If, on the other hand, $s \in As_{\neg B}(\varphi_k)$ then $s \in end(M)$ and by definition $M, s \models_{As_{\neg B}} \varphi_k$. To show that $As^0(\varphi_k) \subseteq As^1(\varphi_k)$ we move to examine part (b). We see that $As^1(\varphi_k) = TmpB(\varphi_k) \cup As_{\neg B}(\varphi_k)$. Let s be a state in $As^0(\varphi_k)$. If $s \in As_{\neg B}(\varphi_k)$ then automatically $s \in As^1(\varphi_k)$. Otherwise, $s \in Init_B$, which means that $s \models_{As} \varphi_m$, and from our induction hypothesis for φ_m (premise 2) we conclude that CheckStepEdge will create $TmpB$ correctly (states that satisfy φ_m also satisfy $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ and $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) and $s \in As^1(\varphi_k)$.

We now assume both claims hold for every $j < i$, and show them for As^i. Let $s \in init(M)$ be a state such that $s \in As^i(\varphi_k)$. We show that $M, s \models_{As_{\neg B}} \varphi_k$. If $s \in As_{\neg B}(\varphi_k)$ then as before $M, s \models_{As_{\neg B}} \varphi_k$. Assume that $s \in TmpB(\varphi_k)$. The state s has a single successor, which is a state $s' \in init(M_1)$. If CheckStepEdge inserted

s into $TmpB(\varphi_k)$ then it must be that $s' \in Tmp(\varphi_k)$ (assuming that $s \not\models_{As_{\neg B}} \varphi_m$). From the induction hypothesis for the recursive call to CheckGraph we deduce that $M_1, s' \models_{As^{i-1}} \varphi_k$. If $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ then there must be a path $\pi = s_0, \ldots, s_n$ ($s' = s_0$) in M_1 such that either $s_n \in end(M_1) \cap As^{i-1}(\varphi_k)$ or $M_1, s_n \models_{As^{i-1}} \varphi_m$, and for every $j < n$: $M, s_j \models_{As^{i-1}} \varphi_l$. From premises 1 and 2 together we deduce that for every $j < n$: $M, s_j \models_{As_{\neg B}} \varphi_l$. If $s_n \in end(M_1) \cap As^{i-1}(\varphi_k)$ then $M, s_n \models_{As_{\neg B}} \varphi_k$, and if $M_1, s_n \models_{As^{i-1}} \varphi_m$ again we have $M, s_n \models_{As_{\neg B}} \varphi_k$. In both cases we conclude that $M, s' \models_{As_{\neg B}} \varphi_k$ which implies $M, s \models_{As_{\neg B}} \varphi_k$. The proof for $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ is similar, except that instead of reasoning about one path π from s' we talk about all the paths from s'.

We are left with showing that $As^{i-1}(\varphi_k) \subseteq As^i(\varphi_k)$. Let s be a state in $As^{i-1}(\varphi_k)$. The set $As^{i-1}(\varphi_k)$ is created at the end of the "Until" loop, as the union of $TmpB(\varphi_k)$ and $As_{\neg B}(\varphi_k)$.

If $s \in As_{\neg B}(\varphi_k)$ then it will stay there (since $As_{\neg B}(\varphi_k)$ does not change) and so we are guaranteed that $s \in As^i(\varphi_k)$. If $s \in TmpB(\varphi_k)$ then we show that it will also be in $TmpB(\varphi_k)$ when $As^i(\varphi_k)$ is created. To prevent confusion, we use $TmpB^j$ and Tmp^j for the functions $TmpB$ and Tmp created in the jth iteration of the loop, which is the iteration in which $i = j$ and As^j is created.

From the induction hypothesis on the recursive call to CheckGraph, the correctness of CheckStepEdge, and lemma 3.4, we deduce that for every state $t \in init(M)$, and every $j \leq i$, $t \in TmpB^j(\varphi_k) \Leftrightarrow M, t \models_{As^{j-1}} \varphi_k$. Obviously, if $s \in As^{i-1}$ because $M, s \models_{As_{\neg B}} \varphi_m$ then $s \in TmpB^i(\varphi_k)$ and so $s \in As^i(\varphi_k)$. Otherwise, if $s \in As^{i-1}(\varphi_k)$ it is because $M, s \models_{As^{i-2}} \varphi_k$. We know that for every $j < k$ we have $As^{i-2}(\varphi_j) = As^{i-1}(\varphi_j)$, and that $As^{i-2}(\varphi_k) \subseteq As^{i-1}(\varphi_k)$, so $M, s \models_{As^{i-2}} \varphi_k$ implies $M, s \models_{As^{i-1}} \varphi_k$. This last step is true because of the monotonicity of the **EU** and **AU** operators. In the definition of satisfaction under assumptions we see that if the sets of states assumed to satisfy $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ (or $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) grow larger, then the set of states that satisfy $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ (or $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) under this assumption cannot grow smaller. So now we know that $M, s \models_{As^{i-1}}$, and this implies $s \in TmpB(\varphi_k)$, i.e. $s \in As^i(\varphi_k)$. □

Lemma 3.7: Assume that premises 1 and 2 hold. For every state $s \in init(M)$ such that $M, s \models_{As_{\neg B}} \varphi_k$ ($\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ or $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) there exists a number i such that $s \in As^i(\varphi_k)$.

Proof: We prove lemma 3.7 for the case of $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$, the proof for $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ follows a similar line of reasoning.

Let s be a state in $init(M)$ such that $M, s \models_{As_{\neg B}} \varphi_k$. Let $\pi = s_0, \ldots, s_n$ ($s = s_0$) be the shortest path from s that proves this, i.e. it is the shortest path such that either $s_n \in end(M) \cap As_{\neg B}(\varphi_k)$ or $M, s_n \models_{As_{\neg B}} \varphi_m$, and for every $j < n$: $M, s_j \models_{As} \varphi_l$. We define n to be the *depth* of s.

Assume to the contrary that there are one or more states that satisfy φ_k but will not get into any set $As^i(\varphi_k)$. Obviously, these must be states that satisfy B. Let s be the state with minimal depth, and n its depth. Let $\pi = s_0, \ldots, s_n$ be the path that shows this. Then there are two possibilities:

1. The state s_0 is the only state from $init(M)$ along π. In this case we know that the states $s_1, \ldots, s_n$ are all inside M_1. From the premises of the lemma we know that when the first recursive call to CheckGraph on G_1 is done we have $s_1 \in Tmp(\varphi_k)$ because $M_1, s_1 \models_{As^0} \varphi_k$ can be proven on M_1 just by examining the sets of states that satisfy φ_l and φ_m in M'. Therefore we must conclude that CheckStepEdge will create $TmpB$ so that $s_0 \in TmpB(\varphi_k)$ which implies $s_0 \in As^1(\varphi_k)$ in contradiction to the assumption that $s = s_0$ does not get into any set $As^i(\varphi_k)$.

2. There exists a state other than s_0 along π which is in $init(M)$ (the path π goes more than once through the loop). Let s_j $(j > 0)$ be the first such state along π. Since π was chosen as the shortest path that proves $M, s_0 \models_{As_{\neg B}} \varphi_k$, we know that $s_j, \ldots, s_n$ is the shortest path that proves $M, s_j \models_{As_{\neg B}} \varphi_k$, and that the depth of s_j is strictly smaller than the depth of s_0. From the way we chose s we conclude that there is a function As^i such that $s_j \in As^i(\varphi_k)$. This means that $M, s_0 \models_{As^i} \varphi_k$. For similar reasoning as we have done before, using the two premises, we conclude that $s_0 \in As^{i+1}(\varphi_k)$ in contradiction to our assumption that $s = s_0$ never gets into any set $As^i(\varphi_k)$.

□

We can now finally prove theorem 3.2.

Proof: As mentioned before, the proof of the algorithm is by induction on the size of the partition graph. The base case is the application of the given model checking under assumptions procedure on the structure of a single partition graph node. For this case we know that the given procedure is correct. The induction step is according to the topmost structure of the partition graph G. In all the cases bellow we use M_G for the structure of P and M_i for the structure of P_i $(i = 1, 2)$.

- For a graph of $P = P_1; P_2$ as in figure 3.4(A):
  From the induction hypothesis for the smaller graph G_2 we conclude that for every state $t \in init(G_2)$ and every formula $\varphi \in cl(\psi)$ such that $As(\varphi) \neq \bot$, $t \in As_1(\varphi) \Leftrightarrow M_2, t \models_{As} \varphi$. Similar reasoning shows that for every $s \in init(G_1)$, $s \in As'(\varphi) \Leftrightarrow M_1, s \models_{As_1} \varphi$. Using lemma 3.4, and since $init(G_1) = init(G)$ we have: for every state $s \in init(G)$ and every formula $\varphi \in cl(\psi)$ such that $As(\varphi) \neq \bot$, $s \in As'(\varphi) \Leftrightarrow M_G, s \models_{As} \varphi$.

- For a graph of $P =$ "if B then P_1 else P_2 fi" as in figure 3.4(B):
  We use the induction hypothesis for the recursive calls on G_1 and G_2 to conclude that for every $t \in init(G_i)$ and φ such that $As(\varphi) \neq \bot$, $t \in As_i(\varphi) \Leftrightarrow M_i, t \models_{As} \varphi$ $(i = 1, 2)$. From the correctness of CheckStepEdge, and using lemma 3.4, we conclude that for every $s \in init(G)$ and every φ such that $As(\varphi) \neq \bot$, $s \in As'(\varphi) \Leftrightarrow M_G, s \models_{As} \varphi$.

- For a graph of $P =$ "while B do P_1 od" as in figure 3.4(C):
  For this part we use induction on k.

  For $\varphi_k \in AP$, $\varphi_k = \neg\varphi_l$ and $\varphi_k = \varphi_l \vee \varphi_m$ the proof is trivial, since we assume correctness for φ_l and φ_m (the induction hypothesis).

  For $\varphi_k =$**AX**φ_l or $\varphi_k =$**EX**φ_l, we use the induction hypothesis for the recursive call on G_1 to conclude that for every $t \in init(G_1)$, $t \in Tmp(\varphi_l) \Leftrightarrow M_{G_1}, t \models_{As} \varphi_l$. For every state $s \in init(G)$ such that $s \not\models B$, we know that $s \in As_{\neg B}(\varphi_k) \Leftrightarrow s \models_{As} \varphi_k$ from lemma 3.5. For every state $s \in init(G)$ such that $s \models B$ we know that s has only one successor, so $M_G, s \models \varphi_k$ iff its successor state t satisfies $t \in Tmp(\varphi_l)$. Therefore, for every state $s \in init(G)$ we have $s \in As'(\varphi_k) \Leftrightarrow M_G, s \models_{As} \varphi_k$.

  For $\varphi_k =$**A**$(\varphi_l$**U**$\varphi_m)$ or $\varphi_k =$**E**$(\varphi_l$**U**$\varphi_m)$ we have already proven, using both lemma 3.5 and lemma 3.6, that for every $s \in init(G)$, $s \in As'(\varphi_k) \Rightarrow M_G, s \models_{As} \varphi_k$. This proves that every state we mark actually satisfies the formula. We have also shown in lemma 3.6 that with every iteration the set $As^i(\varphi_k)$ can only grow larger, and since there are finitely many states in $init(G)$ the process is guaranteed to stop. In lemma 3.7 we have shown that any state which satisfies φ_k will eventually get into one of the sets $As^i(\varphi_k)$ and so we have $M_G, s \models_{As} \varphi_k \Rightarrow s \in As'(\varphi_k)$, which concludes the proof for the while loop.

□

3.3 Static Analysis Reductions

In this section we use static analysis to reduce the Kripke structures representing programs. As mentioned earlier, a program can be described in two levels, a syntactic and a semantic level. We use control-flow graphs to represent the syntax of programs and Kripke structures for the semantics of programs. Static analysis is the process of examining the control-flow graph of a program (the syntax) to extract information on its semantics, without creating the semantic model. Our purpose is to use static analysis in order to create a smaller semantic model for the program, which we call the *reduced* Kripke structure of the program. The reduced Kripke structure is built so that it is equivalent to the original structure of the program, i.e. a given specification φ is true in the original Kripke structure iff it is true in the reduced one. In fact, the reduced structure will be equivalent to the original structure with respect to any formula that refers to the same set of variables as φ. The specifications we consider are formulas of the logic CTL*, and the logic CTL*-X.

Our algorithms for static analysis are based on syntactic manipulation of expressions, and therefore we allow variables with both finite and infinite domains. When the domains are finite, our methods can be used for automatic verification using an explicit representation of the Kripke structure as well as for verification using a BDD

representation. In either case, the verification algorithm itself is not changed, it just receives a smaller model to work on.

Using static analysis we can create a reduced Kripke structure for a program directly out of the control-flow graph, and never build the full structure. We are therefore able to verify systems that would otherwise be too big to handle.

The advantage of our approach is even more significant when the system is composed of several processes. In such a case, each process is reduced separately and only then they are composed to one Kripke structure. This solution thus serves to reduce the exponential blow-up that occurs when taking the cross product of the Kripke structures of the individual processes.

Another important advantage of using static analysis is that in order to implement our reductions, changes are made only to the compiler (which is relatively simple to do) and there is no need to change the verification tool or the verification algorithm. This enables integration with existing tools at a very low cost. It also means that the overhead of using our reductions is during the (very short) compilation stage and not in the verification process.

We present and compare two methods that use static analysis to create reduced models for programs. The first method, called *path reduction*, reduces according to control, and the second, called *dead-variable reduction*, reduces according to data.

Path reduction creates an equivalent Kripke structure in which there are less program counter locations. We identify paths in the control flow graph on which a process performs a series of consecutive operations that can not influence the specification. Each such path is replaced by a single transition, representing the computation along this path (instead of a series of transitions).

Dead-variable reduction reduces the Kripke structure by excluding some of the possible values that variables can take at given points in the program. We find places in the program in which the value of a given variable is insignificant, and prune out of the program model all the states that differ only on that variable. A variable x is *dead* at a certain point in the code if on all computations from that point on a value is assigned to x before its value is used. This means that the current value stored in the dead variable can not influence the computation. This definition is used traditionally in compiler optimization methods. We use this information in order to reduce the state space of the program by ignoring variable values when the variables are dead.

Furthermore, we expand the traditional definition of dead variables so that a variable can be *partially dead*. Instead of variables being either dead or not at a given point in the program, we define a condition that implies that the variable is dead. Given a variable x, we compute a condition $\mathrm{dead}(l)$ for every program location l so that $\mathrm{dead}(l)$ describes the set of states at location l for which the value of x can be discarded.

Both of our reductions can be used in conjunction with the modular model checking algorithm of the previous section. We explain how to do this in sub-section 3.3.3.

3.3.1 Path Reduction

Our first static analysis reduction is based on compression of computation paths. We identify computation paths along which each state has a single successor, and which can be compressed into a single step without affecting the satisfaction of the specification formula. The specification language preserved by this reduction is CTL*-X.

Path Reduction for Sequential Processes

In this section we define a reduction which can be applied to any sequential program written in our language of non-deterministic while programs. Later on we use the reduction for sequential programs to create a similar reduction for parallel programs.

Given a program P and its control-flow graph CF_P, a reduced Kripke structure $reduced(CF_P)$ is created directly from CF_P. The structure $reduced(CF_P)$ will have less states and less transitions than the original structure $struct(CF_P)$ (which was defined as the semantics of P), but will be equivalent with respect to any CTL*-X formula over some given set of atomic propositions.

We define a set of *breaking points*, which are nodes in the control-flow graph, so that all the commands that may influence the specification are considered breaking points.

When using CTL* specifications (or any subset of CTL*) for programs we assume that the atomic propositions in AP are expressions over program variables. Obviously, the only variables that can influence the satisfaction of a formula are those that appear in (at least) one of these expressions. We call such variables *visible variables*. For the remaining of the discussion on path reduction we fix a CTL*-X formula φ as the specification to be verified.

Definition 3.9: Given a control-flow graph CF_P, the set of *breaking points* BP is a set of graph nodes s.t. $n \in BP$ iff one of the following holds:

1. n is the initial or terminating program location,
2. n is associated with the program location of an assignment that changes a visible variable,
3. n is associated with the program location of a non-deterministic assignment, or
4. n is the head of a "while" statement.

Definition 3.10: A finite simple path $v_1, \ldots, v_k$ in a control-flow graph is called *elementary* if both v_1 and v_k are breaking points, and no other node on the way is a breaking point.

Notice that every elementary path is finite, since every loop in the control-flow graph is broken by at least one breaking point. The set of breaking points was chosen so that elementary paths have two properties. One is that along any elementary path only the first statement might influence the specification. This is why assignments

that may influence atomic propositions and non-deterministic assignments must be breaking points. The second property is that in a single traversal of an elementary path it is possible to compute its underlying semantics, i.e. under what conditions it will execute and what happens to the values of variables when it is executed. This property requires that non-deterministic assignments will be breaking points and that every loop will contain at least one breaking point.

The path-reduced Kripke structure for P, denoted by $reduced(CF_P) = \langle S, R, I \rangle$, is defined so that $S = BP \times \Sigma$ and $I = \{l\} \times \Sigma$ where l is the initial location of the program (which must be a breaking point). Every elementary path τ in CF_P induces a transition relation R_τ so that $R = \bigcup_\tau R_\tau$. In order to determine R_τ we compute for every such path τ the *reachability condition* $RC_\tau : \Sigma \to \{true, false\}$ and the *state transformation* function $ST_\tau : \Sigma \to \Sigma$. The definitions of these functions are adapted from the Floyd proof system [27, 28]. The reachability condition $RC_\tau(\overline{x})$ is a condition on the variables at the beginning of τ that is true iff τ can be traversed. The state transformation function $ST_\tau(\overline{x})$ is a function on states that describes the value of the variables at the end of τ as a function of their values at the beginning of τ, provided that τ is indeed traversed. Both of these are computed syntactically from the control flow graph (i.e. the program text), by manipulation of terms, as described bellow.

As mentioned before, we use $\overline{x} = x_1, \ldots, x_n$ to denote the vector of program variables. In this subsection we are dealing with a single sequential process, so $\overline{x}$ is the set of local variables of this process.

