Implementation and Evaluation of an Interpolation Based Model Checker

LARS HELANDER

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2007
Implementation and Evaluation of an Interpolation Based Model Checker

LARS HELANDER

Master’s Thesis in Computer Science (20 credits) at the School of Computer Science and Engineering Royal Institute of Technology year 2007
Supervisor at CSC was Karl Meinke
Examiner was Johan Håstad

TRITA-CSC-E 2007:004
ISRN-KTH/CSC/E--07/004--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se
Abstract

A Craig interpolant \( C \) of an unsatisfiable formula \( A \land B \) is a formula over the common variables of \( A \) and \( B \) such that (1) \( A \) implies \( C \) and (2) \( C \) is inconsistent with \( B \). The existence of \( C \) is guaranteed in the context of propositional logic.

In this Master’s thesis, a complete interpolation and SAT-based symbolic model checking algorithm is described. The algorithm uses resolution refutations of unsatisfiable bounded model checking instances to compute Craig interpolants that can be used as over-approximations of the traditional image operator used in symbolic reachability analysis.

As part of the Master’s thesis, the interpolation based algorithm has been implemented and evaluated. The report includes benchmarks from the implementation, and it is shown that it compares favorably to traditional methods on many instances.

Additionally, using saturation in combination with the interpolation based algorithm is shown to reduce the run time by one or two orders of magnitude on many benchmarks. This is probably the first time saturation and interpolation based model checking has been combined.
Referat

Implementering och utvärding av en interpolationsbaserad model checker

En Craig interpolant $C$ för en osatisfierbar formel $A \land B$ är en formel över de gemensamma variablerna för $A$ och $B$ sådan att (1) $A$ medför $C$ och (2) $C$ motsäger $B$. $C$'s existens är garanterad när $A$ och $B$ är satslogiska formler.

I detta examensarbete ges en beskrivning av en komplet interpolations- och SAT-baserad model checking algoritm. Algoritmen använder resolutionbevis för osatisfierbara probleminstanser av bounded model checking för att beräkna Craig interpolanter som kan användas som överapproximationer av den traditionella image-operatorn som används i symbolisk reachability analysis.

Som en del av examensarbetet har den interpolationsbaserade algoritmen implementerats och utvärderats. Denna rapport innehåller benchmarks som visar att interpolationsmetoden är jämförelsevis bättre än traditionella metoder på många exempel.

Till yttermera visas att mättning i kombination med den interpolationsbaserade algoritmen reducerar körtiden en eller två tiopotenser på många benchmarks. Detta är troligen första gången mättning och interpolationsbaserad model checking har kombinerats.
Acknowledgements

I would like to thank Prover Technology AB for providing the subject matter of this Master’s thesis, as well as giving me a place to do the work. Special thanks also to my supervisor at Prover, Gunnar Smith, whose help has been much appreciated.
Contents

1 Introduction ........................................... 1
   1.1 General Introduction ............................ 1
   1.2 Formal Verification .............................. 2
   1.3 Preliminaries .................................... 3
      1.3.1 Satisfiability Solving ........................ 3
      1.3.2 Quantified Boolean Formulas .................. 3
   1.4 This Thesis ..................................... 3

2 Logical Foundations .................................. 5
   2.1 Interpolation .................................... 5
   2.2 Interpolation and Resolution ..................... 6
   2.3 Summary ......................................... 10

3 Model Checking ...................................... 11
   3.1 Background ..................................... 11
   3.2 Modeling Systems ............................... 13
   3.3 Symbolic Model Checking ........................ 14
   3.4 Summary ........................................ 18

4 Interpolation Based Model Checking ................. 19
   4.1 Model Checking Based on Interpolation .......... 19
   4.2 An Algorithm for Interpolation Based Model Checking 21
   4.3 Implementation .................................. 23
   4.4 Optimisations .................................. 25
   4.5 Evaluation ..................................... 26

5 Conclusions and Future Work ......................... 31

Bibliography ....................................... 33
Chapter 1

Introduction

Chapter 1 introduces the area of formal verification and model checking. This provides necessary background for motivating the work done in this thesis.

1.1 General Introduction

Few things can be as costly as errors, or bugs, in computer systems. Nowadays computers are used everywhere, and they can be found in almost every safety-critical system. One advantage of using computers in such systems is because they eliminate the risk of mistakes caused by humans. However, if they do not work correctly, the consequences could be fatal. Consider for example the computers that run nuclear power plants, or control space crafts or railway interlocking systems. If they do not work as intended, it would not only be very costly, but it could also put human lives in a considerable danger.

There are a number of famous incidents that can be mentioned in this context [23]:

- On October 30, 1994, a professor at Lynchburg College reports a bug in the floating point unit of Intel’s Pentium processor. Apparently, the processor’s floating point division instruction produced inaccurate results on certain input data. As it turns out, the error was caused by missing entries in a lookup table used by the division algorithm. The bug reportedly cost Intel 470 million dollars.

- On June 4, 1996, the space craft Ariane 5 flight 501 self destructs 37 seconds after launch due to a software malfunction. According to the report by the inquiry board, the error was caused by a data conversion from a 64-bit floating point value to a 16-bit integer value, yielding an arithmetic overflow which triggered a series of failures.

- On February 25, 1991, the US missile defense system called “MIM-104 PATRIOT” failed to intercept an Iraqi Scud missile that hit an army barrack in
CHAPTER 1. INTRODUCTION

Dharan, Saudi Arabia, killing 28 American soldiers. The failed intercept was caused by a software bug in the system’s clock, causing it to drift by one third of a second, equivalent to a position error of 600 meters on the fast moving Iraqi missile.

Apparently, the incentives for verification of computer systems are substantial. Testing has been, and still is, the most common verification procedure, but it has obvious flaws. In a realistic system, testing can never fully cover all possibilities. The hope is then, that formal verification, which will be introduced in the following section, can remedy this flaw.

1.2 Formal Verification

Formal verification is the common name for a number of techniques for verifying systems formally. The techniques are formal in the sense that they are all based on some mathematical theory, e.g. propositional logic or graph theory. By verifying, we mean checking that whatever we are building conforms to its specification, i.e. it does what it is supposed to do. [18] is a good introduction to formal verification of software.

Now, assume that our specification is a text document written in English or some other natural language, and our system consists of 10000 lines of C code, how can we possibly in a mathematically strict way verify that the code fulfills the specification? The answer is that we have to formulate the specification in a mathematical language and also often create an abstract model of the system (in this case the code) that filters away irrelevant parts, while still capturing, on a mathematical format, the “essence” of the system. This is certainly not a trivial task, but fortunately it is something that lies outside the scope of this thesis. However, this is the reason why formal methods has come further in digital hardware verification than in high level software verification, since hardware circuits can be modeled quite easily in propositional logic.

Model checking [5] is a formal technique for automatic verification of systems modeled as finite-state machines. The specification is often expressed as a formula in linear temporal logic, LTL. In this thesis, we will focus on model checking of safety properties. In LTL, a safety property \( p \) would be written as \( Gp \), which means that \( p \) shall always be true. In other words, a safety property should express something that is desired to hold at all times.

Traditional model checking traverses an explicit representation (i.e. a graph with lists of nodes and edges) of the finite-state machine in order to verify properties. One of the main drawbacks of this approach is called state explosion problem. The number of states in the transition system can be extremely large and to traverse it is not feasible. One approach to get a more compact representation is to model the system in some logic, for example propositional logic. This is called symbolic model checking [14]. Traditionally binary decision diagrams (BDDs) have been used in
symbolic model checking, but recently many algorithms based on SAT-solvers (see Section 1.3.1) have emerged.

1.3 Preliminaries

This text assumes that the reader has some basic knowledge in logic. For example, the binary connectives $\land$ (and), $\lor$ (or), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence) should be well known, as well as negation ($\neg$). We will use $\top$ and $\bot$ to denote the values true and false, respectively. Previous experience of model checking would also be helpful, although it is not a requirement. However, there are some notions that the reader should understand before reading this thesis, for example satisfiability solving and quantified boolean formulas.

1.3.1 Satisfiability Solving

Propositional satisfiability (SAT) solving is the problem of finding a satisfying assignment to a propositional formula, or a proof that no such assignment exists. A satisfying assignment is an assignment of truth-values to the variables of the formula such that it makes the formula true. If no satisfying assignment exists, the formula is said to be unsatisfiable, contradictory or inconsistent. As an example, the formula $(a \lor b) \land (\neg a \lor c) \land \neg c$ has exactly one satisfying assignment, which is $\langle a \leftarrow \bot, b \leftarrow \top, c \leftarrow \bot \rangle$. Note that if a formula $f$ is unsatisfiable, then the negation of the formula $\neg f$ is a tautology.

The satisfiability problem is known to be NP-complete [6], which means we probably cannot solve it in reasonable time in the worst case. However, automatic solvers of the satisfiability problem, called SAT-solvers, based on Stålmarck’s method [22] or the Davis-Putnam-Logemann-Loveland procedure [9, 8], have proved that they can handle industrial verification examples with thousands of variables with relative ease.