Definition 3.11: Let $\tau = v_1 \to \ldots \to v_m$ be a finite path in CF_P. We define RC_τ^k and ST_τ^k to be the corresponding reachability condition and state transformation function for the suffix $v_k \to \ldots \to v_m$, by induction on k going from $k = m$ to $k = 1$. We use the notation $expr[a \leftarrow e]$ for the expression that results from exchanging every occurrence of a in $expr$ by the expression e.

Induction basis:
$RC_\tau^m(\overline{x}) = true, \;\; ST_\tau^m(\overline{x}) = \overline{x}.$

Induction step:
RC_τ^k and ST_τ^k are defined according to the command labeling the node v_k for $1 \leq k < m$:

Skip: $RC_\tau^k = RC_\tau^{k+1}, \;\; ST_\tau^k = ST_\tau^{k+1}$

An assignment $y := expr$**:** $RC_\tau^k = RC_\tau^{k+1}[y \leftarrow expr], \;\; ST_\tau^k = ST_\tau^{k+1}[y \leftarrow expr]$

An array assignment $a\ [\ expr_1\ ] := expr_2$**:**
$RC_\tau^k = RC_\tau^{k+1}[a[expr_1] \leftarrow expr_2] = RC_\tau^{k+1}[a \leftarrow (a; expr_1 : expr_2)]$,
$ST_\tau^k = ST_\tau^{k+1}[a[expr_1] \leftarrow expr_2] = ST_\tau^{k+1}[a \leftarrow (a; expr_1 : expr_2)]$ [12]

[12] The notation $(a \leftarrow (a; exp_1 : exp_2))$ means that the array a is substituted with an array which is identical to a except that in the exp_1 cell there is the value exp_2. See [28] for further details on its necessity and use in Hoare style proof systems.

A non-deterministic assignment $y := \{expr_1, \ldots, expr_l\}$**:** Let $expr_i$ be the label on the edge $v_k \rightarrow v_{k+1}$, then $RC_\tau^k = RC_\tau^{k+1}[y \leftarrow expr_i]$, $ST_\tau^k = ST_\tau^{k+1}[y \leftarrow expr_i]$

Positive test: This case is when v_k is an "if" or "while" and v_{k+1} is the positive son. Let B be the boolean condition of the command. Then $RC_\tau^k = RC_\tau^{k+1} \wedge B$, $ST_\tau^k = ST_\tau^{k+1}$

Negative test: This case is when v_k is an "if" or "while" and v_{k+1} is the negative son. Let B be the boolean condition of the command. Then $RC_\tau^k = RC_\tau^{k+1} \wedge \neg B$, $ST_\tau^k = ST_\tau^{k+1}$

Terminate: This case can happen only when the edge $v_k \rightarrow v_{k+1}$ is the self loop of the terminate command.
$RC_\tau^m(\overline{x}) = true$, $ST_\tau^m(\overline{x}) = \overline{x}$.

Finally, $RC_\tau = RC_\tau^1$ and $ST_\tau = ST_\tau^1$. $RC_\tau^m(\overline{x}) = true$, $ST_\tau^m(\overline{x}) = \overline{x}$. Notice that the path $v_1 \rightarrow \ldots \rightarrow v_m$ describes a computation that executes the commands labeling the nodes $v_1, \ldots, v_{m-1}$ but does not execute the command at v_m. This is the reason that RC_τ and ST_τ do not depend on the command at v_m.

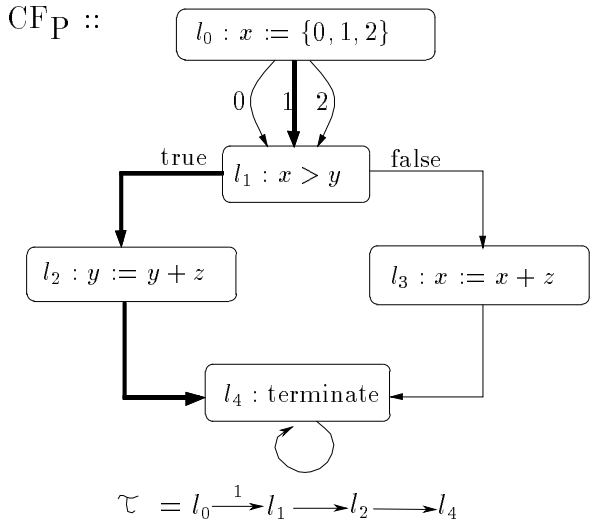
In Figure 3.8, RC_τ and ST_τ are calculated for a given elementary path (marked by bold edges). In this small example the specification does not refer to either of the variables x, y or z so that the only breaking points are l_0 and l_4. The result $RC_\tau = (1 > y)$ means that when the control is at the beginning of the program, there is a computation that travels along τ iff $(1 > y)$. The result $ST_\tau(x, y, z) = (1, y + z, z)$ means that if τ is traversed then $x' = 1$, $y' = y + z$ and $z' = z$ where x, y, z are the variable values at the beginning and x', y', z' are the values at the end of τ.

We can now define the transition relation R_τ created from an elementary path $\tau = v_1, \ldots, v_m$ to be: $R_\tau = \{((l, \sigma), (l', \sigma)) \mid \sigma \models RC_\tau \wedge \sigma' = ST_\tau(\sigma)\}$, where l is the program-counter location associated with v_1 and l' is the program-counter location associated with v_m.

Notice that both the selection of the breaking points and the computation of the reachability conditions and state-transformation functions is done automatically. The user is only required to supply a set of visible variables, or a set of atomic propositions which determine the set of visible variables.

Theorem 3.1 *Given a CTL*-X specification φ, the reduced structure reduced(CF_P) created in the above manner is equivalent with respect to φ to the original Kripke structure struct(CF_P) associated with the program.*

Proof: [Skeleton] From [4] we know that two Kripke structures are equivalent w.r.t CTL*-X formulas if there is a stuttering equivalence relation H s.t.: if $(s, s') \in H$ then s and s' satisfy the same atomic propositions and for every run from s in one system



$RC^4_\tau(x,y,z) = true$ $\quad ST^4_\tau(x,y,z) = (x,y,z)$

$RC^3_\tau(x,y,z) = true[y \leftarrow y+z] = true$ $\quad ST^3_\tau(x,y,z) = (x,y,z)[y \leftarrow y+z] = (x, y+z, z)$

$RC^2_\tau(x,y,z) = true \wedge (x > y) = (x > y)$ $\quad ST^2_\tau(x,y,z) = ST^3_\tau(x,y,z) = (x, y+z, z)$

$RC^1_\tau(x,y,z) = (x > y)[x \leftarrow 1] = (1 > y)$ $\quad ST^1_\tau(x,y,z) = (x, y+z, z)[x \leftarrow 1] = (1, y+z, z)$

Figure 3.8: An example of a calculation of RC_τ and ST_τ

there is a corresponding run from s' in the other system and vice-versa. Two runs are corresponding if they can be partitioned into blocks (finite series of consecutive states) s.t. every state in the ith block of one run is in the relation H with every state in the ith block of the other.

We define a relation H between states of $struct(CF_P)$ and $reduced(CF_P)$ as follows. Every state in $struct(CF_P)$ is of the form (l, σ) where l is a program location (a node in CF_P) and $\sigma \in \Sigma$. The states of $reduced(CF_P)$ are of the same form, only the locations are all in BP. For every location $l \in BP$ and $\sigma \in \Sigma$ we set $((l,\sigma),(l,\sigma)) \in H$. For every state s of $struct(CF_P)$ in which $l \notin BP$ we look at the possible runs from s. We observe that the only states that have more than one successor are states with locations that correspond to a non-deterministic assignment, which is a breaking point. If l is a boolean condition (of an "if" or "while" command) then (l,σ) has only one successor, depending on whether $\sigma \models B$ or $\sigma \not\models B$. Therefore, if the location of s is not a breaking point then there is a single run from s to another state s' with a location which is a breaking point (without passing any other breaking points on the way). Since the location of s' is a breaking point, s' is a state of $reduced(CF_P)$, and we define that $(s,s') \in H$. The states s and s' satisfy the same atomic propositions because the path from s to s' does not include any statements that may change visible variables. Therefore, s and s' give the same values to all visible variables.

It can easily be shown that the above relation is indeed a stuttering equivalence relation. Figure 3.9 shows how every state in $reduced(CF_P)$ corresponds to a block of states in $struct(CF_P)$. Notice that in every block, the only transition that may influence the specification is the first one, all other transitions are guaranteed not to change values of visible variables and not to be a branching point in the structure. Therefore the first state of a block in $struct(CF_P)$ corresponds to one state in $reduced(CF_P)$, and all the rest of the states in the block (including the last one) correspond to another state in $reduced(CF_P)$. Notice that since the first states in both structures give the values to variables (and therefore satisfy the same atomic propositions), and since the first transition is the result of the same command, we may conclude that the next state in $reduced(CF_P)$ (s_4), and the following states in $struct(CF_P)$ ($s_1, \ldots, s_4$) all satisfy the same atomic propositions.

□

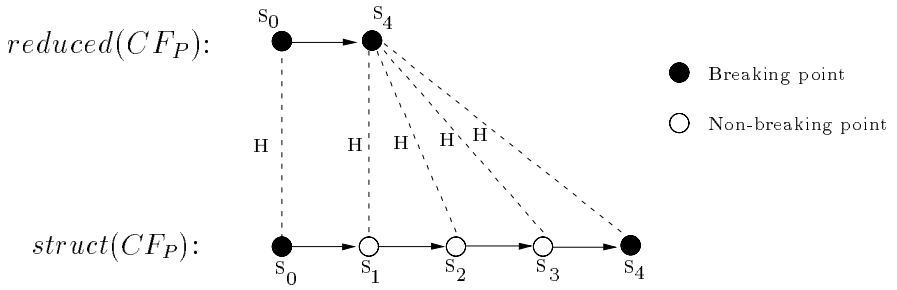


Figure 3.9: Stuttering Bisimulation between $struct(P)$ and $reduced(P)$

The above proof shows, in fact, that the reduced structure is equivalent to the original structure with respect to any formula that uses the same set of atomic propositions. This is because the only attribute of φ that was used was the set of variables to which it refers (the set of visible variables). This means that the same reduction can be used to check several formulas.

Path Reduction for Parallel Programs

The common method for handling parallel programs is to translate each process into a Kripke structure and then create the cross product. As described before, the cross product is created by taking the union of all the local transitions from all the processes, and creating a single transition out of each matching pair of communication commands. It is well known that the main source for state-space explosion in parallel programs is the cross product of several processes, since the product must include all possible interleavings of the individual processes. It is therefore desirable to reduce the sizes of the processes as much as possible before attempting to create their cross product. We propose to reduce each process in a similar fashion to the reduction of sequential processes. The only difference is that we need to handle *send* and *receive* commands.

Given a parallel program $P = [P_1||\ldots||P_n]$ we create the control-flow graph CF_i for each process P_i. We then create a reduced Kripke structure $reduced(CF_i)$ for each process. The definition of $reduced(CF_i)$ is the same as before, except for the following addition to the definition of breaking points. A node n will be in BP if

5. n is labeled by a communication statement (*send* or *receive*), or is the statement immediately following a communication.

Note that from every node of the control-flow graph which is labeled with a communication command there is a single out-going edge, pointing to the next statement to be executed. The additional breaking points make sure that any elementary path in CF_i that contains a communication command will not contain any other commands. The isolation of communications enables us to synchronize "send" and "receive" commands without involving any other local operations in the same transition. To create the reduced structure, every elementary path in the control-flow graph which is not a communication command is translated into a set of transitions (as in the reduction for a single sequential program). The transitions for communication commands are created from matching pairs of communication commands (elementary paths), as described in Section 3.1.3.

Theorem 3.2 *Given a CTL*-X formula φ, the reduced Kripke structure $reduced(P)$ created in the above manner for a parallel program P is equivalent with respect to φ to the original Kripke structure associated with P.*

As before, to prove that our reduction preserves CTL*-X specifications we define a stuttering equivalence relation H.

Proof:

We use the same relations H_i between $struct(CF_i)$ and $reduced(CF_i)$ that were defined for sequential processes. The relation H is the combination of these "local" equivalence relations, so that $((v_1,\ldots,v_n),(r_1,\ldots,r_n)) \in H \Leftrightarrow [(v_1,r_1) \in H_1 \wedge \ldots \wedge (v_n,r_n) \in H_n]$.

To show that H is a stuttering equivalence relation we need the following definition:

Definition 3.12: A transition $(l,\sigma) \rightarrow (l',\sigma')$ in $struct(CF_i)$ is called *distinct* if the location l from which it exits is a breaking point. A transition $s \rightarrow s'$ in $struct(P)$ is called *distinct* if it is a transition in which one process P_i performs a distinct transition and all other process do not change their local state, or if it is a communication.

The set of distinct transitions includes all the transitions that might influence the specification, i.e. change the value of a variable that appears in an atomic proposition or create a branching in the structure. Notice, however, that it also includes transitions that may not influence the specification, for example, when the breaking point is the head of a "while" statement.

Given a run $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ of $struct(CF)$ and a state r_1 of $reduced(CF)$ s.t. $(s_1,r_1) \in H$ we show that there exists a corresponding run π' of $reduced(CF)$ from

r_1. Every state of π is a combination of local states: $s_k = (s_k^1, \ldots, s_k^n)$. We mark the distinct transitions in π and then divide π into blocks so that each block begins after a distinct transition was executed and ends before the next distinct transition is taken (see figure 3.10)

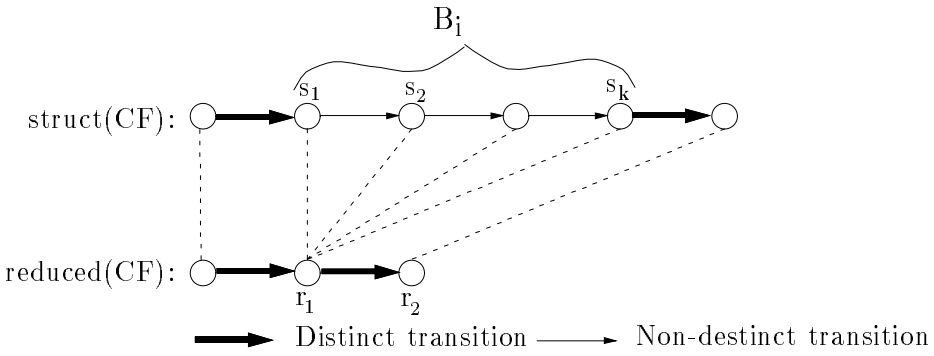


Figure 3.10: Partitioning of a run into Blocks

Lemma 3.3: Let $B = s_1, \ldots, s_k$ be a block of π, and let r_1 be a state of $reduced(CF)$ s.t. $(s_1, r_1) \in H$. Then for every $2 \leq i \leq k$, $(s_i, r_1) \in H$.

To see why this is true, notice that for every process i, if $(s_1^i, r_1^i) \in H_i$ and P_i performs a non-distinct operation, it can lead only to a state which H_i connects to r_1^i.

Lemma 3.4: Let $s = (s_1, \ldots, s_n)$ and $r = (r_1, \ldots, r_n)$ be states in $struct(CF)$ and $reduced(CF)$ respectively s.t. $(s, r) \in H$. Then for every distinct transition $s \to s'$ in $struct(CF)$ there is a corresponding transition $r \to r'$ s.t. $(s', r') \in H$.

If the transition $s \to s'$ is not due to a communication then the difference between s and s' is only in the local state of one process P_i. This transition was created from a distinct transition $s^i \to s^{i'}$ in $struct(CF_i)$. From the state $s^{i'}$ in $struct(CF_i)$ there is only one possible continuation of the execution until the next time it arrives at a state which corresponds to a breaking point. If we take the program locations along this run we can recreate an elementary path in CF_i, which was used to create a set of transitions in $reduced(CF_i)$. There is only one transition in this set that exits r^i (easy to see from the definition of R_τ), and it leads to some $r^{i'}$. By the definition of H_i we have that $(s^{i'}, r^{i'}) \in H_i$, and therefore, if we take r and change the ith element from r^i to $r^{i'}$ we get a state r' such that $r \to r'$ and $(s', r') \in H$.

If the transition $s \to s'$ is a communication then there are two processes P_i and P_j involved, and this communication is an elementary path in both of their control-flow graphs. This means that there is a transition from r into some r' corresponding to the exact same communication between these two processes, and obviously $(s', r') \in H$.

The combination of the above two lemmas allows us to repeatedly choose states in $reduced(CF)$ so that each state corresponds to a block in π, to create the desired run π'.

We have thus far shown that every path π in $struct(CF)$ has a corresponding path π' in $reduced(CF)$. For the other direction, we are given a run π' in $reduced(CF)$ and we need to create a corresponding run in $struct(CF)$. This is simple since every transition in $reduced(CF)$ was created from an elementary path in one of the graphs CF_i, or from a communication. If it is a communication then there is a corresponding transition in $struct(CF)$ and we choose it. Otherwise, we look at the path in CF_i that created the transition and execute each command belonging to each edge along the path. The states we pass on the way will create a block which is equivalent to a single state in π'. □

3.3.2 Dead Variables Reduction

Our second reduction focuses on reducing variable domains. When creating a model for a program the set of states includes all possible valuations to variables. The dead-variable reduction identifies valuations that induce equivalent computations and reduces the size of the model by choosing a representative of each equivalence class.

Instead of presenting the dead-variable reduction in full, we develop it in steps. We begin by introducing a reduction for sequential processes that utilizes *fully dead* variables. Next, we improve this reduction by considering *partially dead* variables. We complete the development of the reduction by showing how to create a reduction for parallel programs as well.

Fully Dead Variables

We say that a variable x is *used* in a statement if the statement is an assignment and x appears in the expression on the right hand side of the assignment, or if the statement is an "if" or a "while" command and x appears in the condition. We say that x is *defined* in a statement if it is the left hand side of an assignment. Notice that in the statement "$x := x + 1$" x is first used, and then it is defined.

Definition 3.13: A program variable x is said to be *dead* at a program location l if on every execution path from l, x is defined before it is used, or is never used at all.

When a variable is dead at a specific program location its value at that point is insignificant since it will not be used. This means that two states that have that location, and differ only in the value given to x, will have identical continuations. To make these states equivalent with respect to CTL* we need to make sure that the value of x does not influence the truth of atomic propositions. These conditions are summarized in theorem 3.5 bellow.

Definition 3.14: Let $\sigma, \sigma' \in \Sigma$ be two valuations to program variables. We write $\sigma \equiv_{-x} \sigma'$ if $\sigma(y) = \sigma'(y)$ for every program variable y such that $y \neq x$.

Theorem 3.5 *Let $l \in Loc$ be a program location in the process S, and x a non-visible program variable which is dead at l. For any two states $(l, \sigma_1), (l, \sigma_2)$ s.t. $\sigma_1 \equiv_{-x} \sigma_2$ it holds that $(l, \sigma_1) \equiv_{CTL*} (l, \sigma_2)$.*

To prove this theorem we use the following Lemma:

Lemma 3.6: Let x be a variable and l a location such that x is neither used nor defined by the command at l. Let $s_1 = (l, \sigma_1)$ and $s_2 = (l, \sigma_2)$ be two states such that $\sigma_1 \equiv_{-x} \sigma_2$. Then for every state (l', σ_1') such that $(l, \sigma_1) \rightarrow (l', \sigma_1')$ there exists a state (l', σ_2') such that $(l, \sigma_2) \rightarrow (l', \sigma_2')$ and $\sigma_1' \equiv_{-x} \sigma_2'$.

Proof: Since the command at l does not use or define x we know that it is either an assignment which does not involve x (on either side) or a branching command ("if" or "while") for which the condition does not involve x. The proof is according to this command.

- Assume that the command at location l is a non-deterministic assignment "$y := \{e_1, \ldots, e_n\}$". Each successor state (l', σ_1') of (l, σ_1) is a result of assigning one of the expressions e_i to y. Since x does not appear in any of these expressions, for every successor (l', σ_1') of (l, σ_1) there must be a successor (l', σ_2') of (l, σ_2) which is the result of assigning the same e_i to y. Because $\sigma_1(e_i) = \sigma_2(e_i)$ we conclude that the same value is assigned to y in both successors and therefore $\sigma_1' \equiv_{-x} \sigma_2'$.

- Assume that the command at location l is an assignment "$y := e$". This case is similar to the previous one because the assignment can be considered as a non-deterministic assignment with only one value to choose.

- Assume that the command at location l is either "if B then S_1 else S_2 fi" or "while B do S od". Since x does not appear in B we know that $\sigma_1 \models B \Leftrightarrow \sigma_2 \models B$. Also, in all successor states the values of variables do not change. Therefore the successor states of (l, σ_1) and (l, σ_2) will have the same location, and the same valuations σ_1 and σ_2 for which we know that $\sigma_1 \equiv_{-x} \sigma_2$.

This concludes the proof of the lemma. □

We can now prove theorem 3.5 by defining a bisimulation relation [49] on the states of the structures representing our program. The resulting relation will contain pairs of states which are equivalent with respect to CTL*.

Proof: Let x be a non-visible program variable. We build a relation $H = H_1 \cup H_2 \subseteq S \times S$ such that

$$H_1 = \{((l, \sigma_1), (l, \sigma_2)) \mid x \text{ is dead at } l \text{ and } \sigma_1 \equiv_{-x} \sigma_2\}$$

$$H_2 = \{((l, \sigma), (l, \sigma)) \mid x \text{ is not dead at } l\}$$

For every $(s_1, s_2) \in H$ we need to prove three things:

1. $L(s_1) = L(s_2)$
2. For every s_1' s.t. $s_1 \rightarrow s_1'$ there exists a state s_2' s.t. $s_2 \rightarrow s_2'$ and $(s_1', s_2') \in H$.
3. For every s_2' s.t. $s_2 \rightarrow s_2'$ there exists a state s_1' s.t. $s_1 \rightarrow s_1'$ and $(s_1', s_2') \in H$.

For every pair $((l,\sigma_1),(l,\sigma_2)) \in H$ it holds that $\sigma_1 \equiv_{-x} \sigma_2$ and, since x is not visible, $L(l,\sigma_1) = L(l,\sigma_2)$. It remains to prove the last two conditions. The case when $(s_1,s_2) \in H_2$ is trivial because then $s_1 = s_2$. The interesting case then is for $(s_1,s_2) \in H_1$ where $s_1 = (l,\sigma_1)$ and $s_2 = (l,\sigma_2)$. Here there are three possibilities:

- x is neither used nor defined by the command at l. By lemma 3.6, for every state (l',σ_1') s.t. $(l,\sigma_1) \rightarrow (l',\sigma_1')$ there is a state (l',σ_2') s.t. $(l,\sigma_2) \rightarrow (l',\sigma_2')$ and $\sigma_1' \equiv_{-x} \sigma_2'$. By definition, this implies that $((l',\sigma_1'),(l',\sigma_2')) \in H$. The same holds for the other direction.

- The command at l is a non-deterministic assignment to x of the set of expressions $\{e_1,\ldots,e_n\}$ such that none of the expressions depend on x (otherwise x would not be dead). For every expression e_i chosen, we know that $\sigma_1(e_i) = \sigma_2(e_i)$, and so the state created by the assignment of $\sigma_1(e_i)$ into x is a successor of both (l,σ_1) and (l,σ_2).

- Assume that the command at location l is an assignment "$y := e$", where e does not depend on x. This is similar to the previous case since a simple assignment can be viewed as a non-deterministic assignment with one possible value.

This concludes the proof of theorem 3.5. □

We now build a reduced equivalent model for a program, in which we keep only one representative of each equivalence class.

Definition 3.15: Let d be a representative value from the domain of x. Given a program P, the reduced model of P is denoted by $reduced(P)$. Recall that the Kripke structure of a process is created from the control-flow graph of P by defining a set of transitions representing every edge in the control-flow graph. For every edge $n \rightarrow n'$ such that x is dead at n' but not at n we create a transition that simultaneously performs the statement in n and an assignment of d into x. Take for example a node n labeled by the assignment "$y := exp$" and the edge $e = n \rightarrow n'$ exiting from it. If the expression exp uses x, but after this assignment x is not used again before it is defined then at n the variable x is not dead, but at n' it is. In this case the transition relation for the edge e is $R_e \stackrel{\triangle}{=} \{((l,\sigma),(l',\sigma')) \mid \sigma' = \sigma[y \leftarrow exp; x \leftarrow d]\}$ (where l is the program location of n and l' is the program location of n'). All other edges are translated into sets of transitions in the usual manner.

The reduced transition system $reduced(P)$ can be created statically (from the control flow graph of P) without building the structure $struct(P)$. The reduced structure will have less reachable states, since every equivalence class from H_1 will be represented by a single state, the one that gives x the chosen value d. Calculating the locations in which x is dead can also be done statically and efficiently, by examining the text of P. Furthermore, in order to produce a smaller model we may perform this reduction for more than one variable. For every variable we wish to use we compute the locations in which it is dead. The definition of the reduced transition system is updated accordingly.

```
l1:   if (y < 0) then
l2:         y := x;
      else
l3:         y := 0;
      fi;
l4:   x := 0;
```

Figure 3.11: An example of a partially dead variable

Partially Dead Variables

We wish to make our reduction more effective (i.e. create an even smaller reachable state-space) by taking into account more information about the possible uses of variables. We notice that in some cases, even though a variable x is not dead at a location l, there are possible computations from l on which x will not be used. For example, in figure 3.11 we see that when the control is at location l_1 the variable x is used before it is defined only if $y < 0$. For every state (l_1, σ) such that $\sigma \not\models (y < 0)$ we can be sure that on every computation that starts from (l_1, σ), x is defined before it is used. However, according to the definition of the previous subsection, x is not dead at l_1 and therefore there will be no reduction. In this subsection we show how to find such cases, and use this information.

We change our definition of "dead variables". Instead of looking at variables that are dead at a given program location, we look at variables being dead at a given state. For a given program location we will have a condition that tells us when x is dead. The method in the previous section can be viewed as a version of this new method, using only the conditions $true$ or $false$.

Definition 3.16: Let x be a program variable, l a program location, and σ a valuation for the program variables. We say that x is *dead* at the state $s = (l, \sigma)$ if on all possible runs from s the value of x is not used before it is defined (either x will not be used, or it will be defined before the first time it is used).

This definition is similar to the definition of x being dead at a program location, except that wherever we referred to a program location we now refer to a specific combination of program location and variable values. Again we find an equivalence between states that differ only on a dead variable.

Theorem 3.7 *If a non-visible variable x is dead at (l, σ), and if $\sigma \equiv_{\_x} \sigma'$, then $(l, \sigma) \equiv_{CTL*} (l, \sigma')$.*

We omit the proof of this theorem, but note that it can easily be developed from the proof of Theorem 3.5.

For the remainder of this section we assume that x is the variable according to which we want to perform our reduction.

We calculate for each program location l a boolean condition over the program variables, called dead(l), so that for every valuation σ it holds that if $\sigma \models \text{dead}(l)$ then x is dead at (l, σ). The condition we calculate is an under-approximation since the implication in the other direction might not be true (i.e. it is possible that x is dead at (l, σ) and yet $\sigma \not\models \text{dead}(l)$). We compute an under-approximation because calculating the exact condition cannot be done in a single traversal of the control-flow graph.

We restrict ourselves to handling only non-array variables, i.e. the variable x for which we want to define the condition dead(l) is not an array.

We calculate dead(l) for every program location by traversing the control-flow graph of P, bottom up. At each step, when we calculate the condition dead(l) for a sub-program, we have already calculated the conditions for its end location.

The first step is to assign a condition to the final program location l_{end} (which is the program counter location of the "terminate" command), so we define: $\text{dead}(l_{end}) = true$. We now describe how to calculate the condition for a sub-program, given that we have already calculated it for its end location.

- For the sub-program l: skip l':
  $\text{dead}(l) = \text{dead}(l')$.

- For the sub-program l: $x := exp\ l'$:
  If the expression exp does not use x then $\text{dead}(l) = true$. Otherwise, $\text{dead}(l) = false$.

- For the sub-program l: $y := exp\ l'$ $(y \neq x)$:
  If exp uses x then $\text{dead}(l) = false$. Otherwise, we change the condition $\text{dead}(l')$ according to the assignment: $\text{dead}(l) = \text{dead}(l')[y \leftarrow exp]$.

- For the sub-program l: $y := \{exp_1, \ldots, exp_n\}\ l'$:
  We add up the influences of all the possible assignments. If x is used by (at least) one of the expressions $exp_1, \ldots, exp_n$ then $\text{dead}(l) = false$. Otherwise, if $y = x$ then $\text{dead}(l) = true$. If x is neither used nor defined by the assignment then $\text{dead}(l) = \bigwedge_{j=1,\ldots,n} \text{dead}(l')[y \leftarrow exp_j]$.

- For the sub-program l: $a[exp_1] := exp_2\ l'$:
  If either exp_1 or exp_2 use x then $\text{dead}(l) = false$.
  Otherwise $\text{dead}(l) = \text{dead}(l')[a[exp_1] \leftarrow exp_2] = \text{dead}(l')[a \leftarrow (a; exp_1 : exp_2)]$.

- For the sub-program l: if B then l_1: S_1 else l_2: S_2 fi l':
  We use a recursive call to calculate the conditions $\text{dead}(l_1)$ and $\text{dead}(l_2)$, using $\text{dead}(l')$ as input for both calculations. If the condition B uses x then $\text{dead}(l) = false$. Otherwise, $\text{dead}(l) = (B \wedge \text{dead}(l_1)) \vee (\neg B \wedge \text{dead}(l_2))$.

- For the sub-program l: while B do l_1: S_1 l_1' od: l':
  Similarly to the "if" case, if B uses x then $\text{dead}(l) = false$.

  Otherwise, we use a recursive call to calculate $\text{dead}(l_1)$. The input to this call (a value for $\text{dead}(l_1')$) is the "safest" approximation we can give, since we do not have any information on what happens at the end of the body after each iteration. If x does not appear in S_1 at all, which can be checked while parsing the program, and if $\text{dead}(l') = true$ then we assume $\text{dead}(l_1') = true$. Otherwise, we have to assume $\text{dead}(l_1') = false$.

  When the recursive call for S_1 is done we define: $\text{dead}(l) = (B \wedge \text{dead}(l_1)) \vee (\neg B \wedge \text{dead}(l'))$

The above definition can be further optimized by adding traversals through the body of each loop. The important characteristic that we must maintain is that we traverse the control-flow graph of the program a constant number of times, and therefore it is more efficient than model checking on the full model of the program. Notice that the reason we need to perform approximations is the while loop. All other constructs create an exact computation of *dead*.

One optimization, which we used in our examples, is to traverse each loop twice so that we can identify situations in which x is dead at the top of the body (i.e. at location l_1), although it is used somewhere inside. Assume that $\text{dead}(l') = true$, for a while loop as in the definition above. In order to have $\text{dead}(l_1) = true$ according to the above algorithm x must be defined on every path from the beginning to the end of the loop body. Instead, we propose to first compute *dead* on S_1 under the assumption that $\text{dead}(l_1') = true$. If under this condition we find that $\text{dead}(l_1) = true$ then we can conclude that x can never be used before it is defined inside the loop. We can therefore set $\text{dead}(l) = true$. If, on the other hand, the calculation result is that $\text{dead}(l_1) \neq true$ then we have calculated our condition for S_1 under false assumptions, and the results cannot be used. We then do another round on S_1, this time with the original safe assumption: $\text{dead}(l_1') = false$.

Notice that the condition $\text{dead}(l)$ can never be dependent on x. When it depends on a variable y it is because during its calculation we passed a statement that evaluates an expression involving y. However, a statement that evaluates an expression that depends on x is a use of x, after which we have $\text{dead}(l) = false$.

We can now define a reduced transition system according to (partially) dead-variable reduction:

Definition 3.17: Given a program P and a choice of a non-visible variable x, compute $\text{dead}(l)$ for all the locations in P. Define $reduced(P)$ so that every edge $e = n \rightarrow n'$ in the control-flow graph is translated into a reduced transition relation according to the information calculated. Let l be the program counter location of n, and l' the location of n'. If x is used by the statement at l, than there is no change in the definition of R_e. Otherwise, we change the definition of R_e from subsection 3.1.3 so that for states that satisfy $\text{dead}(l)$ we add the assignment $x := d$. For example, for an assignment y

$:= exp$ we define: $R_e \equiv y' = exp \wedge ((\text{dead}(l) \wedge x' = d) \vee (\neg\text{dead}(l) \wedge x' = x)) \wedge \ldots$ (the continuation states that all other variables do not change value, and that the program counter advances according to the appropriate labels). The definition of R_e for other commands is updated in a similar manner.

In the reduced structure some of the transitions have been redirected so that many states become unreachable. These states, and perhaps all of their descendants, will not be traversed when performing model-checking. In effect - we have pruned parts of the state-space. To show that the pruned structure is CTL* equivalent to the original structure we build a new bisimulation relation.

Definition 3.18: $H = H_1 \cup H_2 \subseteq S \times S$ such that:

$$H_1 = \{((l, \sigma_1), (l, \sigma_2)) \mid \sigma_1 \models \text{dead}(l) \text{ and } \sigma_1 \equiv_{-x} \sigma_2\}$$

$$H_2 = \{((l, \sigma), (l, \sigma)) \mid \sigma \not\models \text{dead}(l)\}$$

It is easy to see that this is truly a bisimulation relation.

Again we notice that the reduced structure is equivalent to the original structure with respect to any formula over the same set of visible variables, so the same reduction can be used to check several formulas.

From here on, when referring to dead-variable reduction, we are referring to the reduction according to partially dead-variables.

Dead-Variable Reduction for Parallel Programs

The dead-variable reduction for parallel programs proceeds in a similar manner to the path reduction. We first reduce each process separately and then create the cross product of the reduced models. However, in order to perform the dead-variable reduction on a single process we have to augment the computation of dead(l) with instructions for handling communication commands:

- For the sub-program $\underline{l\text{: send}(P_i,exp)\ l'}$:
  If exp uses x then dead$(l) = false$. Otherwise, dead(l) = dead(l').

- For the sub-program $\underline{l\text{: receive}(P_i,x)\ l'}$:
  dead$(l) = true$.

- For the sub-program $\underline{l\text{: receive}(P_i,y)\ l'}$ $(y \neq x)$:
  Since any value may be assigned to y by this operation, to be sure that dead(l') is true after executing the command we must require that it will be true for all possible values assigned to y: dead$(l) = \forall y.\text{dead}(l')$.

This allows us to compute the condition dead(l) for every location of a process and for each process separately.

Theorem 3.8 *Let $P = [P_1 || \ldots || P_n]$ be a parallel program. Then the parallel composition of the dead-variable reduced structures for the processes is bisimilar to the parallel*

composition of the original structures. Formally, if $struct(P) = struct(P_1) \,||\, \ldots \,||\, struct(P_n)$ *and* $reduced(P) = reduced(P_1)||\ldots||reduced(P_n)$*, then* $struct(P) \equiv_{CTL*} reduced(P)$*.*

Proof: Let H_i be the bisimulation relation from definition 3.18 between $struct(P_i)$ and $reduced(P_i)$. Every state s in $struct(P)$ is a tuple $(s_1, \ldots, s_n)$ such that s_i is a state of $struct(P_i)$, and similarly every state t in $reduced(P)$ is a tuple $(t_1, \ldots, t_n)$. We define a relation H such that $(s,t) \in H$ iff for every i, $(s_i, t_i) \in H_i$. It is easy to see that since every H_i is a bisimulation relation, H is also a bisimulation relation. □

3.3.3 Integration with verification techniques

Our methods for reducing the state-space of programs are carried out according to the syntax of the program, creating a reduced model of the program. Given a program P it is possible to apply one or both of the reductions. Furthermore, when applying the dead-variable reduction we can do so for more than one variable. Once our methods have been applied, one can choose any verification method to be used on the result. We now describe in more detail how our reduction methods can be incorporated into several well known verification techniques, and also with our own modular model checking method.

A reduced Kripke structure can be used for *state exploration methods*. These are methods that traverse the state space of the Kripke structure on-the-fly, usually by means of a DFS algorithm. Examples of verification tools that use such methods are *Murphi* [46] and *SPIN* [32]. The space consumption of an algorithm based on a DFS traversal is proportional to the maximal simple path from an initial state (which is the maximal depth of the stack during the search). The time complexity is linear in the number of reachable states. We notice that when creating a reduced Kripke structure we do not have to create the set of program states. We can create only the transition relation, which may be represented by a first order formula (or a set of formulas). This formula is used as a next-state function that given a state produces the set of successors of that state. In the case of path-reduction the maximal simple path (a run that does not go through the same state twice) is shorter and therefore the DFS requires less space. In both reductions the reduced Kripke structure has less reachable states and so the time needed for the DFS is also reduced.