1.3.2 Quantified Boolean Formulas

A quantified Boolean formula (QBF) is a Boolean formula where at least one variable is bound by an existential ($\exists$) or universal ($\forall$) quantifier. For example, the QBF $\exists x. x \land y$ should be read as “for some value of $x$, $x \land y$ holds”. This formula reduces to $y$. By contrast, $\forall x. x \land y$ should be read as “for all values of $x$, $x \land y$ holds”. Naturally, this formula reduces to $\bot$, because for $x \equiv \bot$, $x \land y \equiv \bot$. In general, the identities $\exists x. \phi(x) \equiv \phi(\top) \lor \phi(\bot)$ and $\forall x. \phi(x) \equiv \phi(\top) \land \phi(\bot)$ can be used to remove the quantifiers from a QBF.

1.4 This Thesis

The purpose of this Master’s thesis is to describe, implement and evaluate an interpolation and SAT-based model checking procedure. We will only consider the
problem of checking safety properties of sequential systems, a problem widely studied. This document is organized as follows. First, in chapter 2 we explain what the term *interpolation*, in the field of logic, really means. We also describe the *resolution* proof system, and show how resolution proofs of unsatisfiable formulas can be used to derive interpolants. In chapter 3 we introduce the area of model checking in general, and symbolic model checking in particular. Then, in chapter 4, we combine the knowledge from the previous chapters and show how interpolants derived from resolution proofs can be used to allow fast and fully SAT-based symbolic model checking. The chapter also includes a brief description of the implementation of the procedure, and an evaluation of the actual performance on real industrial benchmarks. Finally, chapter 5 concludes the thesis, and some suggestions for future work are proposed.
Chapter 2

Logical Foundations

*Chapter 2 explains the basic logic notions interpolation and resolution that the interpolation based model checking algorithm rests on.*

2.1 Interpolation

Craig’s interpolation theorem [7] is a basic result in logic. It says that whenever \( A \rightarrow B \) holds, there exists an *interpolant* \( C \) such that \( A \rightarrow C \) and \( C \rightarrow B \). Furthermore, the formula \( C \) refers only to the common variables of \( A \) and \( B \). The theorem holds for many kinds of logic, but we will only consider the case of propositional logic.

**Example** Consider the following propositional formulas.

\[ A = a \land b \text{ and } B = b \lor c. \]

Clearly, \( A \) implies \( B \), and the interpolant becomes \( b \), because \( a \land b \rightarrow b \) and \( b \rightarrow b \lor c \).

It is trivial to prove that there always exists an interpolant in the case of propositional logic [10]. To see this assume that \( A \) implies \( B \) and let \( A = A(p_1, \ldots, p_n) \).

If some \( p_i \), for example, \( p_1 \), does not occur in \( B \), then \( A(\top, p_2, p_3, \ldots, p_n) \rightarrow B \) and \( A(\bot, p_2, p_3, \ldots, p_n) \rightarrow B \). However, since \( A \) implies \( A(\top, p_2, p_3, \ldots, p_n) \lor A(\bot, p_2, p_3, \ldots, p_n) \) and \( A(\top, p_2, p_3, \ldots, p_n) \lor A(\bot, p_2, p_3, \ldots, p_n) \) implies \( B \), the formula \( A(\top, p_2, p_3, \ldots, p_n) \lor A(\bot, p_2, p_3, \ldots, p_n) \) is an interpolant. This formula could also be written more succintly as the QBF \( \exists p_1. A(p_1, \ldots, p_n) \). Of course the formula for the interpolant could be reduced to just \( \top \) or \( \bot \), but in the first case, \( B \), being implied by \( \top \), would be tautological, and in the second case \( A \), implying \( \bot \), would be contradictory. So if neither \( B \) is tautological nor \( A \) is contradictory, the interpolant becomes a well formed formula in propositional logic. Note that the size of this formula can be exponential in the number of variables of \( A \), in the worst case.

If more than one variable fails to occur in \( B \), then repetition of the method will yield a valid interpolant. On the other hand, if all variables of \( A \) occur in \( B \), then the interpolant could simply be chosen to be \( A \) (because \( A \) implies \( A \)). Also, note
CHAPTER 2. LOGICAL FOUNDATIONS

that all variables local to \( B \) (i.e. that does not occur in \( A \)) will of course not be included in the interpolant.

**Example** Returning to the propositional formulas in the previous example:

\[ A(a, b) = a \land b \text{ and } B(b, c) = b \lor c. \]

Since \( a \) does not occur in \( B \), the formula for the interpolant becomes \( \exists a. (a \land b) = (\top \land b) \lor (\bot \land b) \) which of course is equivalent to just \( b \).

In the following, we will sometimes write \( A(\vec{p}, \vec{q}) \), \( B(\vec{p}, \vec{r}) \) and \( C(\vec{p}) \) to denote the formulas \( A, B \) and their interpolant \( C \), respectively. Here, \( \vec{p}, \vec{q} \) and \( \vec{r} \) are disjoint vectors of propositional variables and \( \vec{p} \) are the variables common to \( A \) and \( B \). The intuition about interpolation is thus that since \( B \) does not contain any variables from \( \vec{q} \), whatever \( A \) says about \( \vec{p} \) should be sufficient to imply \( B \). This information is encapsulated in the interpolant.

### 2.2 Interpolation and Resolution

Resolution is a proof system for propositional logic based on a single inference rule, the resolution rule:

\[
\frac{\Gamma \lor p \quad \Delta \lor \neg p}{\Gamma \lor \Delta}
\]

The application of this rule is called *resolution on variable* \( p \), where \( p \) is called the *pivot variable* and the clause \( \Gamma \lor \Delta \) is called the *resolvent* of \( \Gamma \lor p \) and \( \Delta \lor \neg p \). The goal of resolution is to derive the empty clause, which we will denote \( \bot \), by resolving on clauses \( p \) and \( \neg p \). Clearly, \( p \land \neg p = \bot \).

Before we continue, we need to introduce the basic terminology that is used in this section:

- A *variable* is just an ordinary propositional variable, like \( a, b \) or \( p \).
- A *literal* is a variable or the negation of a variable, e.g. \( p \) or \( \neg p \).
- A *clause* is a disjunction of zero or more literals, for example \( a \lor \neg b \lor \neg c \). We assume that no clause contains a variable and its negation, i.e. all clauses are non-tautological. As mentioned above, the empty clause will be denoted \( \bot \). In some literature, a clause is written as a set of literals.
- A formula in *conjunctive normal form*, \( \text{CNF} \) is a conjunction of clauses, often called a clause *set*. We will write \( c \in C \) to denote that the clause \( c \) is a clause in the CNF formula \( C \).

Resolution is used to prove a formula \( \phi \) in *disjunctive normal form*, \( \text{DNF} \) (which is a disjunction of conjunctions) by refuting the CNF formula \( \neg \phi \) (to be precise, \( \neg \phi \) is
2.2. INTERPOLATION AND RESOLUTION

not in CNF but it will become a CNF formula by repeatedly applying DeMorgan's law to it). By refuting we mean deriving the empty clause by a sequence of resolution inferences. The result is called a resolution refutation or resolution proof and is a directed acyclic graph, DAG where the vertices are clauses and a vertex \( c \) that is not a root has exactly two predecessors, \( c_1 \) and \( c_2 \), that \( c \) is the resolvent of. The roots of the resolution proof are clauses from the original formula \( \neg \phi \) and the empty clause is the unique leaf.

**Example** Prove that the clause sets \( A = b \land (\neg b \lor c) \) and \( B = (\neg c \lor d) \land \neg d \) are mutually unsatisfiable, i.e. \( A \land B = \bot \).

![Figure 2.1. A resolution refutation.](image)

A proof that \( A \land B = \bot \) is given as a resolution refutation \( P = (V, E) \) in Figure 2.1.

Now, to see how resolution can be used to derive an interpolant, consider the CNF formula \( A \land B \). If we can produce a resolution refutation for \( A \land B \), it means that \( \neg (A \land B) \), which is equivalent to \( A \rightarrow \neg B \), holds. Then we know that there exists an interpolant \( C \) such that \( A \rightarrow C \) and \( C \rightarrow \neg B \) hold. But the resolution proof gives us even more; it gives us a way to directly construct the interpolant [13, 20]. The procedure, which we will now describe, actually constructs a Boolean circuit whose gates correspond to the vertices in the proof. The procedure takes time that is linear in the size of the resolution proof, unfortunately however, the size of the proof can be exponential, in the number of variables, in the worst case [11].

**Remark** Whenever we write that \( C \) is an interpolant for \( A \land B \), what we really mean is that \( C \) is an interpolant for \( A \rightarrow \neg B \), i.e. the negation of \( A \land B \).

**Definition** Let \( P = (V, E) \) be a resolution refutation of \( A(\bar{p}, \bar{q}) \land B(\bar{p}, \bar{r}) \). For each vertex \( v \in V \), let \( C_v \) be a Boolean formula, such that

- if \( v \) is a root, then
  - if \( v \in A(\bar{p}, \bar{q}) \) then \( C_v \) is the constant \( \bot \),
  - else (if \( v \in B(\bar{p}, \bar{r}) \)) \( C_v \) is the constant \( \top \).
• else
  \[ v = F \lor G \] and the predecessors of \( v \) are \( v_1 = F \lor p_i \) and \( v_2 = G \lor \neg p_i \)
  where \( p_i \in \bar{p} \), then \( C_v = (\neg p_i \land C_{v_1}) \lor (p_i \land C_{v_2}) \),
  \[ \text{else if } v = F \lor G \] and the predecessors of \( v \) are \( v_1 = F \lor q_i \) and \( v_2 = G \lor \neg q_i \)
  where \( q_i \in \bar{q} \), then \( C_v = C_{v_1} \lor C_{v_2} \),
  \[ \text{else, } v = F \lor G \] and the predecessors of \( v \) are \( v_1 = F \lor r_i \) and \( v_2 = G \lor \neg r_i \)
  where \( r_i \in \bar{r} \), then \( C_v = C_{v_1} \land C_{v_2} \).