Partial order reductions [50, 56, 55] are methods of reducing the state-space traversed by a state-exploration verification algorithm, and are implemented in state-space exploration tools such as *SPIN*. In general, a partial order reduction method defines for every state a subset of the transitions exiting the state that will be traversed. This subset is chosen in a way that ensures that although parts of the state-space will not be visited, the result of the search will not change. Our path reduction can be compared to partial order reduction methods. Path reduction creates a program which is still a parallel composition of processes, only that each transition of a reduced process may represent a series of transitions of the original process. Therefore some of

the possible interleavings between processes are restricted beforehand. This reduction might not be achieved by partial order reduction methods, but the reduced transition system that path reduction creates may still include several possible interleavings that can be pruned by a partial order reduction method. There is no need to make changes in our method or in the partial order reduction method in order to combine the two. Note also that since some partial order methods require an initial DFS of the system in order to perform calculations, it is an advantage that the system that these methods get to work on is smaller than the original one.

Our reduction methods can easily be combined with symbolic methods. A reduced Kripke structure can be translated directly into a symbolic representation in the same way the original system would be. In fact, the computation of R_τ and T_τ for path reduction and the computation of dead(l) for dead-variable reduction can be done symbolically, to produce a BDD representation of the resulting transition relation. Since BDDs are representations of formulas, and operations on formulas (such as conjunctions, disjunctions and quantification) are efficient BDD operations, R_τ, T_τ, and dead(l) can be created according to the boolean formulas that define them by a series of BDD operations. The transition relation of the whole Kripke structure is produced by a boolean combination of the BDDs representing each edge.

The size of BDDs is difficult to predict. However, it depends heavily on the number of bits used to represent each state. Path reduction reduces the number of program counter locations, and therefore reduces the number of bits needed for the representation of a single state. For dead-variable reduction it might be preferable to replace the assignment of a single chosen value d into x with a non-deterministic assignment that allows x to have any value. In this way, x might be quantified out of the transition relation (or parts of it). It also appears that in some cases it would be better to use the fully-dead version of reduction (subsection 3.3.2) so as not to introduce dependencies between variables.

Some verification methods which consider weaker specification languages than CTL* use a notion of *fairness* to enhance the expressibility of the language [14, 6, 41]. A special condition determines which computations are fair, and only those computations are confronted with the specification. The fairness condition is usually given in terms of a formula in the same specification language or a weaker one, and a computation is fair if this condition is true in infinitely many states along the computation. In order to use such methods with our reduction we have to consider the fairness condition in the same way we consider the specification. The only change is that variables that appear in the fairness condition will also be considered visible. Assuming that the fairness condition is in the language that the reduction preserves (CTL* or CTL*-X), every state in the original system has an equivalent state in the reduced one (w.r.t the fairness condition), and for every fair computation in the original system the corresponding computation in the reduced system is also fair.

To combine our reductions with the modular model checking algorithm we presented before, we first perform all the calculations needed for the reductions. For dead-variable reduction this means that we calculate dead(l) for every program counter

location l. For path reduction, we add the entry and exit location of every node in the partition graph as a breaking point, so that elementary paths do not span more than one node in the partition graph. We then calculate the reachability condition and state transformation function as usual. When creating the Kripke structure for each partition graph node, we create the reduced structures, instead of the original ones. The modular model checking algorithm need not be aware that these are not the original structures associated with nodes.

3.3.4 Experimental Results

In order to evaluate our reductions we chose several examples and translated them into Murphi code. The Murphi language [46] consists of a list of rules, where each rule has a guarding condition and a body. A rule is executable if its guard evaluates to true. The semantics of Murphi programs consists of a loop in which the set of executable rules is computed, one executable rule is chosen non-deterministically and then its body is executed. This process continues until (if ever) there are no executable rules. We used Murphi to perform a traversal of the state-space of each example, at the end of which we get the number of states and the number of edges in the reachable state-space of the model. Each example was manually translated into a Murphi program that represents its semantics ($struct(P)$). We then performed our reductions using the original program text (the control-flow graph) and created a new Murphi program that represents the reduced model for that example ($reduced(P)$). For each example we performed path reduction, dead-variable reduction, and then a combination of both reductions. All the examples were run with a 350M hash-table. Table 3.2 summarizes the results we obtained using our reductions. For each example we give the number of states and edges in the model, and the time it took Murphi to complete the traversal. Lines in the table that say 'failed' signify examples in which the hash-table was filled before the whole structure was traversed. The numbers in parenthesis show the relative size of the reduced model with respect to the non-reduced model. In figure 3.12 we give a block diagram summarizing the results for which the non-reduced example did not fail. The reduced models are given as a percentage of the non-reduced model.

The first example, *slide*, is a program that simulates the sliding window communication protocol between a sender and a receiver. The results in the table were obtained for an example in which both the receiver and the sender windows were of length 2. The variables chosen for the dead-variable reduction were temporary variables used to store incoming messages. Path-reduction was done with respect to a specification that states that the sender does not advance its window before the receiver has received the first message in the sender's window.

The second example, called *linked-list*, is an example of a sorting algorithm that uses a distributed linked-list of processes. It consists of several processes connected in a row so that each of them keeps a number. When a process receives a new number from the process on its right, it compares it with the number it already has. It keeps the larger of the two and sends the smaller to the next process in the line. The

example includes also a main process that inputs a list of numbers and sends them to the first in line (input is simulated by non-deterministic assignment). The dead-variable reduction was done with respect to a temporary variable that each node keeps whenever it holds two numbers (before it sends one of them to the next node in the line). The path reduction used a specification that states that whenever a node is expecting a number from the preceding node, the preceding node is about to send a number (thus assuring us that the processes will never block each other). We ran this example using 4 and 5 nodes in the list. For the case of 5 nodes the full model and the dead-variable reduced model were too large to handle. However, after performing path reduction the resulting reduced model was small enough for the traversal to terminate, and combining both reductions resulted in an even smaller model.

The third example, *find-max*, is an algorithm for finding extrema on a unidirectional ring of processes, presented in [21]. Each process is assigned a (unique) number and together they find the maximum of these numbers. For a program with n processes, the numbers $1, \ldots, n$ are assigned to processes non-deterministically, so that the process that gets the number n is the one with the maximum. The variables used for the dead-variable reduction were temporary variables used to hold incoming messages. For this example we created path-reduced models according to two different specifications. The algorithm works in phases, and during each phase each process may either be active or not. In the last phase there is only one active process, and it holds the maximum number. The first specification makes sure that when a process receives a message from phase i, the sending process is also in phase i. The second specification states that in any given phase, no two active processes hold the same number.

The last example *simple*, is taken from [3]. This work is closely related to our dead-variable reduction, and the differences and similarities will be discussed in Chapter 6. The simple example involves two processes where the first process repeatedly sends *request* or *switch* messages and the second receives them. Each *request* message is accompanied by a number, chosen non-deterministically. The second process has a state in which a *request* message results in outputting the number received, and another state in which the number is ignored. Every time the second process receives a *switch* message it switches between these two states. In [3] the example is given in the form of two state machines, the first having two states and the second three. We translated it into two processes in our programming language, with the state kept in a program variable. This introduced more states in the Kripke structure of the example because of the introduction of a program counter. Obviously, when the number sent with a *request* message is going to be ignored, the variable that holds this number is dead. Since the program does nothing else than receive the number and output it, the size of the domain of this variable is the main factor in the size of the state-space, and hence the significant effect of dead-variable reduction. This is also the reason why, in this example, path-reduction is not so effective as in other examples.

It is clear that dead-variable reduction in itself does not produce significant reduction. This can be a result of the nature of our examples. We expect this reduction to be useful on programs that perform several tasks, where one task requires many

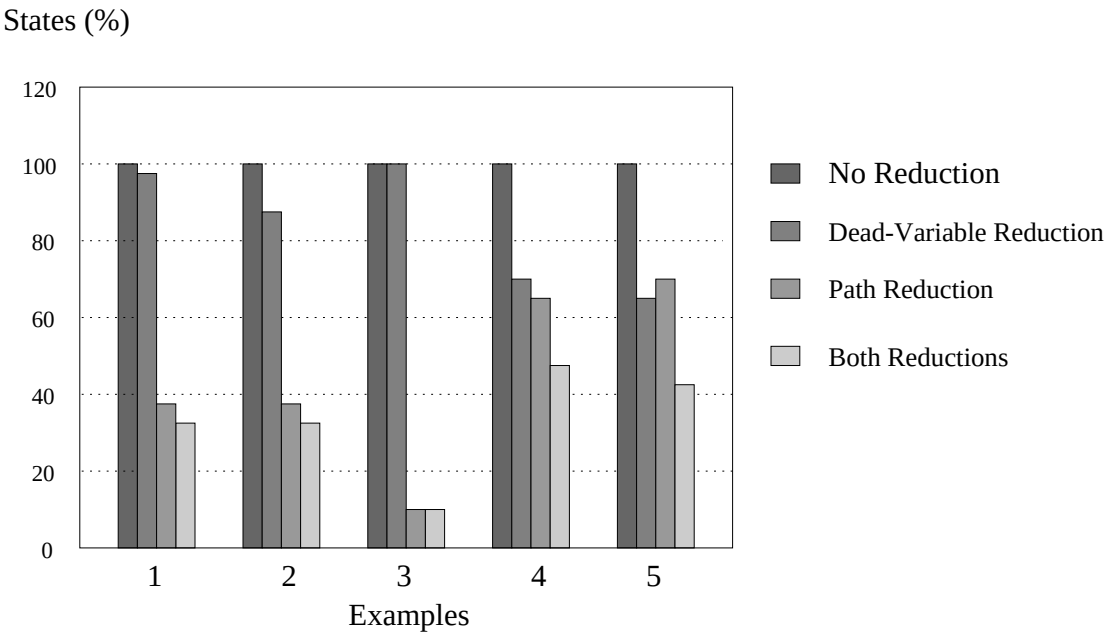


Figure 3.12: Diagram of reduction results

variables that are not needed for other tasks. This situation can occur, for example, when the program does some numerical computation and then goes on to use only the result of the computation. The variables that were used during the computation are now dead. Another situation in which dead-variables reduction can be expected to perform well is when applied to a program that was created by some automatic code generation tool or by an automatic translation from another language. Automatically generated codes tend to include many redundant variables. Our examples, being hand-produced demonstrative examples, are relatively simple, and perform no intermediate computation. It remains to be seen whether real-life examples exhibit similar behavior or not.

Path reduction, in contrast, gives significant reduction in the size of the models, and as a result also reduces computation times. Using path reduction we get reduced models which are between 8% to 37% of the original model. The explanation of this is that a sequential program (or process) performs only one operation at each step - either evaluating a condition, or assigning into a single variable. By condensing all the operations a program does between two observable points into a single step, we create a much more concise model. In the last example, we used two different specifications to create the path-reduced model. The second specification refers to more variables than the first and therefore creates more breaking points. This explains why the reduced model according to the first specification is smaller than the reduced model according to the second specification. Notice that in two cases (linked-list with 5 nodes and find-max with 5 nodes) we could not traverse the non-reduced model since it was too large. However, after using path-reduction we got a model which we could traverse

in full. In these cases we cannot tell the ratio of reduction, since we do not know the size of the non-reduced model, but we see that it is significant enough to allow us to perform model-checking on programs that are otherwise too large to handle. In the case of the find-max example, we could only handle 4 processes in the non-reduced version, whereas using path-reduction we successfully completed the traversal of the model with 8 processes.

When combining both reductions we get an even smaller model in all of the examples. Even though the added efficiency due to dead-variable reduction is not large, it seems useful to use since it is practically 'for free'.

Example	Break-down	Options lr	Options reo	Time (h:m:s)	Memory (K)	Min/Max module size
Learn-Monom	None	-	-	0:03:46	2600	1450/1450
		-	+	0:01:56	1864	15058/15058
		+	-	0:05:00	3625	20860/20860
		+	+	0:02:11	1420	4787/8168
	Moderate	-	-	0:01:49	1352	3024/10661
		-	+	0:01:06	968	3150/10968
		+	-	0:02:25	2025	4495/15502
		+	+	0:01:22	1160	4648/15788
	Extensive	-	-	0:00:36	1192	2911/5357
		-	+	0:50:00	1128	3016/5450
		+	-	0:00:49	1450	4495/5165
		+	+	0:01:18	1545	4787/8168
Stop and Wait	None	-	-	6:46:18	478787	83440/83440
		-	+	7:04:06	6560	36145/36145
		+	-	6:21:51	10975	198592/198592
		+	+	6:35:50	4475	75249/75249
	Moderate	-	-	5:51:22	8130	4908/45129
		-	+	5:58:28	6050	4986/22226
		+	-	10:17:00	13650	4078/120447
		+	+	4:09:43	9225	4727/37508
	Extensive	-	-	47:12:55	29200	695/19890
		-	+	8:40:44	7800	695/8430
		+	-	109:15:03	65000	695/52559
		+	+	8:24:52	12050	695/13023
Sort	None	-	+	12:05:04	61500	109285/109285
		+	+	3:33:31	19100	205829/205829
	Moderate	-	+	8:32:54	73500	839/103324
		+	+	4:13:22	18300	875/205197
	Extensive	-	+	6:24:38	8150	842/49058
		+	+	11:45:27	9500	830/19718

Table 3.1: Results for Modular Model Checking

Example	Reduction used	No. of States	No. of Edges	Time (h:m:s)
slide				
		2895333	6158396	0:6:35
	dead-vars	(98.6%) 2854863	(98.8%) 6085068	(84.3%) 0:5:33
	path	(36.6%) 1061947	(37.3%) 2302118	(23.5%) 0:1:33
	both	(32.1%) 929967	(33.3%) 2051234	(15.9%) 0:1:23
linked-list				
4 nodes		1203536	4147621	0:13:28
	dead-vars	(87.6%) 1054448	(88.7%) 3679149	(86%) 0:11:35
	path	(37/1%) 446849	(35.3%) 1466866	(21.4%) 0:2:53
	both	(32.6%) 392849	(31.4%) 1304450	(19.9%) 0:2:41
5 nodes		failed	-	-
	dead-vars	failed	-	-
	path	12463025	47998708	1:55:55
	both	10213613	39767184	1:38:06
find-max				
4 procs		7450357	25373736	1:38:47
	dead-vars	(100%) 7450357	(100%) 25373736	(100%) 1:38:47
	path (spec1)	(8%) 598913	(6.9%) 1774912	(4.3%) 0:4:15
	path (spec2)	(15.6%) 1168793	(14.4%) 3659832	(11.3%) 0:11:10
5 procs		failed	-	-
	path (spec1)	534532	1979482	0:5:41
	path (spec2)	1280152	5040509	0:15:38
8 procs	path (spec1)	2517233	11205360	0:39:38
	path (spec2)	4560483	20309124	1:24:23
simple				
		2270000	4767582	0:1:44
	dead-vars	(72.3%) 1641400	(74.8%) 3570746	(75.9%) 0:1:19
	path	(67.1%) 1524686	(74.8%) 3566299	(66.3%) 0:1:09
	both	(47.8%) 1085746	(54.7%) 2608879	(50%) 0:0:52

Table 3.2: Results for static analysis reductions

Chapter 4

Exploiting Structure in Hardware Verification

4.1 The Structure of Hardware Designs

Before we present our results in model checking of hardware designs, we discuss the translation of a hardware design into a Kripke structure representing its behavior. This discussion is important since it gives insight into the special structure of Kripke structures that represent hardware designs. This structure will be used in the following sections.

There are many levels of abstraction in which a hardware design can be described, from a very abstract plan of modules, to the layout which is the final goal. We examine hardware designs at the level in which logical behavior is examined, but physical behavior is not considered. This level of description is sometimes referred to as a Register Transfer Level (RTL) model. It consists of memory components (flip-flops) and logical components. The value of a flip-flop bit is determined by a boolean function which is made up of basic logic elements(such as AND, OR, NOT etc.). The parameters of this function can be other memory bits, input signals, or even the previous value of the bit being determined. In synchronous circuits, all the flip-flops change their value synchronously at the beginning of each clock cycle. In asynchronous circuits each component reacts to a change in its inputs and after a given amount of time may update its output. In this work we focus on verification of synchronous hardware designs.

A state of a hardware model is composed of a value for each flip-flop. Throughout this section we use V for the set of flip-flop variables in the design, and I for input signals. All circuit variables are boolean. A state in the model for the design is an assignment to the variables in $I \cup V$.

In this chapter we also use boolean formulas to represent sets of states and transition relations. Sets of states are represented by formulas with free variables (I, V) and transition relations are represented by formulas with free variables (I, V, I', V'). We use the same letter for a set of states (or transition relation) and the formula that

represents it.

The Kripke structure modeling a synchronous circuit is defined as: $M = (S, R, I)$, where

- The set of states S is the set of all assignments for $I \cup V$.
- $Init \subseteq S$ is the set of initial states.
- The transition relation R is defined by a set of functions $N_1, \ldots, N_l$, each defining the next-state value for a single variable $v_i \in V$ by $v_i' = N_i(I, V)$. The *global* transition relation is: $R = \bigwedge_{v_i \in V}(v_i' = N_i(I, V))$. We assume that every N_i is a deterministic total function. Note that the inputs are unrestricted.

4.2 Test sequence generation for synchronous circuits

Given a relatively small critical sub-circuit Sub, with a set of inputs I_{sub}, the designer can construct a set of test sequences that guarantees a good coverage of Sub in a relatively straightforward way, if the sub-circuit is sufficiently small. However, in general the inputs to Sub are not accessible from the exterior of the full circuit. Therefore, it is necessary to produce a set of test sequences for the inputs I of the full circuit that will induce the required set of test sequences on I_{sub}. We present a *test generation* algorithm that given a test sequence over a small sub-circuit produces a test sequence for the full circuit that reproduces the original test sequence.

Before we present the algorithm we require several definitions.