**Theorem 2.2.1** \( C_\bot \) is an interpolant for \( A(\bar{p}, \bar{q}) \land B(\bar{p}, \bar{r}) \). (Remember that \( \bot \) is the unique leaf vertex of \( P \)).

**Proof** We will prove the theorem by first proving Lemma 2.2.2. The proof is due to [3] (although it contained some minor errors) and the idea is to consider a truth assignment \( \tau \) to the variables of \( A \) and \( B \).

If we can prove that for any such \( \tau \), if \( \tau(C_\bot) = \bot \) then \( \tau(A(\bar{p}, \bar{q})) = \bot \) (in other words \( \neg C_\bot \rightarrow \neg A(\bar{p}, \bar{q}) \equiv A(\bar{p}, \bar{q}) \rightarrow C_\bot \) and if \( \tau(C_\bot) = \top \) then \( \tau(B(\bar{p}, \bar{r})) = \bot \) (or equivalently \( C_\bot \rightarrow \neg B(\bar{p}, \bar{r}) \)), then we have proved that \( C_\bot \) is in fact an interpolant for \( A(\bar{p}, \bar{q}) \land B(\bar{p}, \bar{r}) \) since \( C_\bot \) according to the method that constructed it only contains the common variables of \( A(\bar{p}, \bar{q}) \) and \( B(\bar{p}, \bar{r}) \), i.e. \( C_\bot = C_\bot(\bar{p}) \).

**Lemma 2.2.2** If \( \tau \) is a truth assignment such that \( \tau(v) = \bot \) for some \( v \in V \), then

\[
\begin{align*}
\tau(C_v) = \bot & \implies \tau(a) = \bot \text{ for some clause } a \in A \\
\tau(C_v) = \top & \implies \tau(b) = \bot \text{ for some clause } b \in B
\end{align*}
\]

**Proof** We will prove the lemma by induction. The base case is when \( v \) is a root.

**Base:**

1. If \( v \in A(\bar{p}, \bar{q}) \) then \( C_v = \bot \) by definition and therefore \( \tau(C_v) = \bot \) is certainly true. Also, the hypothesis of the lemma is that \( \tau(v) = \bot \) so this case is trivially true.

2. Else \( v \in B(\bar{p}, \bar{r}) \) and \( C_v = \top \) by definition. The case is otherwise similar to 1.

**Step:**

1. \( v = F \lor G \) and the predecessors of \( v \) are \( v_1 = F \lor p_i \) and \( v_2 = G \lor \neg p_i \) where \( p_i \in \bar{p} \).

   By definition: \( C_v = (\neg p_i \land C_{v_1}) \lor (p_i \land C_{v_2}) \).

   Since \( \tau(v) = \bot \), we have that \( \tau(F) = \tau(G) = \bot \).

   If \( \tau(C_v) = \bot \), then
2.2. INTERPOLATION AND RESOLUTION

a) Case $\tau(p_i) = \top$, then $\tau(C_{v_2}) = \bot$ and $\tau(v_2) = \bot$. By the induction hypothesis, we can conclude that $\tau(a) = \bot$ for some $a \in A$.

b) Case $\tau(p_i) = \bot$, then $\tau(C_{v_1}) = \bot$ and $\tau(v_1) = \bot$. By the induction hypothesis, $\tau(a) = \bot$ for some $a \in A$.

Else if $\tau(C_v) = \top$, then

a) Case $\tau(p_i) = \top$, then $\tau(C_{v_2}) = \top$ and $\tau(v_2) = \bot$. By the induction hypothesis, $\tau(b) = \bot$ for some $b \in B$.

b) Case $\tau(p_i) = \bot$, then $\tau(C_{v_1}) = \top$ and $\tau(v_1) = \bot$. By the induction hypothesis, $\tau(b) = \bot$ for some $b \in B$.

2. $v = F \lor G$ and the predecessors of $v$ are $v_1 = F \lor q_i$ and $v_2 = G \lor \neg q_i$ where $q_i \in \bar{q}$.

By definition: $C_v = C_{v_1} \lor C_{v_2}$.

Since $\tau(v) = \bot$, we have that $\tau(F) = \tau(G) = \bot$.

If $\tau(C_v) = \bot$, then $\tau(C_{v_1}) = \tau(C_{v_2}) = \bot$.

a) Case $\tau(q_i) = \top$, then $\tau(v_2) = \bot$. By the induction hypothesis, $\tau(a) = \bot$ for some $a \in A$.

b) Case $\tau(q_i) = \bot$, then $\tau(v_1) = \bot$. By the induction hypothesis, $\tau(a) = \bot$ for some $a \in A$.

The other case, where $\tau(C_v) = \top$, is not as simple. We know however that at least one of $\tau(C_{v_1})$ and $\tau(C_{v_2})$ is true, because of the definition of $C_v$. Also, at least one of $v_1$ and $v_2$ is true (obviously, since $v_1 = F \lor q_i$ and $v_2 = G \lor \neg q_i$).

This gives us nine cases as depicted in Table 2.1.

<table>
<thead>
<tr>
<th>$\tau(v_1)$</th>
<th>$\tau(v_2)$</th>
<th>$\tau(C_{v_1})$</th>
<th>$\tau(C_{v_2})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

Table 2.1. The possible truth assignments to $v_1$, $v_2$ and $C_{v_1}$, $C_{v_2}$.

The cases (2), (4), (7) and (8) are all trivially true, because either $\tau(v_1) = \bot$ and $\tau(C_{v_1}) = \top$ or $\tau(v_2) = \bot$ and $\tau(C_{v_2}) = \top$. In either case, we can conclude by the induction hypothesis that $\tau(b) = \bot$ for some $b \in B$. 

9
CHAPTER 2. LOGICAL FOUNDATIONS

We approach the remaining cases noting that neither $C_{v_1}, C_{v_2}$ nor $B(\bar{p}, \bar{r})$ contains the pivot variable $q_i$ (in fact, they don’t contain any $q$-variable). Therefore, we define

$$\tau'(q_i) = \begin{cases} \bot & \text{if } \tau(q_i) = \top, \\ \top & \text{if } \tau(q_i) = \bot \end{cases} \quad \text{and } \tau' = \tau \text{ on all other variables.}$$

Now, for cases (5), (6) and (9) we have that $\tau(v_1) = \top$ and $\tau(C_{v_1}) = \top$. Since $\tau'(v_1) = \bot$ (remember that $v_1 = F \lor q_i$ and $\tau(F) = \bot$) and $\tau'(C_{v_1}) = \top$, we can conclude by the induction hypothesis that $\tau'(b) = \bot$ for some $b \in B$. As noted however, $\tau'(b) = \tau(b)$ so therefore it also holds that $\tau(b) = \bot$ for some $b \in B$.

For the remaining cases, (1) and (3), we have that $\tau(v_2) = \top$ and $\tau(C_{v_2}) = \top$. By the same reasoning, we can conclude that $\tau(b) = \bot$ for some $b \in B$.

3. $v = F \lor G$ and the predecessors of $v$ are $v_1 = F \lor r_i$ and $v_2 = G \lor \neg r_i$ where $r_i \in \bar{r}$.

This is the dual case of (2) and can be proven in a similar manner.

This completes the proof of Lemma 2.2.2 and also of Theorem 2.2.1, since $\tau(\bot) = \bot$ for all $\tau$. □

2.3 Summary

In this chapter, we have defined a Craig interpolant over an unsatisfiable set of clauses $A(\bar{p}, \bar{q}) \land B(\bar{p}, \bar{r})$ as a formula $C(\bar{p})$ such that:

1. $A(\bar{p}, \bar{q})$ implies $C(\bar{p})$,

2. $C(\bar{p})$ is inconsistent with $B(\bar{p}, \bar{r})$, and

3. $C(\bar{p})$ is expressed over the common variables of $A(\bar{p}, \bar{q})$ and $B(\bar{p}, \bar{r})$.

We have also shown that, at least for propositional logic, there always exists an interpolant for two mutually unsatisfiable sets of clauses. The strongest interpolant can be found by quantifying away the variables local to $A$, i.e. $\exists \bar{q}.A(\bar{p}, \bar{q})$ is the strongest interpolant. We have also shown how an (other) interpolant can be derived from a resolution refutation $P = (V, E)$ in linear time.
Chapter 3

Model Checking

Chapter 3 goes into the details of model checking, especially symbolic model checking.

3.1 Background

The formal verification technique model checking was pioneered in 1981 by Edmund Clarke and Allen Emerson at Carnegie Mellon University [4]. Using a state transition graph to represent the system model, they devised an algorithm for verifying formulas expressed in the temporal logic CTL (computational tree logic). In a temporal logic like CTL, it is possible to express properties that can vary over time, by using the usual logical operators like $\neg$, $\land$ and $\lor$, combined with certain tense operators like $G$ ("always"), $F$ ("eventually") and $X$ ("next"). As an example of a temporal logic formula, $GFp$ states that it is always the case that the proposition $p$ eventually becomes true. This is the same as saying that $p$ will be true infinitely often. The CTL logic, and its more general version called CTL*, also allows the path quantifiers $A$ and $E$. Informally, $A$ applied to some formula $f$ says that $f$ shall hold for every possible computation path of the system. By contrast, $E$ applied to $f$ says that $f$ must hold for some computation path.

The state transition graph used to represent the system were called Kripke structure for historical reasons. A Kripke structure consists of a set of states – of which some are initial states – a set of transitions between states and a function that labels states with the atomic propositions that are true in that state. Formally, a Kripke structure $M$ is a tuple $M = (S, I, T, I)$ where

1. $S$ is a finite set of states.
2. $I \subseteq S$ is the set of initial states.
3. $T \subseteq S \times S$ is a binary relation on states, called transition relation. The relation must be total, i.e. every state must have a successor (we must never reach a dead end).
4. $l : S \rightarrow 2^\mathcal{AP}$, where $\mathcal{AP}$ is a set of atomic propositions, is a function that labels each state with the set of atomic propositions true in that state.

**Example** Consider the following simple program:

```plaintext
x := 0;
while true do
    x := (x + 1) mod 2
od
```

One straightforward translation of this system into a Kripke structure is shown in Figure 3.1. The figure shows the graphical representation of the Kripke structure $M = (S, I, T, l)$ where $S = \{0, 1\}$, $I = \{0\}$, $T = \{(0, 1), (1, 0)\}$ and $l(0) = \{x = 0\}, l(1) = \{x = 1\}$.

The first model checking algorithms used the *explicit* representation of the Kripke structure (i.e. as a graph with lists of nodes and edges) to search for models and counter-models. All of these algorithms suffer from a problem called the *state explosion* problem, due to the fact that the number of states of the system model grows exponentially in the number of system variables. For example, if our system only consists of Boolean variables, adding just one variable to the system will double the number of states of the system model.

Luckily, it is possible to represent the Kripke structure *implicitly* using formulas in first order logic. If we define the state space of the Kripke structure to be all possible valuations of the system variables then we could let a first order formula $I$ over those variables represent the initial states, in the sense that exactly those states (i.e. valuations of the system variables) that makes the formula true belongs to $I$.

More precisely, if we let $V = \{v_1, \ldots, v_n\}$ be the set of system variables over some domain $D$ then the state space $S$ of the Kripke structure will be $S = D \times D \times \cdots \times D = D^n$. A state $s \in S$ is just a valuation $s : V \rightarrow D$. For example, if $D = \mathbb{N}$, $V = \{v_1, v_2\}$ and $I(V) \equiv v_1 \leq 5 \land v_2 = v_1$ then the state $s = \langle v_1 \leftarrow 1, v_2 \leftarrow 1 \rangle$ is an initial state. In a similar manner, we can represent the transition relation by a first order formula $T$. However, we first need to introduce a set $V' = \{v'_1, \ldots, v'_n\}$ of *next state* variables, because we need to be able to talk about the values of the system.
variables in both the present and the next state. \( \mathcal{T} \) then becomes a formula over \( V \) and \( V' \). If we rename \( x \) as \( v_1 \) in the example depicted in Figure 3.1, we could write \( \mathcal{T} \) as \( \mathcal{T}(V, V') \equiv v'_1 = (v_1 + 1) \mod 2 \) or equivalently \( \mathcal{T}(V, V') \equiv v'_1 = 0 \leftrightarrow v_1 = 1 \land v'_1 = 1 \leftrightarrow v_1 = 0 \).

In Section 3.3 we will show how we can solve the model checking problem in some cases by manipulating these formulas symbolically and thus avoid the state explosion problem. This is called symbolic model checking [14]. First however, we will show what kind of systems we are interested in verifying and how we can model these systems as Kripke structures.

### 3.2 Modeling Systems

When modeling a system, one needs to find a suitable level of abstraction. For example, when modeling a digital circuit, actual voltage levels of signals should not be of interest. Instead, one should settle with the abstraction that a signal can have one of two values, 1 and 0, as this will make it possible to model the circuit in propositional logic. For the interested reader, [5] includes a good overview of how different systems, like programs or asynchronous circuits, can be modeled. In this thesis, we will only consider synchronous sequential logic circuits.

A sequential logic circuit, like the one depicted in Figure 3.2, is built from unit delays and Boolean gates (by contrast, a circuit built entirely from Boolean gates is called a combinational logic circuit). If the unit delays, which we will call latches, are controlled by a global clock (that forces the latches to output their values synchronously every clock cycle), the circuit is a synchronous circuit. Thus, at every clock cycle, a synchronous circuit will assume a new state based on the values of the latches in the previous cycle and on the inputs to the circuit. In order for the circuit to have a well-defined state initially, every latch must be given an initial definition. In the circuit in Figure 3.2 the latch \( v_0 \) is initially 1, and the latch \( v_2 \) is initially 0.

The process of modeling synchronous circuits in propositional logic as Kripke structures in terms of an initial constraint \( \mathcal{I} \) and a transition constraint \( \mathcal{T} \) can be illustrated by deriving \( \mathcal{I} \) and \( \mathcal{T} \) from the circuit in Figure 3.2.

We first identify that the state of the circuit is uniquely determined by the values of the signals \( v_0, v_1 \) and \( v_2 \). Hence we let \( V = \{v_0, v_1, v_2\} \). As usual, we make a copy \( V' = \{v'_0, v'_1, v'_2\} \) of the system variables that will represent their values in the next state. We now define the transitions between the primed variables and their unprimed predecessors. The input signal \( v_1 \) is free to behave in any way, so we cannot put a transition constraint on it. The next value of \( v_0 \) however, is the logical and between \( v_0 \) and \( v_1 \). Hence \( v'_0 \leftrightarrow v_0 \land v_1 \). Similarly we can define \( v'_2 \) as \( v'_2 \leftrightarrow \sim v_1 \lor v_2 \). Since the circuit is synchronous, these transitions occur at the same time, hence \( \mathcal{T}(V, V') = (v'_0 \leftrightarrow v_0 \land v_1) \land (v'_2 \leftrightarrow \sim v_1 \lor v_2) \). As initial constraint, we simply conjunct the initial definitions of the latches. Since \( v_0 \) initially is 1 and \( v_2 \) initially is 0, we have that \( \mathcal{I}(V) = (v_0 \leftrightarrow \top) \land (v_2 \leftrightarrow \bot) \), or equivalently \( \mathcal{I}(V) = v_0 \land \neg v_2 \).

It is not difficult to extend this example to the general case. In the general
case, we would derive propositional formulas $f_i(V)$ for every state variable $v_i$ that is not an input, and let $\mathcal{T}(V, V') = \bigwedge_i v'_i \leftrightarrow f_i(V)$. Similarly, we would let $\mathcal{I}$ be the conjunction of the initial definitions $q_i$ of all latches, i.e. $\mathcal{I}(V) = \bigwedge_i v_i \leftrightarrow q_i$.

3.3 Symbolic Model Checking

We will begin this section by a small example of how we can prove properties by simple manipulation of Boolean formulas. First note, however, that in this section and the remainder of the thesis, we will only allow systems over Boolean variables, i.e. our domain $D = \{0, 1\}$ and our state space $S = \{0, 1\}^n$. This also makes for a more compact notation; as in Section 3.2 we can write just $x$ instead of $x = 1$ and $\neg x$ for $x = 0$.

**Example** Let $\mathcal{I}(x) = \neg x$ and $\mathcal{T}(x, x') = x' \leftrightarrow \neg x$. Prove that the property $AG(x \lor Xx)$ holds.

We first observe that the Kripke structure these formulas represent is equivalent to the one shown in Figure 3.1 in Section 3.1. The CTL$\ast$ formula $AG(x \lor Xx)$ says that for every computation path of the system, and at each state along such a path, either $x$ is true or in the next timestep, $x$ is true. This property obviously holds for this system, and to show this we will rewrite the CTL$\ast$ formula as a quantified Boolean formula (QBF) in terms of $\mathcal{T}$, $x$ and $x'$:

$$\forall x. \forall x'. (\mathcal{T}(x, x') \rightarrow x \lor x')$$

This formula should be interpreted as “for all states $x$ and $x'$, if there is a transition between $x$ and $x'$ then it is the case that $x$ or $x'$ holds”. In the general case, we would have quantified over a vector of variables, but since $x$ is the only variable of
3.3. SYMBOLIC MODEL CHECKING

the system, the state of the system is uniquely determined by the value of \( x \). By expanding the definition of \( T \) and using the identity \( \forall x.\phi(x) \equiv \phi(T) \land \phi(\bot) \) we can evaluate the formula as follows:

\[
\forall x.\forall x'.(T(x, x') \rightarrow x \lor x') \\
\equiv \forall x.((T \leftrightarrow \neg x \rightarrow x \lor T) \land (\bot \leftrightarrow \neg x \rightarrow x \lor \bot)) \\
\equiv (T \leftrightarrow \neg T \rightarrow T \lor T) \land (\bot \leftrightarrow \neg \bot \rightarrow \bot \lor \bot) \\
\equiv T
\]

To conclude matters, we arrive at the expected result, i.e. that the formula holds for this system. However, this informal technique cannot be applied in the general case for more realistic systems.