Definition 4.1:

- A *trace* of M is a sequence $\Pi = s_0, s_1, \ldots, s_n$ such that $(s_i, s_{i+1}) \in R$.
- Let $U \subseteq (I \cup V)$ be a set of variables. A *partial assignment* with respect to U is an assignment that gives values only to variables in U (as opposed to a state, which is an assignment that gives values to all the variables of the circuit). A partial assignment σ with respect to U represents the set of states s that *agree* with σ on U, i.e. for every $v \in U$, $s(v) = \sigma(v)$.
- The *projection* of a state s on U is a partial assignment over U that agrees with s. The projection of a trace Π on U is the trace obtained by taking the projection of each state in Π on U.
- Given a partial assignment σ with respect to U, an *expansion* of σ to U' (s.t. $U \subset U'$) is a partial assignment σ' over U' that agrees with σ on U.
- A *test sequence* over U is a series of partial assignments over U.

Formally, our problem is defined as follows. Given a set $I_{sub} \subseteq I \cup V$ of variables, which are the inputs to the critical sub-circuit Sub, and a test sequence $\Pi_{sub} = t_0, \ldots, t_n$ over I_{sub}, we must produce an initial state s_{init} and a test sequence $\Pi_{in} = in_m, \ldots, in_0, \ldots, in_n$ $(m \leq 0)$ where every in_i is a partial assignment over I. Let $\Pi = s_m, \ldots, s_0, \ldots, s_n$ be the trace generated by Π_{in} and s_{init} $(s_m = s_{init})$ [1]. We require that the projection of $s_0, \ldots, s_n$ on I_{sub} will be identical to Π_{sub}.

The algorithm has two stages. The first is a backward search that creates a series of sets of states. Any computation path that goes through these sets (starting at an initial state) is a solution to our problem. Given $\Pi_{sub} = t_0, \ldots, t_n$ we construct a series of sets of states $A_m, \ldots, A_0, \ldots, A_n$ in reverse order, i.e. we start by computing the set A_n and end with A_m. For every $m \leq i \leq (n-1)$, every state $s \in A_i$ has a successor in A_{i+1} (see figure 4.1). Using a slight abuse of notation, we view every t_i in the test sequence as a set of states, namely, the set of states (assignments to all variables) that agree with t_i. When creating the sets $A_0, \ldots, A_n$ we make sure that $A_i \subseteq t_i$. Thus, $A_0, \ldots, A_n$ represent the set of traces that agree with Π_{sub}. In order to make sure that a trace that runs through these sets can be created starting at an initial state, we continue to compute A_{-1}, A_{-2} and so on, until we arrive at a set A_m in which there is an initial state. If such an initial state can be found, we know that there is a trace Π that, from some point on, reproduces the test sequence Π_{sub}. If, however, we arrive at a set $A_i = \emptyset$ or $A_i \subseteq \bigcup_{j=i+1}^{0} A_j$, we can conclude that there is no input sequence Π_{in} that can be used from an initial state to reproduce Π. This means that when the sub-circuit is run within this design, the test sequence $t_0, \ldots, t_n$ can never appear at its inputs, and we report this.

In the second stage we traverse the sets, from A_m up to A_n and find one suitable trace $\Pi = s_m, \ldots, s_n$. The test sequence Π_{in} that generates this trace is created by taking the projections of the states along Π on I. The output of the algorithm is an initial state s_m, and a sequence of inputs $in_m, \ldots, in_n$.

The algorithm below uses the following functions and operators:

- The operator $Pred$ computes the set of predecessors of a set of states A according to the transition relation R. It is defined by: $Pred(A) = \exists I', V'.R(I, V, I', V') \wedge A(I', V')$ [44].
- The function $choose(A)$ receives a set A and produces a single state (assignment to $I \cup V$) $s \in A$.
- The operator $Succ(s)$ returns the set of successors of a state according to R.

Stage I:

1: $A_n := t_n$
2: for $j = n - 1$ downto 0 do

[1] Notice that the transition relation R defines the next-state value for variables in V. By choosing a next-state value for the inputs we deterministically define a successor state. Therefore, s_{init} and Π_{in} (together with R) uniquely determine Π.

3: $A_j := t_j \cap Pred(A_{j+1})$
4: endfor
5: $All := \emptyset$
6: while $(A_j \neq \emptyset) \wedge (A_j \cap Init = \emptyset) \wedge (A_j \not\subseteq All)$ do
7: $All := All \cup A_j$
8: $j := j - 1$
9: $A_j := Pred(A_{j+1})$
10: endwhile
11: $m := j$

Stage II:
12: if $(A_m = \emptyset) \vee (A_m \subseteq All)$ then
13: print "Sequence cannot be generated"
14: else
15: $s_m := choose(A_m \cap Init)$;
16: $in_m :=$ the projection of s_m on I;
17: for $j = m + 1$ to n do
18: $s_j := choose(Succ(s_{j-1}) \cap A_j)$;
19: $in_j :=$ the projection of s_j on I;
20: endfor
21: endif

Figure 4.1: The sets produced by the test sequence generation algorithm

The correctness of our algorithm is asserted using three theorems. It is easy to see that if stage I completes successfully, i.e. the algorithm will not report failure in line 13, then stage II creates a sequence of inputs $in_m, \ldots, in_n$ that create a trace $s_m, \ldots, s_n$ that goes through the sets A_i found in stage I. We want to show that this trace is in fact a solution to our problem.

Theorem 4.1: Every trace $\pi = s_m, \ldots, s_n$ such that for every $m \leq i \leq n$, $s_i \in A_i$ recreates the given test-sequence $t_0, \ldots, t_n$ on the signals in I_{sub}.

Proof: From line 3 in the algorithm it is clear that $s_i \in t_i$ for every $0 \leq i \leq n$. Therefore, if $s_m, \ldots, s_n$ is in fact a trace through the sets A_i, then the suffix $s_0, \ldots, s_n$ induces $t_0, \ldots, t_n$ on the signals in I_{sub}. □

We next show that there exists a trace through the sets A_i from an initial state. We do this using a slightly stronger claim.

Theorem 4.2: For every $m \leq i \leq n$ and every state $s_i \in A_i$ there exists a trace $s_i, \ldots, s_n$ such that for every $i \leq j \leq n$, $s_j \in A_j$.

Proof: The proof is by induction on the length of the path $s_i, \ldots, s_n$ (i.e. induction on $n - i$). The base case for $i = n$ is obvious. For the induction step we assume that from every state s_{i+1} there is a trace through $A_{i+1}, \ldots, A_n$. We prove the claim for A_i. Let s_i be some state in A_i. If $i \geq 0$ then A_i was created in line 3, and if $i < 0$ then A_i was created in line 9. In both cases we see that $s_i \in Pred(A_{i+1})$. This means that there is a state $s_{i+1} \in A_{i+1}$ such that $R(s_i, s_{i+1})$. From our assumption on A_{i+1} there must be a trace $s_{i+1}, \ldots, s_n$ through $A_{i+1}, \ldots, A_n$, and so there exists the trace $s_i, s_{i+1}, \ldots, s_n$ through $A_i, A_{i+1}, \ldots, A_n$, which proves the claim. □

The combination of the two theorems above shows that when the algorithm finishes successfully it produces a correct result. We now show that when the algorithm fails, it is also a correct result.

Theorem 4.3: If the algorithm claims that the sequence generation failed, then there is no trace from an initial state that reproduces $t_0, \ldots, t_n$ on the signals in I_{sub}.

Proof: We prove this theorem by showing that if there exists a trace from an initial state that reproduces $t_0, \ldots, t_n$, then the algorithm will **not** fail. Assume there is a trace $s_m, \ldots, s_0, \ldots, s_n$ such that for every $0 \leq i \leq n$, $s_i \in t_i$, and $s_m \in Init$. Assume also that this is the shortest trace that reproduces the given test sequence, i.e. $s_m, \ldots, s_0$ is the shortest trace from an initial state that reproduces $t_0, \ldots, t_n$. We show that for every A_j that the algorithm computes $s_j \in A_j$, and for $j < 0$ also $s_j \notin \bigcup_{i=j+1}^{0} A_i$. This will prove that the while loop in line 6 will terminate only when $A_j \cap Init \neq \emptyset$, and the algorithm will not report failure. We prove this by induction on $n - j$. For $j = n$ we know that $A_n = t_n$ and also $s_n \in t_n$ so obviously $s_n \in A_n$. Now we assume that the sets $A_{j+1}, \ldots, A_n$ were already computed and for each A_i we have $s_i \in A_i$. If $j \geq 0$ then the set A_j is computed by $A_j = t_j \cap Pred(A_{j+1})$ (line 3). Since $s_{j+1} \in A_{j+1}$ we have $s_j \in Pred(A_{j+1})$, and s_j was chosen so that $s_j \in t_j$, so we conclude that $s_j \in A_j$. If $i < 0$ then $A_j = Pred(A_{j+1})$ (line 9) and by similar reasoning we have that $s_j \in A_j$. For $j < 0$ we need also to show that $s_j \notin \bigcup_{i=j+1}^{0} A_i$. Assume to the contrary that there exists a set A_k, $j < k \leq 0$, such that $s_j \in A_k$. This means, as we have already proven, that there is a trace from s_j that goes through the sets $A_k, \ldots, A_0, \ldots, A_n$ and reproduces $t_0, \ldots, t_n$. We know that there is a trace $s_m, \ldots, s_j$ that leads from an initial state to s_j, and so we conclude that $s_m, \ldots, s_{j-1}, s_k, \ldots, s_n$ is a trace from an initial state that reproduces $t_0, \ldots, t_n$ and is shorter than the trace $s_m, \ldots, s_n$ that we started with, in contradiction to the fact that $s_m, \ldots, s_n$ was chosen to be the shortest such trace. □

4.3 Dynamic Transition Relations

The algorithm presented in the previous section might not be practical for very large circuits. In such circuits, the transition relation R is too big even if it is represented by a BDD and the operator $Pred$ becomes too expensive. This problem is not unique to our test-generation algorithm. Other algorithms, such as model checking, are also heavily dependent on the $Pred$ operator and will fail on large circuits.

To alleviate this problem we exploit the partitioning of the transition relation into functions N_i that define the next state variables v_i'. Recall that a state of our model gives values to all the variables in $I \cup V$.

Definition 4.2: A set of states A *is independent* of a variable v_i, if for every state s in A, the state that differs from s only on v_i is also in A. A formula f is independent of a variable v_i if for every two assignments σ and σ' that differ only on v_i, $\sigma \models f$ iff $\sigma' \models f$ [2].

We show that when A is independent of a variable v_i, the function N_i (which determines the value of v_i in the next state) can be omitted from the transition relation used in the computation of $Pred(A)$.

Definition 4.3: Let f be a formula (representing a set of states or a transition relation). The *support* of f is the set of variables on which f depends. Also, define $sup(f) = \{v \in (V \cup I) \mid f \text{ depends on } v\}$ and $sup'(f) = \{v \in (V \cup I) \mid f \text{ depends on } v'\}$. The set $sup(f)$ is the set of current-state variables that f depends on and $sup'(f)$ is the set of next-state variables that f depends on.

We define a dynamic version of $Pred$, called $Pred_D$. The operator $Pred_D(A)$ computes the set of predecessors of states in A according to the *partial transition relation* $\bigwedge_{v_i \in sup(A)}[N_i(s) = v_i']$, which is a transition relation that includes N_i if and only if A depends on v_i. The operator $Pred_D$ is formally defined as: $Pred_D(A) = \exists I', V'[A(I', V') \wedge \bigwedge_{v_i \in sup(A)}[N_i(s) = v_i']]$

Lemma 4.1: For any set A, $Pred(A) = Pred_D(A)$.

Proof: Assume that the variable v_i does not appear in the support of A. We start with the definition of $Pred$:

$$\text{Pred}(A) = \exists I', V'.[(v_1' = N_1(I,V)) \wedge \ldots \wedge (v_l' = N_l(I,V)) \wedge A(I',V')]$$

$$\equiv \exists I' \exists v_1', .., v_{i-1}', v_{i+1}', .., v_l'.[(v_1' = N_1(I,V)) \wedge ..(v_{i-1}' = N_{i-1}(I,V)) \wedge (v_{i+1}' = N_{i+1}(I,V)) \wedge .. (v_l' = N_l(I,V)) \wedge \exists v_i'.[(v_i' = N_i(I,V)) \wedge A(I',V')]]$$

Since $A(I,V)$ does not depend on v_i, $A(I',V')$ does not depend on v_i' and we can move it through the $\exists v_i'$ quantifier to get:

[2] Notice that a formula f is independent of v_i iff the set of states that it represents is independent of v_i.

$$\equiv \exists I' \exists v'_1, .., v'_{i-1}, v'_{i+1}, .., v'_l.[$$
$$(v'_1 = N_1(I,V)) \wedge ..(v'_{i-1} = N_{i-1}(V)) \wedge (v'_{i+1} = N_{i+1}(I,V)) \wedge ..$$
$$(v'_l = N_l(I,V)) \wedge A(I',V') \wedge \exists v'_i.[v'_i = N_i(I,V)]]$$

We assume that every N_i is a total function, i.e. for every V there exists v'_i s.t. $v'_i = N_i(I,V)$, so $\exists v'_i.[v'_i = N_i(I,V)] \equiv true$ and we get:

$$\equiv \exists I' \exists v'_1, .., v'_{i-1}, v'_{i+1}, .., v'_l.[$$
$$(v'_1 = N_1(I,V)) \wedge ..(v'_{i-1} = N_{i-1}(I,V)) \wedge (v'_{i+1} = N_{i+1}(I,V)) \wedge ..$$
$$(v'_l = N_l(I,V)) \wedge A(I',V')]$$

The above shows that for every variable v_i that does not appear in the support of A we can drop the term $v'_i = N_i(I,V)$ from the transition relation part of $Pred(A)$ without changing the result. If we do this for all variables not in the support of A we get $Pred_D(A)$. □

4.4 Test Generation using Dynamic Transition Relations

4.4.1 A dynamic algorithm

We now present an updated algorithm for test generation that uses dynamic transition relations. This version not only uses $Pred_D$ in stage I, but uses the same dynamic transition relations in the forward search of stage II. The idea is that in most designs the next-state value of each variable depends only on a few of the other variables, and so the support of the sets we compute will remain small.

We recall that given a state s, the application of R to s determines the next-state values for the variables in V, but not for the variables in I. The input variables I are chosen arbitrarily by the environment. The dynamic algorithm uses partial assignments σ_i instead of the full states s_i, and partial transition relations R_i instead of R. The output sequence $in_m, \ldots, in_0, \ldots, in_{n-1}$ generated by the algorithm is a series of partial assignments over some (but perhaps not all) of the variables in I. When in_i does not give a value for a variable $i \in I$ it means that i does not influence the parts of the circuit that are being considered, and its value can be chosen arbitrarily.

In the dynamic algorithm we use the following functions:

- $choose(A,U)$ accepts a set of states represented by a formula A and a set of variables U such that $sup(A) \subseteq U$. It returns a partial assignment σ over U that satisfies A. If we view σ and A as sets, then the chosen σ is a subset of A. Notice that if A happens to be given as a partial assignment a, then the resulting σ

will be an assignment over U that agrees with a. Notice also that the function *choose* that was used in the algorithm of the previous section is simply a call to this function with $U = I \cup V$.

- $project(\sigma, U)$ receives a partial assignment (or a full state) σ defined over some set of variables U', and a set of variables U such that $U \subseteq U'$, and returns the projection of the former on the later, i.e. it returns a partial assignment over U that agrees with σ.

- $apply(R_i, \sigma)$ receives a partial transition relation R_i and a partial assignment σ over $sup(R_i)$. The partial transition relation is of the form $R_i = \bigwedge_{v \in U}(v' = N_v(I, V))$ for some set of variables U. The result is a partial assignment σ' over U, such that for every $v \in U$: $\sigma'(v) = N_v(\sigma)$.

Stage I:

$A_n := t_n$
for $j = n - 1$ downto 0 do
  $R_{j+1} := \bigwedge_{v_k \in sup(A_{j+1})} v'_k = N_k(I, V)$
  $A_j := t_j \cap Pred_D(A_{j+1})$
endfor
$All := \emptyset$
while $(A_j \neq \emptyset) \wedge (A_j \cap Init = \emptyset) \wedge (A_j \not\subseteq All)$ do
  $All := All \cup A_j$
  $j := j - 1$
  $R_{j+1} := \bigwedge_{v_k \in sup(A_{j+1})} v'_k = N_k(I, V)$
  $A_j := Pred_D(A_{j+1})$
endwhile
$m := j$.

Stage II:

if $(A_m = \emptyset) \vee (A_m \subseteq All)$ then
  print "sequence cannot be generated"
else
  $s_{init} := choose(A_m \cap Init, I \cup V)$
  $\sigma_m := project(s_{init}, sup(R_{m+1}))$
  $in_m := project(s_{init}, I \cap sup(R_{m+1}))$
  for $j = m + 1$ to $n - 1$ do
    $tmp := A_j \cap apply(R_j, \sigma_{j-1})$
    $\sigma_j := choose(tmp, sup(R_{j+1}))$
    $in_j := project(\sigma_j, I)$
  endfor
  $tmp := A_n \cap apply(R_n, \sigma_{n-1})$
  $\sigma_n := choose(tmp, sup(A_n))$
  $in_n := project(\sigma_n, I)$

28: endif

Stage I of the algorithm creates the same sets A_m, ..., A_n that were created in the previous algorithm, but uses $Pred_D$ instead of $Pred$. Notice that by the end of stage I we have that for every $m < i \leq n$, $sup(A_{i-1}) \subseteq sup(R_i)$ and $sup'(R_i) = sup(A_i) \cap V$. This allows the use of apply in lines 21 and 25.

In stage II, the forward search for a path through the A_i's is done using partial assignments σ_m, ..., σ_n instead of states (which are full assignments). Every partial assignment σ_i represents a set of states which differ only on variables not in the support of A_i. When moving from σ_{i-1} to σ_i, in lines 21 and 25, we use the same partial transition relation R_i that was used to create A_{i-1} from A_i (see figure 4.1). After applying R_i, we expand the result to the support of R_{i+1} (line 22) so that we can apply R_{i+1} in the next iteration of the loop. The output of the algorithm is the initial state s_{init} and the inputs $in_m, \ldots, in_{n-1}$. The inputs calculated by the algorithm do not necessarily give values to all input variables in I. When giving inputs to a simulation tool we need to decide on values for all the input variables. We therefore expand every in_i to I by choosing arbitrary values for the extra input variables.

To show the correctness of the dynamic algorithm, we first notice that the sets $A_m, \ldots, A_0, \ldots, A_n$ computed by the dynamic algorithm in stage I are exactly the same sets computed by the static algorithm. This is because the only difference in their computation is the use of $Pred_D$ instead of $Pred$, and we have already shown that they are equivalent. This means that theorems 4.1, 4.2, and 4.3 hold for the dynamic algorithm also. What is left to show is that the sequence of partial inputs $in_m, \ldots, in_n$ computed in stage II of the dynamic algorithm will induce a trace through the sets $A_n, \ldots, A_m$ no matter how they are expanded to the full set of inputs I.

Theorem 4.1: Let $in_m, \ldots, in_n$ and s_{init} be the output of the dynamic algorithm. Then any trace that starts at s_{init} and follows a sequence of inputs that agrees with $in_m, \ldots, in_n$ will be a trace through the sets $A_m, \ldots, A_n$ computed in stage I of the algorithm.

Proof: Let $in'_m, \ldots, in'_n$ be a series of full inputs to the circuit that agree with $in_m, \ldots, in_n$, i.e. every in'_i is an assignment that gives value to all the variables in I, and for every variable $v \in I$ such that in_i is defined over v we have $in_i(v) = in'_i(v)$. Let $s_m, \ldots, s_n$ be the path created when starting at $s_m = s_{init}$ and driving the inputs $in'_m, \ldots, in'_n$. This means that given a state s_j, which agrees with in'_j on I, we apply the full transition relation R and get a partial assignment s'_{j+1} which gives values to V (since R does not determine the next state values for variables in I). We then expand s'_{j+1} to the state s_{j+1} by adding the values that in_{j+1} gives to I (thus s_{j+1} agrees with in_{j+1} on I). We show that for every $m \leq j \leq n$: $s_j \in A_j$.

We first show that in every iteration $\sigma_j \subseteq A_j$, by induction on j. For σ_m (line 18) it is obvious, since $s_{init} \in A_m$. Assume that $\sigma_{j-1} \subseteq A_{j-1}$ ($j < n$). From line 21 we know that $tmp \subseteq A_j$, so from line 22 we conclude that $\sigma_j \subseteq A_j$. For σ_n the reasoning is similar, since $tmp \subseteq A_n$ (line 25), and $\sigma_n \subseteq tmp$ (line 26). To show that for every

j, $s_j \in A_j$ we need only show that s_j agrees with σ_j on $sup(A_j)$. We prove a stronger claim, that s_j agrees with σ_j on $sup(R_{j+1})$, for $j = m, \ldots, n-1$, and s_n agrees with $sigma_n$ on $sup(A_n)$. We do this by induction on j.

The basis is trivial since σ_m was chosen so that s_{init} and σ_m agree on $sup(R_{m+1})$. Assume that s_{j-1} agrees with σ_{j-1} on $sup(R_j)$. The transition relation R includes all the processes N_i, including all those that appear in R_j. This means that for every $v_i \in sup'(R_j)$ the value that s'_j gives to v_i is the same value that $apply(R_j, \sigma_{j-1})$ gives to v_i (because they were calculated by the same N_i). Since $sup'(R_j) = sup(A_j) \cap V$, we conclude that s'_j agrees with $apply(R_j, \sigma_{j-1})$ on $sup(A_j) \cap V$ (line 21). Since the sets A_j were created so that every state in A_{j-1} has a successor in A_j, the set tmp cannot be empty. The conjunction with A_j only limits the possible values for variables in I, so every state in tmp agrees with s'_j on $sup(A_j) \cap V$. We then choose σ_j (line 22) from tmp, so we conclude that s'_j agrees with σ_j on $sup(A_j) \cap V$. The partial assignment σ_j, however, may also include values for variables in I. In line 23 in_j is created so that it is defined over every variable in I which σ_j is defined over. Since s_j is created from s'_j by adding the values for I that in'_j gives, and since in'_j agrees with in_j, we must conclude that s_j agrees with σ_j on variables from both V and I. In fact, they agree on all the variables that σ_j is defined over, which means they agree on $sup(R_j)$. To conclude the proof we need to show that s_n and σ_n agree on $sup(A_n)$. Lines 25-27, which define σ_n and in_n are similar to lines 21-23, except that σ_n is chosen over $sup(A_n)$ (line 26) instead of $sup(R_{j+1})$ (line 22). Similar reasoning as above leads us to conclude that s_n agrees with σ_n on $sup(A_n)$. □

4.4.2 BDD implementation

Both the original test-generation algorithm and the dynamic algorithm can easily be implemented using BDD representations. The components $Init$, S and $N_1, \ldots, N_l$ of the model are represented using BDDs in the usual manner. In addition, the sets A_j computed by the algorithm are represented by BDDs. The input to the algorithm is a sequence of binary vectors over T. It is straightforward to translate each vector into a BDD that represents the set t_i needed for the algorithm. Most BDD libraries include a function to compute sup and sup', which are simply the sets of current or next state variables that appear in the BDD. All other operations used in our algorithm are standard BDD operations.

A BDD implementation of the algorithm will benefit significantly from the use of partial transition relations. The size of a BDD representing a set A is generally related to the size of $sup(A)$. In many cases, each N_i will not depend on all the variables in $I \cup V$. Thus, taking fewer N_i's will result in a smaller support for the partial transition relation $\bigwedge_{v_i \in sup(A)}[N_i(s) = v'_i]$. The BDDs computed at intermediate stages in the computation of $Pred$ using the partial transition relation will depend on less variables and will often be smaller.

Model	Vars	Seq.	Time			Space		
		length	Static	Dynamic	Relative	Static	Dynamic	Relative
#1	5	30	0.93	0.95	102%	0.76	1.23	161%
#2	4	25	1.35	1.51	112%	1.02	1.50	146%
#3	8	1	49.27	2.08	4%	1.95	1.80	93%
#3	8	3	50.68	24.72	49%	1.95	2.42	124%
#3	8	5	51.27	27.57	54%	1.95	2.42	124%
#3	8	10	50.78	35.4	70%	1.95	2.2	124%
#4	8	5	2301	73.39	3.2%	595	2.51	0.4%
#4	8	10	3947	61.32	1.6%	598	2.51	0.4%
#4	8	15	4550	89.65	2%	598	2.51	0.4%
#5	3	8	fail	2.4	-	fail	3.77	-
#6	15	5	fail	7.59	-	fail	3.19	-
#6	10	5	fail	6.65	-	fail	3.19	-
#7	8	5	13.92	0.73	5.2%	17.98	1.74	9.7%
#7	8	5	16.43	0.44	2.7%	17.98	1.53	8.5%
#8	8	5	fail	18	-	fail	7.29	-
#8	8	5	fail	31.72	-	fail	7.29	-
#9	4	5	fail	21.51	-	fail	5.54	-

Table 4.1: Results for symbolic test-sequence generation algorithms

4.4.3 Results

Both the original and the dynamic algorithm were implemented by Fady Copty at Intel Israel. The algorithms were used to create test sequences on hardware models designed at Intel.

Nine different examples were used, and for each one both algorithms were applied to create test sequences. Since different test sequences can create different behaviors of the algorithm, for some examples more than one test sequence was used. The results are presented in table 4.1.

Times are given in seconds and measures of space are given in megabytes. The "Dynamic" columns relate to the dynamic algorithm, while the "Static" columns relate to the static algorithm. The "Relative" column gives the dynamic result divided by the static result. The numbers in the table are rounded, but the "relative" column is calculated from the original numbers. The "Vars" column is the sub-circuit input vector width (the number of variables in I_{sub}). The "Seq. length" column shows the length of the test sequence for the sub-circuit (the constant n in the algorithm).

Out of the 17 examples, in 11 examples both the static and dynamic algorithms completed successfully, while in the others (mostly the largest) the static algorithm could not complete in the given amount of memory but the dynamic algorithm could.

As can be seen from these results, the dynamic algorithm is better suited for large examples. In the smallest ones (the first two), there were no gains from using dynamic transition relations. In fact, the dynamic algorithm required more run time and more space than the static algorithm. This happens because it is necessary to reconstruct the transition relation at each iteration, which in small examples may take more time and space (due to intermediate computations) than simply using the transition relation for the complete circuit in all stages. On large examples, however, the gains are significant, as for model #4. On the second example for this model the dynamic algorithm ran in 1/64 of the time taken by the static one, while using only 1/238 of the memory.

The results for model #3, however, are puzzling, since in 3 out of 4 sequences the gains are an order of magnitude less than those for the first sequence, and for the larger examples. The dynamic algorithm required more space than the static, although less time. This indicates that the method can be more efficient for some types of circuits than others. Future work includes the characterization of which are the "good" circuits and which are the "bad" ones.

Even better results were achieved by the examples that only finished with the dynamic algorithm. For model #5, the dynamic version ran for 2.4 seconds using only 3.77M of memory. During experiments we killed a process whenever it used all of the memory available, 800M. This example was killed after running more than 2 hours. From this we can conclude that in this example the dynamic algorithm ran in *less than* 1/3000 (2 hour / 2.4 seconds) of the time using *less than* 1/212 (800/3.77) of the memory of the original algorithm.

These results show that the dynamic transition relations method can provide significant gains in verification time and space, in some cases up to two orders of magnitude. As expected, it does not work in the same way for all kinds of circuits, but our experiments seem to indicate that it works extremely well for several types of circuits that are used in industry today.

4.5 Dynamic Transition Relations in Model Checking

4.5.1 Incorporating $Pred_D$ into symbolic model checking

The most expensive computation step in CTL model checking algorithms is the application of the **EX** operator. State explosion often occurs during this step. We notice that the **EX** operator is exactly the $Pred$ operator that was defined earlier in this section. We replace the computation of **EX** by the operator $Pred_D$ that uses a partial transition relation. As before, we compute the partial transition relation *dynamically* according to the set to which **EX** is applied. Since in most cases each N_i is defined over a small number of (unprimed) variables, by referring to the smallest number of N_i's, we reduce the number of variables used in intermediate computations.

The same treatment also handles model checking for Fair-CTL which is the logic

CTL, extended with fairness constraints that restrict the set of paths in the model which are required to satisfy a given formula [14, 6].

In many cases it is useful to know the set of reachable states when performing a model checking algorithm. This is done by taking the set of initial states and repeatedly computing the set of successors, until no new states can be found. This process is called a *forward search* (or reachability). We propose to use dynamic transition relations in this case also. The result of the forward search will be an over-approximation of the set of reachable states. This is in contrast to previous uses of dynamic transition relations where precise results were obtained. We are not guaranteed to find only reachable states, but we may still find a set of states which is significantly smaller than the set of all possible states.

The operator that computes the set of successors is called $Succ$ and is defined as: $Succ(A) = \exists I', V'.\ R(I', V', I, V) \wedge A(I', V')$ (the result is the set $S(I, V)$ of states that satisfy the formula). We define a dynamic operator $Succ_D$ that will use only some of the transition relations N_i, but not all of them.

As before, we determine which transition relations to use according to the support of the set on which we are operating. This time, however, we choose the functions N_i according to their current state support, and not their next state support. We define the set of processes which are relevant to the set A as: $\text{relevant}(A) = \{i \mid sup(N_i) \cap sup(A) \neq \emptyset\}$. We then define: $Succ_D(A) = \exists u.\ A(u) \wedge \bigwedge_{i \in \text{relevant}(A)} v_i' = N_i(u)$.

When examining the formula for $Succ_D(S)$ we see that $Succ(A) \subseteq Succ_D(A)$, which means that we do not lose reachable states by using $Succ_D$ instead of $Succ$. We will, however, take into account states which are not necessarily reachable. This may happen if for some i and j we have that $i \in relevant(A)$,$j \notin relevant(A)$, and N_i and N_j depend on common variables. This dependency may disallow certain combinations of values of v_i' and v_j'. By omitting a relation we relax the condition on next-state variables and allow combinations which were originally impossible. We expect that using the approximate set of reachable states will help model checking, even if it is a larger set than the real one.

It is also possible to compute the minimal set of processes that need to be used in order to get accurate results. To do this, we start with the set of indices $\text{exact}(A) = \text{relevant}(A)$ and iteratively add to $\text{exact}(A)$ every i such that $sup(N_i) \cap \bigcup_{j \in \text{exact}(A)} sup(N_j) \neq \emptyset$, until no indices are added. Using $\text{exact}(A)$ instead of $\text{relevant}(A)$ in $Succ_D(A)$ results in a larger transition relation, and less saving in time and space, but gives an accurate result.

4.5.2 Implementation and Experimental Results

The dynamic transition relations method was implemented in SMV. In order to test the ideas proposed the SMV code was modified in the following way.

- A table is created, which associates a variable in the model (v_i) to a transition relation describing the value of that variable in the next state (N_i).

- Each ASSIGN statement in the program describes the next-state value for a variable as a function of the current-state values of several variables. The expression generated by each ASSIGN is used to fill the table above. This replaces the original SMV code that produces the global transition relation.
- The computation of the set of predecessors of a state set S is performed by determining the support set of S, and using those variables in $Pred_D$.

This implementation does not support many features of SMV including the TRANS statement, asynchronous modules, and fairness. Moreover, it has not been optimized, while the original SMV code contains many optimizations. Because of this the results obtained are preliminary. We expect better results once these issues are addressed.

We have tested our method on models that have already been verified by SMV such as the robotics controller described in [11] and the PCI local bus [9]. Table 4.2 summarizes the results obtained for the following examples:

1 The distributed heterogeneous real-time system described in [10];

2 A simplified cache coherence protocol derived from the PCI Local bus;

3 The model of the PCI Local bus discussed in [9];

4 The robotics controller presented in [11];

5 The model of a real-time pipelined system with 9 phases. This model checks the timing properties of the architecture.

Time measurements are given in seconds, space measurements and transition relation sizes (TR) are given in 1000's of BDD nodes. The "St." columns refer to the static version (using $Pred$) and the "Dyn." columns refer to the dynamic version (using $Pred_D$). The transition relation size reported for the dynamic algorithm is the average of the sizes of the transition relations used in all iterations. The "D/S" columns summarize the gains of the method by dividing the dynamic result by the static result (and multiplying by 100 to get percentages). The column "Var" presents the total number of variables in the model (source code variables, not boolean variables) and the maximum number of (source code) variables used by $Pred_D$ at any iteration.

From this table we can see that the gains in time were significant, but the gains in space were not. In fact, in most cases the method used more memory than the original one, an unexpected effect. This may be caused by an unoptimized feature in the preliminary implementation, but it cannot be guaranteed. One positive result is a significant decrease in the transition relation size on average. Also, it is important to see that the gains are relevant even in the cases where the number of variables considered during the search was close to the total number of variables. Of particular interest is the last example where in spite of the fact that all variables were considered at some point in the search, the average transition relation size was a quarter of the total size.

Ex	Time			Space			TR			var
	St.	Dyn.	D/S	St.	Dyn.	D/S	St.	Dyn.	D/S	
1	223	43	19%	39	32	82%	4.4	.9	20%	14/9
2	115	56	49%	248	294	118%	9.6	4.9	51%	33/26
3	515	161	31%	147	354	240%	4.9	4.9	100%	31/26
4	19	13	68%	238	186	78%	4.7	3.6	77%	10/8
5	136	67	49%	104	233	224%	70	18	26%	13/13
Av			43%			148%			55%	

Table 4.2: Results of using dynamic transition relations in SMV

One other example was verified, an extremely large and complex cache coherence protocol derived from a more detailed specification of the PCI bus than the one mentioned above. The model for this circuit is very large and on our machines we have not been able to finish the verification. We have run both algorithms only on the first three iterations of this model. The original SMV took 6952 seconds to complete these iterations (about two hours), used 160,000 BDD nodes and the transition relation size was 10,983 BDD nodes. This model has 51 variables. The dynamic transition relations algorithm was much more efficient, taking 146 seconds to perform the same search, a gain of almost 50 times! It used 186,000 BDD nodes and the average transition relation had 2209 BDD nodes. The algorithm reported a maximum number of variables used of 22.

Unfortunately, we cannot extrapolate these results because it is often the case that the initial steps in the search use significantly less variables than later ones. However, this indicates another very important use for the dynamic transition relations method. During early phases of the design errors appear very frequently, and they are usually found in short execution sequences. The method proposed can then be used to perform shallow searches in a much more efficient manner than the original one, and in this way we may considerably speed up the debugging phase of the design.

These results show that the dynamic transition relations method can provide significant gains in verification time, even though more research is needed to study the behavior of the algorithm with respect to space requirements. We expect better gains once the prototype implementation is optimized.

Chapter 5

SoftVer

5.1 Overview

The SoftVer system is a model-checker that implements *modular model checking* for the temporal logic CTL. Its features include:

- A simple, structured programming language, with boolean, integer and array types.
- Control over the partitioning of a program into modules by use of a special directive.
- Local reachability. When enabled, this option may result in smaller memory consumptions.
- A BDD based model checker that utilizes all the advantages of symbolic model checking. The tool uses a BDD library by David Long [43].

The following sections will elaborate on the above.

The main two units of the SoftVer code are the compilation and the model-checking units. The compiler module parses the program, creates the partition graph, and creates the transition relation for each module in the partition graph. The model-checking unit receives the partition graph created by the compiler unit and performs modular model-checking according to the algorithm described in section 3.2.

5.2 The SoftVer Programming Language

The SoftVer language is based on the simple *non-deterministic while programs* language that was presented in the theoretical part of this work, and allows verification of sequential processes. Figure 5.2 gives an example of a short program written in the SoftVer programming language (the numbers on the left are used as reference and are

not part of the programming language). The example program inputs an array of numbers, and then performs a bubble sort on it.

Program variables can be of type boolean, integer, array of boolean, or array of integers. Comments are written in the C style, starting with '/*' and ending with '*/'. All the usual operators can be used in expressions: the boolean operators &&, ||, $->$ and ! (not); integer operators +, - (both binary and unary minus); and comparison operators =, !=, $<, >, <=, and >=$. The program starts with the preamble in which variables are declared (lines 1-5). The program body is inclosed in curly-brackets (lines 6 and 38). The #MODULE# directive (line 7) is inserted at the beginning of each module. The parser checks that the partitioning adheres to the rules. In the present version, non-deterministic assignment (line 11) is allowed only when assigning into basic variables, and not when assigning into arrays. The command 'label' (line 37) is used to name specific program counter locations, which will later be referred to in the specification. Its semantics is identical to the 'skip' command.