As the observant reader might have noticed, what we really did in the previous example was to prove the tautology \( x' \leftrightarrow \neg x \rightarrow x \lor x' \). Alternatively, we could have proved that the negation of the formula is contradictory. This can be done by propositional satisfiability testing, i.e. if the formula is unsatisfiable, then it is a contradiction. This is how satisfiability testing is used in symbolic model checking — by proving or disproving propositional formulas.

A property written on the form \( AGp \) where \( p \) is a propositional formula (possibly augmented with the next operator \( X \)) is called a safety property because it must be valid in every reachable state of the Kripke structure. A state in which the formula does not hold, is an unsafe state. We will also call unsafe states bad states. A state \( s \) is a bad state if and only if \( p(s) \) is false. A general strategy for proving a safety property \( p \) is simple induction, which proves \( p \) by proving that the formulas

\[
I(W_0) \rightarrow p(W_0) \tag{1} \\
T(W_k, W_{k+1}) \land p(W_k) \rightarrow p(W_{k+1}) \tag{2}
\]

are valid. If we can prove that the property \( p \) holds in the set of initial states (1), and that given a set of states where \( p \) holds, it also holds in the successor states (2), then we have indeed proved that \( p \) holds in every reachable state.

Although sound, simple induction is not complete, because for a given system there might exist unreachable states for which we cannot prove the step formula (2). As an example of this situation, consider the Kripke structure shown in Figure 3.3. Here \( I(V) = v_1 \land \neg v_2 \) and \( T(V, V') = v'_1 \leftrightarrow v_1 \land v'_2 \leftrightarrow \neg v_2 \). As can be seen in the figure, this Kripke structure has two states that cannot be reached from an initial state, and they are said to be unreachable. Now, assume that we would like to prove the property \( p(V) = v_1 \lor v_2 \) using simple induction. This property is valid for all states in the reachable part of the state space, but not for all states in the unreachable part. However, for one of the states in the unreachable part, the property holds, which means we cannot prove the step formula that for this system becomes \( (v'_1 \leftrightarrow \neg v_1 \land v'_2 \leftrightarrow \neg v_2) \land (v_1 \lor v_2) \rightarrow (v'_1 \lor v'_2) \). This formula has the counter-model \( \{v_1, v_2, \neg v'_1, \neg v'_2\} \) as expected.
The incompleteness of simple induction has resulted in a method called **induction with depth** which tries to prove a property by looking at paths of (unique) states along which the property holds, instead of just a single state. The hope is that there does not exist long paths in the unreachable part of the state space along which the property holds, followed by a state that violates the property. The different attempts to prove properties using induction will not be investigated further however, because it is not relevant for this thesis, other than as an example of an additional method of proving safety properties of sequential systems. The interested reader is instead referred to [21], which describes methods for proving safety properties using induction.

By contrast, an approach that may be more meaningful for understanding the interpolation based method, is **symbolic reachability analysis** [1]. The idea of this method is to create a formula $\mathcal{R}(V)$ representing the set of reachable states. If such a formula can be constructed, the problem of checking a safety property $p$ reduces to checking satisfiability of $\mathcal{R} \land \neg p$, in the sense that if the formula $\mathcal{R} \land \neg p$ is unsatisfiable, then we have proved that we cannot reach a state where $p$ does not hold. The formula $\mathcal{R}$, can be created by first computing formulas $r_k$ that represents the set of states reachable from an initial state in exactly $k$ transitions. These formulas can be computed iteratively as follows:

$$
r_0(W_0) = \mathcal{I}(W_0)$$

$$
r_{k+1}(W_{k+1}) = \exists W_k. \mathcal{T}(W_k, W_{k+1}) \land r_k(W_k)
$$

From these we can construct a formula $\mathcal{R}_n(V)$ that represents the set of states reachable from an initial states by at most $n$ transitions. $\mathcal{R}_n(V)$ is created simply by using the analogue of the set union operator – logical or – and dropping the time subscripts for the state variables:

$$
\mathcal{R}_n(V) = \bigvee_{k=0}^{n} r_k(V/W_k)
$$
3.3. SYMBOLIC MODEL CHECKING

The set \( \mathcal{R}_n(V) \) of reachable states will grow until it reaches a fixed point, at which time \( \mathcal{R}_n(V) = \mathcal{R}(V) \) (convergence is guaranteed for finite state systems, but not for infinite state systems). The fixed point will be detected when the implication \( \mathcal{R}_{n+1}(V) \rightarrow \mathcal{R}_n(V) \) holds, which means the last iteration did not add any new information to the set of reachable states.

At a first glance, reachability analysis might seem a bit complicated. To make things a bit clearer, let’s illustrate the method by computing the set of reachable states for the Kripke structure given in Figure 3.3. This Kripke structure has, as we have already noted, two unreachable states and two reachable ones. The latter can be represented by the formula \((v_1 \land \neg v_2) \lor (\neg v_1 \land v_2)\), that can be computed by fixed point iteration:

\[
\begin{align*}
\mathcal{R}_0(V) &= \mathcal{I}(V) \\
&= v_1 \land \neg v_2 \\
\mathcal{R}_{1}(V') &= \exists V. T(V, V') \land \mathcal{R}_0(V) \\
&= \exists (v_1, v_2), v'_1 \leftrightarrow \neg v_1 \land v'_2 \leftrightarrow \neg v_2 \land v_1 \land \neg v_2 \\
&= \exists v_1.(v'_1 \leftrightarrow \neg v_1 \land v'_2 \leftrightarrow \neg \top \land v_1 \land \neg \top) \lor \\
&\quad (v'_1 \leftrightarrow \neg v_1 \land v'_2 \leftrightarrow \neg \bot \land v_1 \land \neg \bot) \\
&= (v'_1 \leftrightarrow \neg \top \land v'_2 \leftrightarrow \neg \bot \land v_1 \land \neg \top) \lor \\
&\quad (v'_1 \leftrightarrow \neg \bot \land v'_2 \leftrightarrow \neg \bot \land v_1 \land \neg \bot) \\
&= \neg v'_1 \land v'_2
\end{align*}
\]

Carrying on, we will find that \( \mathcal{R}_2(V) = \mathcal{R}_0(V) \) and in general \( \mathcal{R}_{2i}(V) = v_1 \land \neg v_2 \) and \( \mathcal{R}_{2i+1}(V) = \neg v_1 \land v_2 \) for \( i \in \mathbb{N} \). Hence, a fixpoint will be reached after just two iterations and the formula for the reachable states becomes \( \mathcal{R}(V) = \mathcal{R}_0 \lor \mathcal{R}_1(V/V') = (v_1 \land \neg v_2) \lor (\neg v_1 \land v_2) \) as expected.

Another method of symbolic model checking of interest to us, is bounded model checking [2]. Whereas induction and reachability analysis tries to prove properties directly, bounded model checking is instead basically a search for counterexamples (“bugs”) of increasing length. The search can be terminated when the search depth has passed beyond the diameter of the system, which is defined as the longest path among the set of shortest paths between any two states. Determining the diameter of the system can be very tricky however, making bounded model checking an incomplete method in practice.

The search for counterexamples starts by satisfiability testing of the formula \( \mathcal{I}(V) \land \neg p(V) \). If a satisfying assignment is found for the formula, it is a counterexample for the property \( p \) of length 0, because no transitions are needed to reach a state where the property does not hold. The search continues with the depth incremented. At iteration \( k \), the formula \( \mathcal{I}(W_0) \land T(W_0, W_1) \land \ldots \land T(W_{k-1}, W_k) \land \neg p(W_k) \) is tested for satisfiability. Again, a satisfying assignment constitutes a counterexam-
ple of length \( k \), because it is a model of a path of \( k \) steps from an initial state to a bad state (a state where \( p \) does not hold).

3.4 Summary

In this chapter, we have introduced the automatic verification technique called *model checking*. Traditional model checking traverses a finite state-transition graph that represents the system, in order to prove properties expressed in a temporal logic like LTL or CTL.

In this thesis, the focus is on model checking of *safety properties*. A safety property is something that is desired to hold of each state in the reachable part of the state-transition graph.

By representing the state-transition graph symbolically using formulas in first order logic, symbolic techniques like SAT-solving can be used to check safety properties without explicit traversal of the state-transition graph. Of particular interest, *symbolic reachability analysis* proves safety properties by computing a formula that represents the set of reachable states, and *bounded model checking* looks for counterexamples by searching for satisfying assignments that models paths from initial states to bad states (i.e. states where the safety property does not hold).

A recent summary of different SAT-based approaches to model checking can be found in [19].
Chapter 4

Interpolation Based Model Checking

Chapter 4 describes how interpolation can be used in model checking. The interpolation based model checking procedure was invented in 2003 by Kenneth McMillan, and this chapter is based on his work [15, 16].