The specification to be checked is given after the program (line 39). It is a CTL formula that may use all the above mentioned operators, plus the temporal operators AX, EX, AU, EU, AF, EF, AG, and EG. The operators AU and EU are binary and are written as A[p U q] or E[p U q] (where p and q are boolean expressions).

5.3 Creating The Transition Relation of a Program

The process of creating a transition relation for the program starts with the allocation of BDD variables. Knowing the number of program locations, we calculate the minimum number of bits needed for the program counter and allocate the BDD variables $\overline{pc} = pc_1, \ldots, pc_k$ and $\overline{pc}' = pc'_1, \ldots, pc'_k$. The BDD variables $\overline{pc}$ are used to represent the current program location, and $\overline{pc}'$ are used for the next program location. For each program variable we calculate the number of bits needed to represent its value (1 for boolean, more for integer). We then allocate two sets of BDD variables: $\overline{v} = v_1, \ldots, v_n$ are used to represent the current state and $\overline{v}' = v'_1, \ldots, v'_n$ are used to represent the next state.

The transition relation of a module is a disjunction of the relations of the individual commands. Before we can create transition relations for the individual commands we must be able to create BDD representations for the different expressions that may appear on the right hand side of assignments and in boolean expressions (in "if"s and "while"s). This is done according to the structure of the expression. We start by defining BDD representations for constants and variables (of all types), and then define the effect of the different operators. We give here the definitions for only a few representative operators, as the list of operators that SoftVer supports is quite long. BDD representations are given as boolean formulas over BDD variables.

- A boolean constant c:
  The boolean constants "TRUE" and "FALSE" are represented by the BDDs $true$ and $false$.

```
1:  int tmp;
2:  int index;
3:  int sub_ind;
4:  boolean flag;
5:  int a[5]; /* array with places 0-4 *

6:  {
7:  #MODULE#
8:  /* input numbers into array */
9:  index := 0;
10: while (index < 4) do
11:     tmp := {0,1,2,3};
12:     a[index] := tmp;
13:     index := index + 1;
14: od;
15: flag := TRUE;
16: index := 4;

17: #MODULE#
18: while (flag && (index > 0)) do
19:     #MODULE#
21:     sub_ind := 0;
22:     flag := FALSE;
23:     while (sub_ind < index) do
24:         if (a[sub_ind] > a[sub_ind + 1]) then
25:             tmp := a[sub_ind];
26:             a[sub_ind] := a[sub_ind + 1];
27:             a[sub_ind + 1] := tmp;
28:             flag := TRUE;
29:         else
30:             skip;
31:         fi;
32:         sub_ind := sub_ind + 1;
33:     od;
34:     index := index - 1;
35: od;

36: #MODULE#
37: label halt;
38: }

39: SPEC AF (label(halt) && (a[0]<=a[1]) && (a[1]<=a[2]) &&
                          (a[2]<=a[3]) && (a[3]<=a[4]))
```

- A boolean variable x:
  Let u_i be the BDD variable that represents the program variable x. The formula representing the expression x is u_i.

- An integer constant c:
  Integers are represented by an array of BDDs. All integers are of the same fixed length $integer\_length$. An expression of type integer is an array of length $integer\_length$ of boolean expressions, that represents numbers in binary code. For example, if $integer\_length = 3$ then the constant 5 is represented by the array $[true, false, true]$.

- An integer variable x:
  Assume that $u_1, .., u_m$ are the BDD variables allocated for x ($m = integer\_length$). The ith element of the array representing x is the BDD u_i.

- A boolean array dereference $b[e]$ (e is an integer expression):
  At this point of the compilation the BDD for e has already been created. Actually, since e is an integer expression it is represented by an array of BDDs, each representing a bit in binary code. Let $u_1, \ldots, u_t$ be the variables that represent b, i.e. u_1 represents $b[1]$, u_2 represents $b[2]$ and so on. The value of $b[e]$ is $b[1]$ if $e = 1$, $b[2]$ if $e = 2$ and so on. Therefore, the BDD for $b[e]$ is: $(e = 1 \wedge u_1) \vee (e = 2 \wedge u_2) \vee \ldots \vee (e = t \wedge u_t)$. The formula $e = 1$, for example, is created by equating the least-significant bit to $true$ and all the rest to $false$.

- An integer array dereference $a[e]$ (e is an integer expression):
  The result of dereferencing an integer array is an integer number. To create a BDD representation for an integer we create an array of length $integer\_length$ of BDDs. Each place in the array represents a bit in binary code, and is created in a similar way to the expression for a boolean array dereference. Let $u_1, \ldots, u_t$ be the BDD variables allocated for the ith bit of a. This means that u_1 is the ith bit of the number $a[1]$ and u_t is the ith bit of the number $a[t]$. The ith bit of the BDD representation for $a[e]$ is $(e = 1 \wedge u_1) \vee (e = 2 \wedge u_2) \vee \ldots \vee (e = t \wedge u_t)$.

- The boolean operator && (logical AND):
  If $F(\overline{v})$ is the BDD for the boolean expression f, and $G(\overline{v})$ is the BDD for the boolean expression g, then the BDD for the expression f && g is $F(\overline{v}) \wedge G(\overline{v})$.

- The integer operator +:
  Given two numbers, represented in binary code as arrays of BDDs, the operator + is computed in the usual manner. The least-significant bit of the result is the exclusive-or of the least-significant bits of the parameters. The carry is computed as the conjunction of both least-significant bits, and then the next bit can be calculated.

The above is only a partial list, that shows the basic types of operations used to compute BDD representations for expressions. For more details refer to the code itself.

Following is a description of the translation of commands into relations, using BDD operations [1]. Each relation R is described as a formula with $\overline{pc}, \overline{v}, \overline{pc'}, \overline{v'}$ as its free variables. The label l is a number representing the current program location, i.e. the location of the command being translated. The label l' is a number representing the location of the subsequent command. These numbers do not appear in the text of the program, but are created by the parser.

- l: skip l':
  The transition relation of the skip command reflects the fact that the program counter location moves from l to l', but no other variable changes.

$$R(\overline{pc}, \overline{v}, \overline{pc'}, \overline{v'}) \equiv (\overline{pc} = l) \wedge (\overline{pc'} = l') \wedge (\bigwedge_{1 \le i \le n} v_i = v_i')$$

- l: $x := e\ l'$: (where x is a simple variable and e is an expression)
  We break the set of BDD variables into two parts, $\overline{u} = u_1, \ldots, u_k$ are the variables associated with x, and $w_1, \ldots, w_n$ are all the rest. Let $[e_1(\overline{u}, \overline{w}), \ldots, e_k(\overline{u}, \overline{w})]$ be the array of BDDs that represents the expression e. If x is boolean (and so is e) then $k = 1$. Otherwise, $k = integer\_length$.

$$R(\overline{pc}, \overline{u}, \overline{w}, \overline{pc'}, \overline{u'}, \overline{w'}) \equiv (\overline{pc} = l) \wedge (\overline{pc'} = l') \wedge (\bigwedge_{i=1}^{n} w_i = w_i') \wedge \bigwedge_{j=1}^{k} u_j' = e_j$$

- l: $a[ind] := e\ l'$:
  Assume that $\overline{u^j} = u_1^j, \ldots, u_k^j$ is the set of BDD variables that is associated with $a[j]$ $(1 \le j \le t)$, and let $\overline{w}$ be all the rest of the BDD variables. The expression e must be of the same type as each element of a, so it is represented by the BDDs $e_1, \ldots, e_k$. If a is a boolean array then $k = 1$, otherwise $k = integer\_length$.

$$R(\overline{pc}, \overline{u}, \overline{w}, \overline{pc'}, \overline{u'}, \overline{w'}) \equiv (\overline{pc} = l) \wedge (\overline{pc'} = l') \wedge (\bigwedge_{i=1}^{n} w_i = w_i') \wedge$$

$$\bigvee_{i=1}^{t} (ind = i) \wedge (\bigwedge_{j \in \{1,\ldots,i-1,i+1,\ldots,l\}} \overline{u^j} = \overline{u^j}') \wedge (\bigwedge_{j=1}^{k} u_j^{i\prime} = e_j)$$

- $x := \{e_1, \ldots, e_n\}$
  The transition relation for a non-deterministic assignment is the disjunction of the relations of the assignments "$x := e_1$" through "$x := e_n$".

- l: if B then $l_1 : S_1$ else $l_2 : S_2$ fi l':
  The transition relations of S_1 and S_2 have already been computed and disjuncted

[1] This description is a simplified version. The actual code is slightly more complicated since it incorporates some optimizations.

into the global transition relation. All we need to add are the transitions in which B is evaluated:

$$R(\overline{pc}, \overline{u}, \overline{w}, \overline{pc}', \overline{u}', \overline{w}') \equiv (\overline{pc} = l) \wedge (\bigwedge_{1 \leq i \leq n} v_i = v_i') \wedge ((B \wedge (\overline{pc}' = l_1)) \vee (\neg B \wedge (\overline{pc}' = l_2)))$$

- l: while B do l_1: S od l':
  Similarly to the previous case:

$$R(\overline{pc}, \overline{u}, \overline{w}, \overline{pc}', \overline{u}', \overline{w}') \equiv (\overline{pc} = l) \wedge (\bigwedge_{1 \leq i \leq n} v_i = v_i') \wedge ((B \wedge (\overline{pc}' = l_1)) \vee (\neg B \wedge (\overline{pc}' = l')))$$

5.4 Analyzing the performance of the tool

We now describe the methods that were used when analyzing the performance of SoftVer and the modular model checking algorithm. This is important because, as we will show, not all methods of analysis that seem plausible are in fact accurate.

When analyzing the performance of any algorithm we look at memory consumption and run-times. Run-times are relatively easy to measure, using the Unix command *timex*, which gives the overall running time, the amount of time used for system (swapping and other system operations), and the amount of time used by the program itself. The running times reported throughout the work are the actual user times. The system part of the run time is ignored, since it never went over 3 or 4 seconds (in most cases it was less than a second).

Analyzing the memory consumption of a model checking algorithm is a little more tricky, especially if it is BDD based, as SoftVer is. Any BDD library includes a function that, when called, will return the number of BDD nodes currently in use. Naturally, garbage collection algorithms are used to dispose of unused nodes every once in a while, since it would be inefficient to dispose of every node as soon as it is freed. It would seem natural to examine the number of BDD nodes in the system just before each time garbage collection is performed, and regard the maximum of these numbers as the maximal memory consumption of the algorithm. However, this method is inaccurate. Garbage collection is only performed between BDD operations, and memory consumption usually (almost always in fact) reaches its peak in the middle of difficult operations, usually the computation of a relational product as in the calculation of the **EX** or **AX** operators. To get an exact measure of how much memory a run of the algorithm used we used the Unix command *limit*. This command enabled us to limit the amount of memory a shell process can use. When a process tries to exceed this amount it will fail, and report failure of allocation or some other similar message. We ran each example several times, each time changing the limit on the amount of data the process is allowed to use, until we found a tight bound on the actual amount of memory used. A tight bound is a pair of numbers l, h such that when limiting the shell to l K (Kilo Bytes) the process failed, but when limiting the shell to h K it was able to finish, and the difference $h - l$ is less than 1% of h.

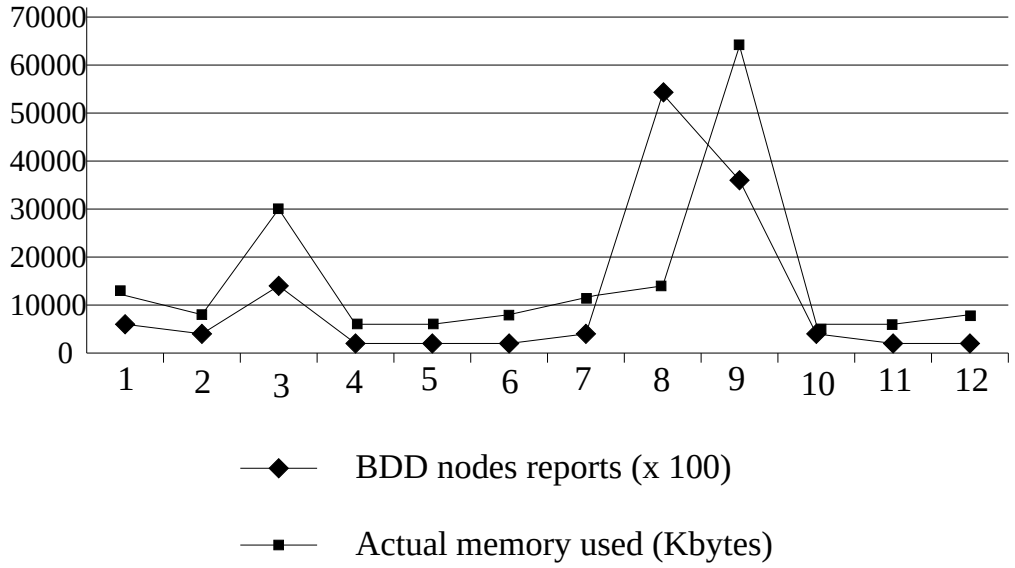


Figure 5.1: Comparison of methods for analysis of memory consumption.

The method of determining memory consumption described above requires a lot of work, since each example must be run many times. However, we find that it is necessary since the method of printing the number of BDD nodes used is inaccurate. To prove this we used both methods for our examples and compared them. Figure 5.1 shows a graph depicting the memory consumption of several runs of the Stop and Wait example, using both measurement methods. Although generally the two lines rise and fall together, runs 8 and 9 prove that in some cases the method of printing BDD nodes can lead us to believe that one example used less space than the other, when in fact the opposite is true. For this reason, all the memory consumption results given for the modular model checking algorithm were calculated using the accurate method. The results given in the hardware verification part of the work were done using the accurate method.

5.5 Variable Reordering and Local Reachability

The SoftVer model checker supplies two options that may help in handling large programs: variable reordering and local reachability.

As mentioned before, an ordering is defined over BDD variables, and this ordering may influence the size of the BDD. Variable reordering is a heuristic algorithm that attempts to find a better ordering for the variables. When the reordering option is enabled, the variable reordering algorithm is triggered when the BDD size exceeds a certain limit.

In Figure 5.2 we give graphs that show the effect of variable reordering on space (figure 5.2(a)) and time (figure 5.2(b)) consumption. There is no graph for the Sort example, since it could not finish at all without variable reordering. It is easy to

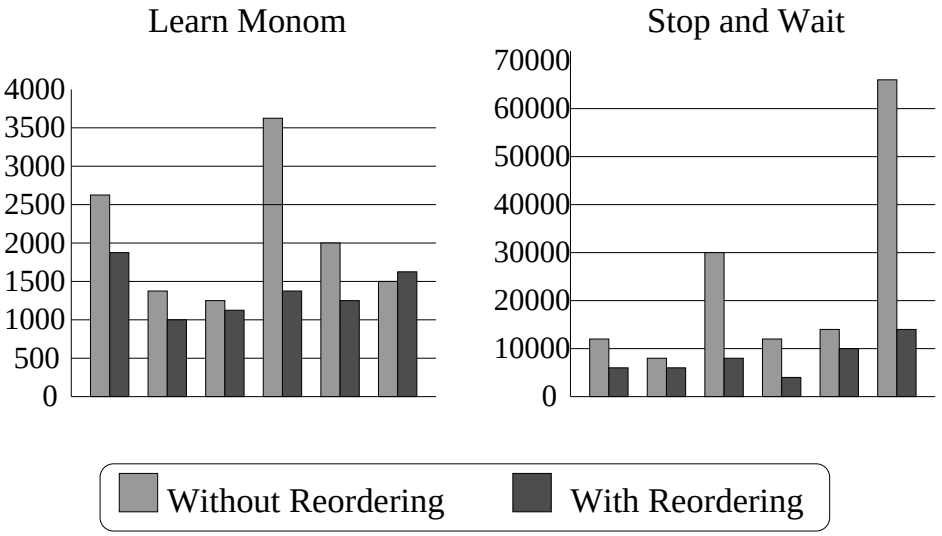
see that variable reordering is a valuable tool in handling large programs. In most cases variable reordering reduced both space and time consumption, and the most impressive reductions were achieved in the largest examples. It seems that although the variable reordering algorithm itself takes a significant amount of running time, this time is well spent since it saves time in the overall run. There is another advantage in using variable reordering in our case. Since we work on different modules separately, when variable reordering is enabled each module may have a different reordering, one which is suitable for this module and may not be suitable for others.

Computing the set of reachable states before performing model checking can sometimes reduce the space requirements needed for model checking, and it is a common practice in symbolic model checking. However, this computation may be expensive in both running time and space requirement. The SoftVer model-checker offers the possibility to perform local reachability. Local reachability means that we do not compute the exact set of reachable states, and yet we limit the state-space used for model-checking.

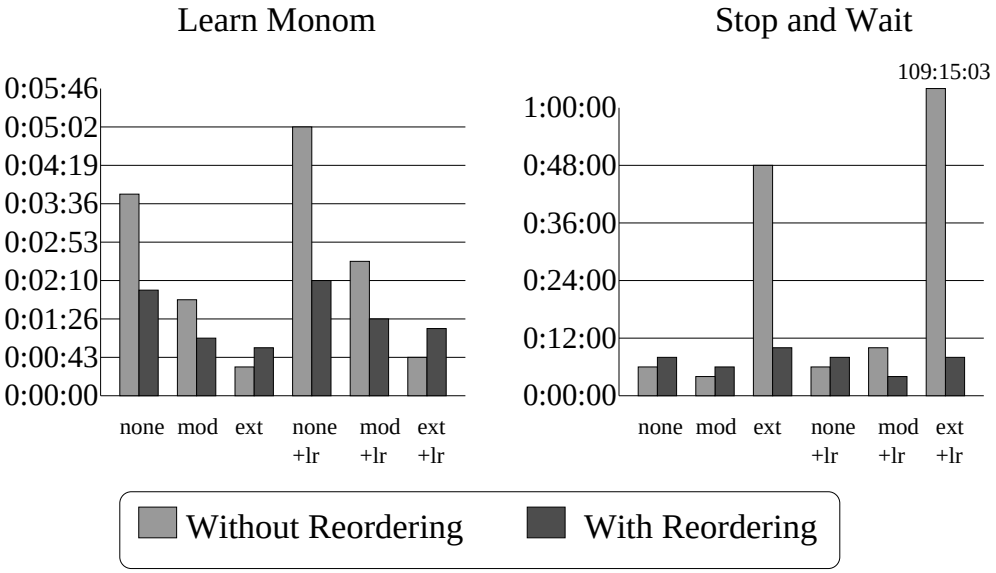
The reason why reachability can be so difficult is because wherever there is a loop in the program, the reachability algorithm must pass through this loop time after time until reaching a fix-point.

When performing local reachability we compute the reachable state-space in the regular way as long as we are translating simple commands: skip, assignment, and conditions. After creating the transition relation for such a statement we compute the set of ending states according to this transition relation, and limit the transition relation of the next statement to include only transitions that start from reachable states. For a program without loops this process will produce the same result as regular reachability. When there is a loop, however, we refrain from computing the fix-point. We translate the head of the loop, the point at which the boolean condition is evaluated, into a BDD representation according to the definition given in the previous subsection. The body of the loop is created using local reachability under the assumption that all initial states of the body are reachable (which, of course, might not be true). Using local reachability means that the state-space of our model will now include some unreachable states, but not as much as without reachability.

In Figure 5.3 we give graphs that show the effect of using local reachability on space (figure 5.3(a)) and time (figure 5.3(b)) consumption. From these graphs we see that in most cases the use of local reachability actually increased both space and time consumption. However, there are cases in which local reachability improved performance. It is interesting to note that in one case (learn-monom, no partitioning with reordering) space consumption was reduced while time consumption was increased, and in another case (stop-and-wait, moderate partitioning with reordering) space consumption was increased while time consumption was reduced. All of the above suggests that local reachability should not be a default option, since in many cases it is not useful, but in cases where SoftVer runs too long, or requires too much space, one should try using local reachability, in the hope that it would help in this case. It is worth noting that local reachability seems to help mainly when combined with variable reordering.



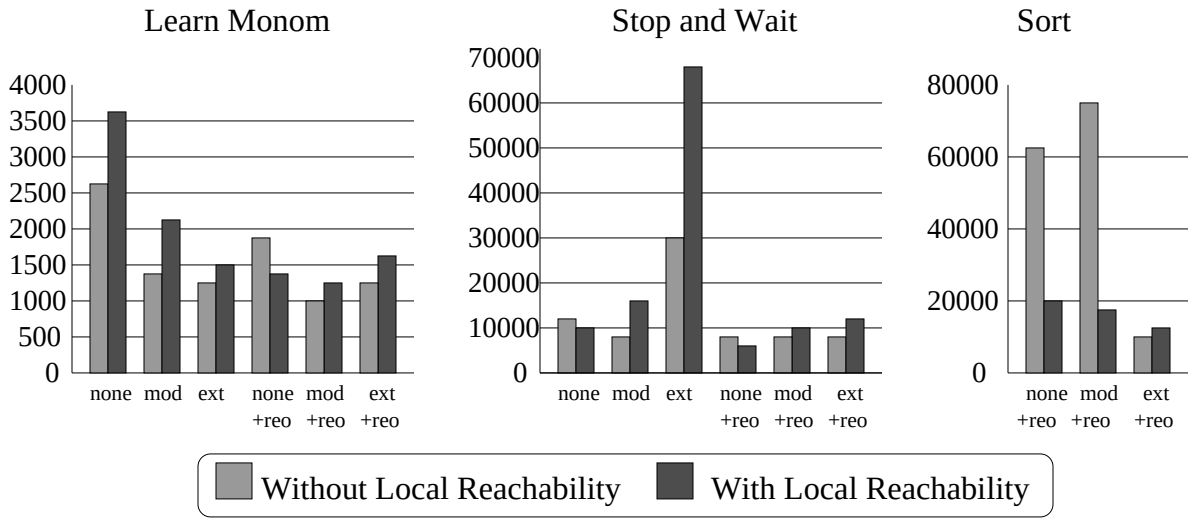
(a) Graph of memory consumption



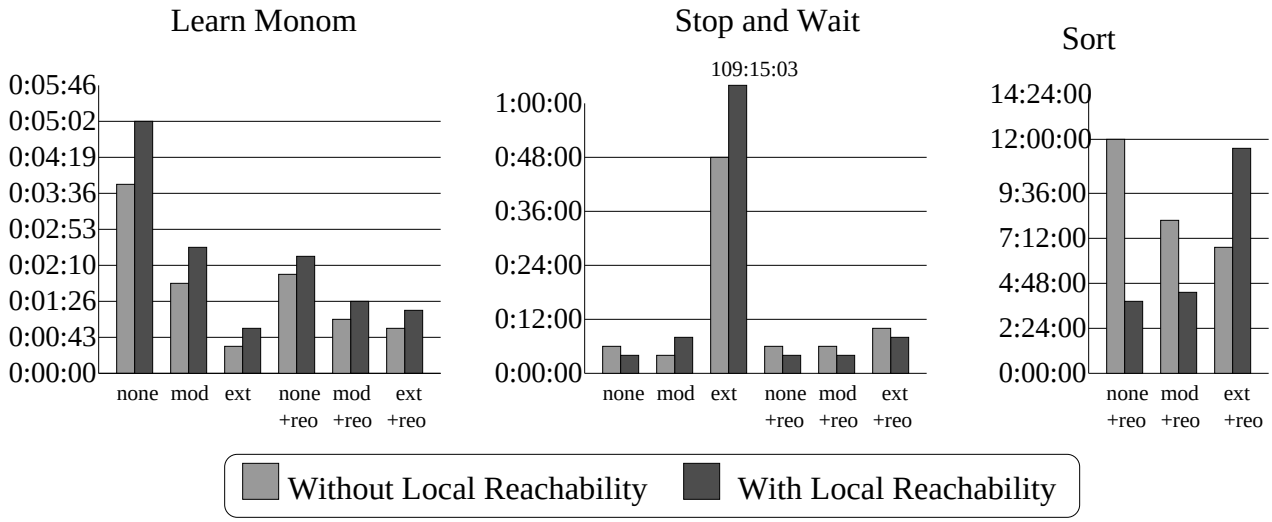
(b) Graph of time consumption

Figure 5.2: The effect of variable reordering

This is a known behavior of BDDs where reducing the number of states in a set does not necessarily reduce the size of the BDD representing this set. However, when the states that are eliminated are "irregular" in some sense, then there may be a variable ordering that will reduce the size of the BDD.



(a) Graph of memory consumption



(b) Graph of time consumption

Figure 5.3: The effect of local reachability

Chapter 6

Conclusions

6.1 Related Works

There are not many works that deal with automatic verification of software systems, and especially in programs written in high level languages.

A very interesting work by Godfroid [29] presents a verification method for software written in actual programming languages such as C or C++. This method searches the state-space of a concurrent program by repetitively running each process until it reaches its next communication statement. The SPIN system [32] is another example of model checking for high-level languages. It uses a language called PROMELA and is mainly used to verify communication protocols. Both of these tools are not modular in nature, and it is not clear whether they can make use of our modular model checking algorithm, but they may be able to incorporate our reduction methods into their verification process.

Works that use process algebras to represent a program can be considered as handling software, although not written in a high-level language. One of the most relevant works, which bears some resemblance to our modular model checking algorithm, is the work of Burkart and Steffen [7, 8]. They present model checking algorithms for context-free processes, and for a generalization of context-free processes called pushdown processes. They consider the semantics of 'fragments', which are interpreted as 'incomplete portions' of the process. The model checking algorithms they propose are based on the calculation of the *property transformer* of each fragment, which is a function that represents the semantics of a fragment with respect to alternation-free μ-calculus formulas. A property transformer receives a set of μ-calculus formulas which are true at the exit point of a fragment, and returns the set of μ-calculus formulas true at the entry to the fragment. After calculating the property transformers of all fragments, the property transformer of the main fragment is given the set of μ-calculus properties that hold when a process halts, and the result computed by the property transformer is the result of the model-checking algorithm. The main draw-back of this algorithm, in our opinion, is that it is defined for Pushdown Processes Systems, which can hardly be considered as a high level programming language, and μ-calculus

properties, which are not easy to use. This makes it an interesting theoretical result, but not useful for use by real designers of hardware or software systems. It should also be noted that the property transformers of all the fragments are computed together, one depending on the other, and it is not clear whether using secondary memory to store partial results would prove useful, or even possible.

As mentioned before, our path reduction method is closely related to partial order reduction methods since it excludes possible interleavings between processes. The dead-variable reduction can also be considered a partial order reduction since it excludes some of the successors of a state in which x is dead. It should be noted, though, that the reduction we achieve is different than the reduction achieved by most partial order methods, and it should be beneficial to use one on-top of the other.

A similar type of reduction to our path reduction was introduced by Miller and Katz in [36]. Their approach was to eliminate invisible states from the model of a program, where invisible states are states for which all the entering transitions cannot influence the specification. Their method constructs the projected visible state space relative to a specification through a DFS traversal that eliminates invisible states. The construction of the visible state space requires a linear traversal of a model that is somewhat reduced from the original model of the system, but is still larger than the reduced model which is produced. The difference from our approach is that we produce the reduced model from the syntactic model of the program (the control-flow graph) and not from the Kripke structure representing it. The syntactic model is significantly smaller since it expands only the program counter and not the program variables, which are the source for the enormous size of the semantic model.

The main advantage of our reduction methods is that they use static analysis to create the reduced Kripke structure. In [33, 38] static analyses are used for partial order reductions. An analysis of the statements in the program ([33]) or of the control-flow graphs of the processes [38] is used to determine the transitions to be traversed (ample sets) and to create a reduced model on which a full state-space exploration is performed. The main difference between our reduction methods and theirs is in the model that is produced, each method creates a different reduced model. Another important distinction is that our reductions work for either CTL* or CTL*-X whereas the reductions presented in [33, 38] are appropriate only for the subset LTL-X.

The closest related work to our dead-variable reduction is [3], where a live variable analysis is used to create a reduced model for asynchronous processes that communicate via queues. The analysis they present is similar to what we present as *fully dead* variable analysis. Our dead-variable reduction is more effective since it allows variables to be partially dead. We used the example that is presented in [3] and it shows better results than our other examples. However, we believe that this reduction is achieved because the example was tailored for demonstrating the effectiveness of the method. Our examples are implementations of known protocols.

Our state-space reductions are also related to works like [1, 15, 20, 37] that use abstract interpretation techniques [17] to obtain reduced models that preserve subsets of the logic CTL^*. Their reductions, however, are not fully automatic since the user

must define the abstract domain. Furthermore, these works provide weak preservation while our reductions provide strong preservation.

Test sequence generation from precomputed tests has been studied out of the context of symbolic model checking [47, 12, 40, 39, 54, 26]. However, all of these techniques depend on the internal structure of the circuit being tested, (e.g. in some cases they rely on a regular bus structure in the design), and consequently are not as general as the one presented here. In the context of model checking the problem has been addressed in [25], but their work is concerned mostly with expressing the test sequence and not with the complexity of dealing with large circuits. In fact, in [25] the authors state that they have only used their method on small examples. Another work relating to model checking and test sequence generation is [2], in which ATPG algorithms (Automatic Test Pattern Generation) are used to perform model checking. ATPG algorithms do not guarantee that they will eventually find a test sequence even in cases when it does exist, as our algorithm does.

One important aspect of our work is that it does not build a model of the complete circuit beforehand and that it may never actually construct such model. In this aspect it relates to techniques such as partitioned transition relations [13]. It differs from it, however, because in that case all partitions are used in every iteration, and this may not be necessary. Examples of techniques that may not consider all parts of the circuit are the cone of influence reduction [13]. However, these techniques are static in the sense that they determine only once which parts of the circuit can be ignored. Our method does it dynamically taking advantage that not all parts of the circuit are relevant during all iterations. Because of this our method produces better results, since it can use less of the circuit during most iterations. In fact, the cone of influence can be seen as an upper bound on the behavior of our algorithm.

6.2 Directions for further research

There are several directions for research which we have not yet pursued. The first is utilizing the procedural structure of programs. Procedures are sub-programs which are defined once, and then used in several places in the program, with some changes in variables. It would be interesting to find a way to verify the program while examining the body of each procedure only once. The ideal solution would be some algorithm that would examine the structure of the procedure, extract the minimal information needed, and then, when working on the program that called the procedure, use only the information kept about the procedure, without examining its structure once again. For this to be possible we need to keep information about the semantics of the procedure that would enable us to verify every call to the procedure, although different calls may use different actual parameters. The problem we encountered here is that any information we keep that does not include the full branching structure of the body of the procedure was not enough for an exact model checking algorithm. A possible solution might be to settle for partial knowledge, and get results which under or over

approximate the correct results. Another possibility is to find a specification language which is not as sensitive to branching structure as CTL is, and yet be compositional, and expressive enough to be interesting.

Another possible continuation of our work in software verification is to exploit other kinds of static analysis algorithms. There are other attributes of programs and variables which can be extracted using static analysis, and may be helpful in reducing the state space of a program, or making the model checking algorithm more efficient. It may also be possible to extend the reductions we present to handle more cases. For example, in our dead-variable reduction we cannot reduce according to an array variable. This is because our simple calculations cannot determine to which element of the array an expression such as $a[i]$ refers to. The only safe assumption that can be made is that such an expression is a reference to the whole array, which would make the reduction un-effective. Extending the method to handle arrays will require a new type of static analysis.

As for our research in hardware verification, our algorithm for test-sequence generation should be extended to handle sets of test-sequences (instead of working on a single test-sequence for the critical sub-circuit). This would prove very practical since in real life a design is checked using many different test-sequences.

The idea of dynamic transition relations seems to be very useful, and a similar result may be obtained for asynchronous models. Such a result would be practical for both asynchronous hardware designs as well as parallel programs.

Bibliography

[1] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. V. Bochmann and D. K. Probst, editors, *Proceedings of the Fourth Workshop on Computer-Aided Verification*, pages 251–263, July 1992.

[2] V. Boppana, S. Rajan, K. Takayama, and M. Fujita. Model checking based on sequential ATPG. In *Computer Aided Verification*, 1999.

[3] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. State space reduction based on live variables analysis. In *Static Analysis Symposium*, Venezia, Italy, September 1999.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2), July 1988.

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

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[7] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. LNCS 630, pages 123–137. Springer, 1992.

[8] Olaf Burkart and Bernhard Steffen. Pushdown processes: Parallel composition and model checking. LNCS 836, pages 98–113. Springer, 1994.

[9] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.

[10] S. Campos, E. M. Clarke, and O. Grumberg. Selective quantitative analysis and interval model checking: verifying different facets of a system. In *Computer Aided Verification*, 1996. To appear in Formal Methods in System Design.

[11] S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.

[12] K. Chakrabarty, B. Myrray, and V. Iyengar. Built-in test pattern generation for high-performance circuits using twisted-ring counters. In *IEEE VLSI Test Symposium*, 1999.

[13] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[14] E. M. 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, 1986.

[15] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16, 5:1512–1542, September 1994.

[16] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1989.

[17] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.

[18] Dennis Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, Eindhoven, Holland, 1996.

[19] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.

[20] D.R. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *IFIP working conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, San Miniato, Italy, June 1994.

[21] D. Dolev, M. Klawe, and M. Rodeh. An O(nlogn) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms, volume 3*, pages 245–260, 1992.

[22] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.

[23] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 997–1072. Elsevier Science Publishers, 1990.

[24] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. pages 84–96, January 1985.

[25] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *TACAS*, 1997.

[26] F. Beenker et. al. A testability strategy for silicon computers. In *Proc. Int. Test Conf.*, 1989.

[27] R.W. Floyd. Assigning meanings to programs. In *Proc. AMS Sump. Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, Providence, R.I., 1967.

[28] N. Francez. *Program Verification.* Addison-Wesley, 1992.

[29] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principle of Programming Languages*, January 1997.

[30] Patrice Godefroid. *Partial order methods for the verification of concurrent systems.* PhD thesis.

[31] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[32] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editors, 1991.

[33] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. of FORTE 1994 Conference, Bern, Switzerland*, 1994.

[34] C.B. Jones. Specification and design of (parallel) programs. In *Proc. of IFIP'83*, pages 321–332. North-Holland, 1983.

[35] B. Josko. Modelchecking of CTL formulae under liveness assumptions. In Thomas Ottman, editor, *Automata, Languages and Programming*, volume 267 of *Lecture Notes in Computer Science*, Berlin etc., 1987. Springer-Verlag.

[36] S. Katz and H. Miller. Saving space by fully exploiting invisible transitions. *Formal Methods in System Design*, 14(3):311–332, May 1999.

[37] P. Kelb. Model checking and abstraction: A framework preserving both truth and failure information, 1994. OFFIS, Oldenburg, Germany.

[38] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In B. Steffen, editor, *Proc. of TACAS'98, LNCS 1384*, pages 335–357, 1998.

[39] J. Lee and J. H. Patel. An architectural level test generator for a hierarchical design environment. In *Proc. 21st Fault-Tolerant Computing Symp.*, 1991.

[40] J. Lee and J. H. Patel. An architectural level test generator for a hierarchical design environment based on nonlinear equation solving. In *Journal of Electronic Testing*, 1993.

[41] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, January 1985.

[42] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–45, 1995.

[43] D. E. Long. A bdd library, carnegie-mellon university. available at: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/modck/pub/www/bdd.html.

[44] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[45] R. Milner. An algebraic definition of simulation between programs. In *In proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

[46] Murphi description languange and verifier. The URL for the home page of murphi is: http://sprout.stanford.edu/dill/murphi.html.

[47] Brian Murray. *Hierarchical Testing using Precomputed Tests for Modules.* PhD thesis, Computer Science and Engineering, University of Michigan, 1994.

[48] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis.* Springer, 1999.

[49] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.

[50] D. Peled. Ten years of partial order reduction. In A. J. Hu and M. Y. Vardi, editors, *Proc. of the 10th International Conference on Computer Aided Verification, LNCS 1427*, 1998.

[51] A. Pnueli. The temporal logic of concurrent programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, 1977.

[52] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F.* Springer-Verlag, 1984.

[53] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[54] C-C su and C. R. Kime. Multiple path sensitization for hierarchical circuit testing. In *Proc. Int. Test Conf.*, 1990.

[55] A. Valmari. A stubborn attack on the state explosion problem. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 1990 Workshop on Computer-Aided Verification*, June 1990.

[56] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of CUNCUR'93*, volume 715 of *LNCS*, pages 233–246. Springer Verlag, 1993.