4.1 Model Checking Based on Interpolation

Let's approach the subject of interpolation based model checking by returning to the method of symbolic reachability analysis. The idea of reachability analysis is to construct a first order formula $\mathcal{R}(V)$, over the state variables, that represents the set of reachable states. The formula $\mathcal{R}(V)$ is computed by a fixpoint iteration that includes computing an exact forward image of the set of states reached in the previous iteration. We will define the forward image operator w.r.t. $\mathcal{T}$ of a set of states $\phi$ to be $\text{Img}(\phi) \overset{\text{def}}{=} \exists V.\mathcal{T}(V, V') \land \phi(V)$, i.e. the forward image is the set of states reachable in exactly one transition. Note that the forward image is over $V'$, i.e. $\text{Img} = \text{Img}(V')$. Computing the forward image is an expensive operation, as it involves quantifier-elimination, so what one would like to do instead is to find an over-approximation $\text{Img}'$ of the forward image operator such that $\text{Img} \rightarrow \text{Img}'$, that is easier to compute, but that is still strong enough to prove the property. Being an over-approximation means that all states that are included in $\text{Img}$, are also included in $\text{Img}'$. The latter, however, may contain additional states.

Now, suppose that we use the over-approximate image operator $\text{Img}'$ to compute an over-approximation of the set of reachable states, $\mathcal{R}'(V)$, in the usual way. If we can prove that this over-approximation does not include a bad state, i.e. the formula $\mathcal{R}'(V) \land \neg p(V)$ is unsatisfiable, then we have proved that a bad state cannot be reached at all – because of the over-approximation. However, if some of the additional states of the over-approximation (i.e. those that are not included in the set of reachable states), can reach a bad state, then we will not be able to prove the property $p$ even though it may be valid.

What one would like to do then is to carefully over-approximate the image operator relative the property $\neg p$, so that no states that can reach a bad state are
introduced in the over-approximation. We will call such an over-approximate image operator adequate. In other words, if a set of states $\phi$ cannot reach $\neg p$ in any number of steps, then an adequate over-approximate forward image $Img'(\phi)$ also cannot reach $\neg p$ in any number of steps. The problem, of course, is how to find an adequate image operator; after all, if we knew which states could reach a bad state, then we would not need model checking in the first place.

Let’s leave that question open for a second and instead consider the problem of bounded model checking described in Section 3.3. In bounded model checking, the existence of a counterexample of exactly $k + 1$ steps is posed as a satisfiability problem on the instance $\mathcal{I}(W_0) \land_{i=0}^{k} T(W_i, W_{i+1}) \land \neg p(W_{k+1})$. This formula is tested for satisfiability for increasing values of $k$, and any satisfying assignment is a counterexample of $k + 1$ steps. Alternatively, we could try to find counterexamples of all lengths from 1 to $k + 1$ directly by formulating the problem as $\mathcal{I}(W_0) \land_{i=0}^{k} T(W_i, W_{i+1}) \land \bigvee_{j=1}^{k+1} \neg p(W_{j})$.

So far this is nothing new, but suppose we break this formula in two parts, $A$ and $B$ so that

$$
A = \mathcal{I}(W_0) \land T(W_0, W_1), \text{ and }
B = \bigwedge_{i=1}^{k} T(W_i, W_{i+1}) \land \bigvee_{j=1}^{k+1} \neg p(W_{j}).
$$

If $A \land B$ is unsatisfiable, then we know that there does not exist any counterexamples of length $1 \ldots k + 1$. In addition, we know that there exists an interpolant $C$ over $W_1$ such that $A$ implies $C$ and $C$ is inconsistent with $B$. This interpolant can be derived in linear time from a resolution refutation of $A \land B$, provided $A \land B$ is first translated into conjunctive normal form (CNF). Now let’s look a little closer at the properties of the interpolant $C$. We know that $\mathcal{I}(W_0) \land T(W_0, W_1) \rightarrow C(W_1)$ holds. This means that there exists an interpolant $C'(W_1)$ such that $\mathcal{I}(W_0) \land T(W_0, W_1) \rightarrow C'(W_1)$ and $C'(W_1) \rightarrow C(W_1)$. This interpolant can be found by the naive method described in Section 2.1, i.e. by formulating it as the QBF $C'(W_1) = \exists W_0. \mathcal{I}(W_0) \land T(W_0, W_1)$. However, the interpolant $C'$, which is stronger than $C$ (if not $C' \equiv C$ of course), is the forward image of $\mathcal{I}$ w.r.t. $T$. Therefore, $C$ is an over-approximation of the forward image of $\mathcal{I}$.

Now as things hopefully are getting interesting, let’s return to the question of finding an adequate image over-approximation. Remember that an over-approximate image operator $Img'$ is adequate w.r.t. $\neg p$ if for any $\phi$ that cannot reach $\neg p$ in any number of steps, $Img'(\phi)$ cannot reach $\neg p$ in any number of steps either. Is our interpolant $C$, adequate? The answer is maybe. We know that $C$ cannot reach $\neg p$ in $k$ steps, because $C$ is inconsistent with $B$, so one might say that $C$ is $k$-adequate. However, if $k$ is not smaller than the diameter of the state space, then $k$-adequate becomes adequate, because by definition, any state that can be reached can be reached within no more steps than the diameter of the state space.

Thus, by repeatedly increasing $k$, we will either find a counterexample, or, when our over-approximate image becomes adequate, we will be able to prove the property
4.2. AN ALGORITHM FOR INTERPOLATION BASED MODEL CHECKING

procedure INTERPOL( $M = (I, T)$, $p$ )
    if SAT( $I \land \neg p$ ) return false
    let $k = 1$
    while true do
        let $R' = I$
        while true do
            let $A = \text{toCNF}( R'(W_0) \land T(W_0, W_1) )$
            let $B = \text{toCNF}( \bigwedge_{1 \leq i \leq k} T(W_i, W_{i+1}) \land \bigvee_{1 \leq i \leq k} \neg p(W_i) )$
            if SAT( $A \land B$ )
                if $R' \equiv I$ return false
                else break
            else ( let $P$ be the resolution refutation of $A \land B$ )
                let $C(W_1)$ be an interpolant derived from $P$
                if $C(W_0/W_1)$ implies $R'(W_0)$ return true
            let $R' = R' \lor C(W_0/W_1)$
        od
        let $k = k + 1$
    od
end

Figure 4.1. A complete procedure for checking safety properties by computing Craig interpolants.

$p$. In practice, our image over-approximation will often become adequate for relative small values of $k$.

4.2 An Algorithm for Interpolation Based Model Checking

We will now give a complete procedure for model checking of safety properties based on interpolation. The procedure will take as input the initial constraint $I$, the transition constraint $T$, the property to be checked $p$. It will return true if and only if $\neg p$ is unreachable from $I$ under $T$. In other words, if and only if there exists no path from an initial state to a state not satisfying $p$. Otherwise, the procedure will return false.

The procedure relies on two external functions. The function $\text{toCNF}$ translates arbitrary propositional formulas into conjunctive normal form (CNF) and the function $\text{SAT}$ performs satisfiability testing of CNF formulas and, in the unsatisfiable case, produces a resolution refutation $P = (V, E)$.

The procedure is shown in Figure 4.1. As can be seen from the figure, it consists of an outer and an inner loop. The outer loop just increases $k$ until we find a counterexample or our $k$-adequate image over-approximation becomes adequate. The more interesting inner loop performs the computations. The procedure runs
CHAPTER 4. INTERPOLATION BASED MODEL CHECKING

as follows. First we look for all counterexamples of length 0. If there are no such counterexamples we set $k$, the current search depth, to be 1 and enter the loop. We let $\mathcal{I}$ be the initial over-approximation $\mathcal{R}'$ of the set of reachable states. Then we partition the system. We let $A$ be the set of states reachable in exactly one transition from $\mathcal{R}'$, and we let $B$ be the remaining $k - 1$ transitions to states where $\neg p$ holds.

We then apply the SAT-solver to the clause form of $A \land B$. If the formula turns out to be satisfiable and $\mathcal{R}' \equiv \mathcal{I}$ then we have indeed found a true counterexample from an initial state to a state not satisfying the property $p$. However, if $\mathcal{R}' \neq \mathcal{I}$ then it might be the case that we have over-approximated too much, because we never know explicitly when our over-approximated image becomes adequate, and that we have found a false counterexample. If this is the case we simply abort the loop and increase $k$.

On the other hand, if we have that $A \land B$ is unsatisfiable, and the SAT-solver provides us with a resolution proof $P$ of $\neg (A \land B)$, then we can derive an interpolant $C$ from $P$, by the method described in Section 2.2. Because the interpolant $C$ is over $W_1$, and our over-approximated set of reachable states $\mathcal{R}'$ is over $W_0$ we need to substitute the $W_1$-variables in $C$ with their corresponding $W_0$-variables. This is expressed in the pseudo-code in Figure 4.1 as $C(W_0/W_1)$. Then, if $C(W_0)$ implies $\mathcal{R}'(W_0)$ (which can be determined by the SAT-solver by proving that $\neg (C \rightarrow \mathcal{R}')$ is unsatisfiable), it means that all states in the image over-approximation $C$ are also included in $\mathcal{R}'$, hence a fixpoint has been reached. And since $\mathcal{R}'$ is an over-approximation of the reachable states, we can conclude that $\neg p$ indeed is unreachable, because we have refuted a formula $(A \land B)$ that tries to reach $\neg p$ from $\mathcal{R}'$ in $1 \ldots k$ transitions. (And since we have reached a fixed point, $\mathcal{R}'$ is an invariant of $\mathcal{T}$).

Otherwise, if the fixpoint has yet to be reached, we update $\mathcal{R}'$ by adding $C$ to it, and continue.

Although our reasoning above and in Section 4.1 basically guarantees the soundness of our algorithm, we will now give a more formal proof of it, according to [15].

**Theorem 4.2.1** For any initial constraint $\mathcal{I}$, total transition constraint $\mathcal{T}$ and property $p$, the procedure INTERPOL( $M = (\mathcal{I}, \mathcal{T})$, $p$ ) returns true iff $\neg p$ is unreachable from $\mathcal{I}$ under $\mathcal{T}$.

**Proof** Suppose first that the procedure returns false. Then either $\mathcal{I} \land \neg p$ or $\mathcal{I} \land (\bigwedge_{0 \leq i < k} \mathcal{T}_i) \land (\bigvee_{1 \leq i \leq k} \neg p_i)$ is satisfiable. Thus $p$ is $\mathcal{T}$-reachable from $\mathcal{I}$ in from 0 to $k$ steps.

On the other hand, if the procedure returns true, then we know that $C(W_1) \rightarrow \mathcal{R}'(W_1)$. Additionally, since $\mathcal{R}'(W_0) \land \mathcal{T}(W_0,W_1) \rightarrow C(W_1)$ we know, by transitivity of implication, that $\mathcal{R}'(W_0) \land \mathcal{T}(W_0,W_1) \rightarrow \mathcal{R}'(W_1)$. In other words, advancing one transition from our set of reachable states does not yield any new states. Hence, we have reached a fixed point, and since $\mathcal{I} \rightarrow \mathcal{R}'$ (because $\mathcal{R}' = \mathcal{I} \lor C_1 \lor C_2 \ldots$), we know that all states that are reachable from $\mathcal{I}$ are included in $\mathcal{R}'$. Finally, we have
4.3 IMPLEMENTATION

that $\mathcal{R}' \land \neg p$ is unsatisfiable, because $\mathcal{I} \land \neg p$ is unsatisfiable initially, and at each iteration $C \land \neg p$ is unsatisfiable. Hence $p$ is $\mathcal{T}$-unreachable from $\mathcal{I}$. □

It is also easy to see that the procedure eventually will terminate. Let’s start our argumentation for this by looking at the inner loop of the procedure. For a given $k$ and for each iteration of the loop, $\mathcal{R}'$ will grow. Because we have a finite state space, eventually a fixed point will be reached. However, if we find a false counterexample due to the over-approximation before this happens, we will abort and increase $k$. But when $k$ has grown beyond the diameter of the state space and $A \land B$ is still unsatisfiable the first iteration, we know that $\mathcal{I}$ cannot reach $\neg p$ in any number of steps. Also, the interpolant $C$ cannot reach $\neg p$ in any number of steps, because it is inconsistent with $B$. Thus, $\mathcal{R}'$ cannot reach $\neg p$ in any number of steps, so by induction it follows that $\mathcal{R}'$ will never be able to reach $\neg p$. Hence $\mathcal{R}'$ will continue to grow until it reaches a fixed point, at which time the procedure terminates.

4.3 Implementation

The interpolation based algorithm has been implemented as part of a stream based proof engine for sequential logic. The stream based environment differs somewhat from the state-transition systems that in this thesis have been defined by two first order formulas, one being an initial constraint and the other a transition constraint. In a stream based environment, these formulas do not exist explicitly. Instead, the state-transition system is defined by streams, which should be viewed as time dependent Boolean variables. An input stream, for example, is in every time step a free Boolean variable, while a combinational stream is defined in terms of two other streams and a Boolean operator. Latch streams can be constructed by letting them be defined by the values of streams not only in this time step but also in the previous time step. As usual, the latch streams are given a special initial value for the first time step.

To illustrate how we can model systems by the use of streams, let’s return to the sequential logic circuit depicted in Figure 3.2. We defined this circuit by the initial constraint $v_0 \land \neg v_2$ and the transition constraint $(v'_0 \leftrightarrow v_0 \land v_1) \land (v'_2 \leftrightarrow \neg v_1 \lor v_2)$. In a stream based environment, we would instead define this system by the following equations:

\[
\begin{align*}
I(v_0) & = \top \\
X(v_0) & = v_0 \land v_1 \\
I(v_2) & = \bot \\
X(v_2) & = \neg v_1 \lor v_2
\end{align*}
\]

Here, the $I$-operator assigns an initial definition to a latch stream, and the $X$-operator defines the next stream of a latch, i.e. $X(s)[t] = s[t+1]$, for a given instance in time $t$.  

23
Clearly, it is easy to create the initial and transition constraints from the stream definitions: To create $I$, we simply conjunct the initial definitions of the latch streams and to create $T$ we conjunct the next definitions of the latches. With that in mind, we will describe the implementation of the interpolation based algorithm with a stream based notation.

In a stream based environment, the task of checking that a safety property is always true in a given system, becomes checking if a stream $s$, which we will call a proof obligation, is true in every time step.

Without going into details, the implementation works roughly as follows. We first construct a function $toCNF$ that takes a stream $s$ and returns the clause form of the definition of $s$. The function is recursive and the base case is when $s$ is an input or latch stream, in which case it just returns without doing anything. On the other hand, if $s$ is a combinational stream, defined in terms of two other streams $t$ and $u$ and one of the Boolean operators $\rightarrow$ or $\leftrightarrow$, we will convert the definition in the following way.

- If $s = t \leftrightarrow u$, we create the four 3-clauses: $s \lor t \lor u$, $s \lor \neg t \lor \neg u$, $\neg s \lor t \lor \neg u$ and $\neg s \lor \neg t \lor u$. Note that the conjunction of these clauses will be satisfiable if and only if the original formula is satisfiable.

- If $s = t \rightarrow u$ the conversion process is done in two parts. First, the formula is converted into the disjunction $s = \neg t \lor u$. Then, the disjunction is, if possible, collapsed with other disjunctions created in the conversion process. For example, if we have the disjunctions $a = b \lor c$ and $c = d \lor e$ and $c$ does not exist elsewhere, we can collapse the disjunctions into $a = b \lor d \lor e$. This reduces the number of variables in the CNF instance and has improved the performance of the interpolation based algorithm considerably on certain problems.

- In the last step, all disjunctions are converted into a set of clauses in the following way. If $s = t_1 \lor t_2 \lor \ldots \lor t_n$ we create $n$ clauses of the form $s \lor \neg t_i$ for $1 \leq i \leq n$ and one clause $\neg s \lor t_1 \lor t_2 \lor \ldots \lor t_n$. The binary clauses can be viewed as implications of the form $t_i \rightarrow s$, i.e. if any $t_i$ is true then $s$ is true. Also, the clause $\neg s \lor t_1 \lor t_2 \lor \ldots \lor t_n$ is equivalent to $\neg t_1 \land \neg t_2 \land \ldots \land \neg t_n \rightarrow \neg s$; in other words, if all the $t_i$:s are false then $s$ is also false.

For the remaining Boolean operators such as $\land$ and $\lor$ we can use simple rewriting rules (e.g. $a \land b \equiv \neg (a \rightarrow \neg b)$ and $a \lor b \equiv \neg a \rightarrow b$) to get the definitions on the format that $toCNF$ requires.

Having defined the conversion function, we are now ready to describe the implementation of the algorithm. First, from the set of latches $L = \{l_1, \ldots, l_n\}$ and their initial definitions $Q = \{q_1, \ldots, q_n\}$, we create a stream $R = \bigwedge_{i=1}^{n} l_i \leftrightarrow q_i$, that represent the initial constraint. Then, assuming our proof obligation is a stream named $s$, and $s_t$ is the stream that represent $s$ at timestep $t$, i.e. $s_t = X^t(s)$ \footnote{We define $X^t(s)$ recursively by $X^0(s) = X(s)$, $X^{t+1}(s) = X(X^t(s))$}, we
4.4. OPTIMISATIONS

partition the system in two sets of clauses $A$ and $B$ such that

$$A = \text{toCNF}(R) \land \text{toCNF}(s_1), \quad \text{and}$$

$$B = \text{toCNF}(s_2) \land \ldots \land \text{toCNF}(s_k) \land (\neg s_1 \lor \ldots \lor \neg s_k)$$

Note that, when the definition a stream $x$ has been converted to clauses by $\text{toCNF}$ it will be marked “visited”, so that no redundant clauses are introduced.

It is not obvious to see that $A$ represent $\mathcal{I}$ and one instance of $\mathcal{T}$, and $B$ the remaining instances of $\mathcal{T}$ together with the negation of the property to be proved. However if we examine the formulas more closely, we have that (1) the stream $R$ defines the state of the system initially, because it includes all the initial definitions of the latches, and (2) the definition graph of each $s_i$ – as traversed by $\text{toCNF}$ – can be viewed as a tree (it is really a DAG), with latches and inputs as leaves. The transition relation will exist implicitly in this definition graph, because most likely a stream $s$ at some timestep $i$ will depend on the values of streams of timestep $i-1$, if it is not solely depending on inputs.

Having partitioned the system, we can now invoke the SAT-solver on the input $A \land B$. If we find a satisfying assignment, it is a counterexample. Otherwise we derive the interpolant $C$ from the resolution refutation of $A \land B$. Then we check if $C$ implies $R$. If it does, we have reached a fixed point and we terminate. If not, we let $R = R \lor C$ and iterate.

4.4 Optimisations

Three optimisations of the basic interpolation based algorithm have been implemented and evaluated. We will describe them now in order of importance.

1. As described in this chapter, in each iteration of the algorithm, we compute the forward image of $\mathcal{R}'$, which is the over-approximation of the entire set of reachable states. However, this is not necessary, and only leads to doing the same work over and over again. Instead, as in traditional symbolic reachability analysis (c.f. Section 3.3), it is sufficient to compute the forward image in each iteration of only the “new” states, i.e. the previous forward image. For our purposes, this means that at each iteration (except the first) we compute the over-approximated forward image of the interpolant $C$, and not $\mathcal{R}'$. This optimisation leads to significant speed-ups – in some cases of several orders of magnitude.

2. Consider the $B$ part of the system, which can be written as

$$B^k_j = \bigwedge_{i=1}^{k-1} \mathcal{T}(W_i, W_{i+1}) \land \bigvee_{i=j}^k \neg p(W_i).$$

Thus far, we have only considered $B^k_j$, i.e. we check for all counterexamples of length $1\ldots k$. However, we could choose a higher value for $j$, and
CHAPTER 4. INTERPOLATION BASED MODEL CHECKING

the algorithm would still be sound, because we are still over-approximating. Choosing a higher value for \( j \) means we need to refute the final condition at fewer depths, which is an easier problem to solve. The drawback is that we lose the completeness of the algorithm, because now there is no guarantee that our over-approximate image operator will become adequate (even if \( k \) is larger than the diameter of the system, the property could still be violated at depths \( \leq j \)). However, in practice, almost all systems are solvable for \( j = k \), and only in a few cases has divergence been observed. In all these cases setting \( j \) to a lower value has been enough to solve the problem. Choosing \( j = k \) means we only have to refute the final condition (\( \neg p \)) at a single step, so the decrease in run time observed has been substantial.

3. For a final optimisation, we note that \( B \) does not change from one iteration to the next, except when \( k \) is increased. Clearly, it is unnecessary to rebuild this formula at every iteration, although the time that this takes is negligible compared to the time that is spent by the SAT-solver. However, of more importance, it is theoretically possible to make the SAT-solver learn from mistakes (i.e. conflicting decisions) at previous iterations. More precisely, we keep the so called conflict clauses inferred from \( B \) that the SAT-solver derived in the previous iteration in proving that \( A \land B \) is unsatisfiable. These clauses are all implied by \( B \) so they do not introduce any inconsistencies, but instead they strengthen the system, and forces the SAT-solver not to make the same conflicting decisions again (hence they prune the search space). The efficiencies gained from this optimisation will be investigated further in the next section.

4.5 Evaluation

The standard interpolation based model checker has been compared to a state-of-the-art induction-based model checker. In the following, when we refer to the standard interpolation based algorithm, we mean the algorithm described in Figure 4.1 enhanced with all three optimisations from Section 4.4. Five suites of different benchmarks have been used for the comparison. The benchmarks come from real industrial designs and are provided by Prover Technology’s customers. We will call the benchmark suites A-E and characterise them as follows:

A. Contains requirements for railway interlocking and supervision systems.

B. Various examples coming from embedded software design tools. Some examples are from hardware designs.

C. Contains requirements for hardware circuits.

D. Same as suite C, but with larger formulas (in terms of the number of variables).

E. Contains mainly equivalence checking formulas for hardware circuits.

26
4.5. EVALUATION

It should be noted that the examples of benchmark suite A and most of the examples of suite B are known to be difficult for the induction based method. Choosing such a collection of benchmarks is motivated by the fact that there is no real use in having a second method that performs well on problems that already can be handled by existing methods. All benchmarks are valid properties, with two exceptions. These two have counterexamples at depth 3 and at depth 32, respectively. The reason for this is that using the interpolation based algorithm for finding counterexamples is pointless, since the bounded model checking algorithm is always faster at this task.

All tests were made on an Athlon T-bird 1.4 GHz workstation with 2 GB main memory, running Red Hat Linux version 7.2. A time out value of 10000 seconds were used for each property and the results are summarized in Table 4.1. The second column of Table 4.1 contains the number of properties to be proved for each benchmark suite. The column Resolved shows how many properties that could be solved by the given method, and Wins how many properties that were resolved faster by the given method.

<table>
<thead>
<tr>
<th>Suite</th>
<th>Properties</th>
<th>Interpolation</th>
<th>Induction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Resolved</td>
<td>Wins</td>
</tr>
<tr>
<td>A</td>
<td>80</td>
<td>79</td>
<td>76</td>
</tr>
<tr>
<td>B</td>
<td>39</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.1. A comparison of interpolation- vs induction-based model checking.

As can be seen in Table 4.1, the interpolation based method clearly outperforms the induction based method, so further comparisons between these two have not been made. On the other hand, comparing different versions of the interpolation based method might be of greater interest. In particular, we will compare the standard version of the algorithm with two variants, one where conflict clauses are not reused (i.e. the third optimisation of Section 4.4 is not used), and one where saturation is used to preprocess the system. Saturation is the propagation procedure of Stålmarck’s method [22] and will not be explained in detail here. However, the result of saturating the system is a set of equivalences between the streams (latches and combinational streams) of the system. These equivalences can be added to our SAT-instance \((A \land B)\), and will thus strengthen the system.

\(^{2}\)A value of \(j = k\) (c.f. Section 4.4) for the interpolation based method was used for all properties. However, one benchmark from suite E diverged for this value but could be solved in 0.4 seconds with \(j = k - 2\). This property is not included in the Resolved column. Additionally, all properties that the induction-based method could solve, was also solved by the interpolation based method, with one exception. The average depth at which the interpolation based algorithm proved the valid properties, was 4. The maximum depth was 19 and the minimum 1.
CHAPTER 4. INTERPOLATION BASED MODEL CHECKING

First, in Figure 4.2, we show the results of comparing the standard version of the interpolation method with the one not reusing conflict clauses. The figure shows the run time for each property of the standard version on the x-axis and the run time of the “unoptimised” version on the y-axis. Thus, a point above the diagonal indicates a win for the version reusing conflict clauses. Again, a time limit of 10000 seconds was used for each property, so a run time of 10000 seconds indicates that the property could not be solved within the time limit. In some cases the algorithm ran out of memory before the time limit was reached, and these properties will also be plotted at a value of 10000 seconds. The set of benchmarks used was the same as previously, but in some cases many similar properties have been combined in one point in the graph, that indicates the total time for solving them all. (Otherwise there would be too many points in the graph).

![Graph showing comparison of interpolation methods](image)

**Figure 4.2.** A comparison of two different versions of the interpolation method.

We count 24 wins for the optimised version and 13 wins for the unoptimised version, with 9 properties solved by neither version. The total time for all properties solved by both versions was 12300 seconds for the optimised version and 18900 seconds for the unoptimised. As expected, it seems like the optimised version has a slight advantage. However, the result of one property is particularly hard to explain; the unoptimised version solves this property in just over 300 seconds, while the optimised version fails to solve it within the given time limit (it actually runs out of memory after around

\[ A \text{ value of } j = k \ (\text{c.f. Section 4.4}) \text{ were used for these runs.} \]
3000 seconds). To explain why such a result is even possible, we must consider how the SAT-solver works. The search path (leading to a satisfying assignment or proof of unsatisfiability) explored by a DPLL\(^4\)-based SAT-solver [9, 8] (which is the kind of solver we use) is highly arbitrary. Running the solver twice on the same formula but with a different variable ordering can typically result in widely different run times. Also, the resolution proof generated in the unsatisfiable case can vary greatly in size and structure, depending on exactly what clauses are used in the proof, and in which order the resolution steps were made. Thus, since different resolution proofs will generate different interpolants, the run times of the interpolation based algorithm is to some extent randomised. For example, if we are lucky and the DPLL-generated resolution proofs give us relatively strong interpolants, the algorithm might converge to a result more rapidly. To confirm that this is most probably the reason why the optimised version failed on this particular benchmark, we ran the algorithms again, giving a directive to the SAT-solver to use explicit randomisation in the variable ordering. In doing that, we were not able to make the optimised version prove the property, but we did manage quite easily to get the unoptimised version to fail in the same manner as the optimised failed – by running out of memory after 3000 seconds. Thus it seems we were very lucky that we managed to prove the property at all with the unoptimised version.

Second, the results of the comparison of the standard version of the interpolation based method to a version that uses saturation to preprocess the system, are shown in Figure 4.3. The same benchmarks and time out value were used as before\(^5\).

We count 19 wins for the standard version and 20 wins for the version with preprocessing. However, the total time on the set of benchmarks that both methods solved (excluding a benchmark that took just over 9700 seconds for the preprocessing version) was 11700 seconds for the standard version and just 4100 seconds for the version with saturation as preprocessing. In many cases when the preprocessing version wins, it wins by one or even two orders of magnitude. Many of the wins for the standard version could probably be explained by the overhead that the saturation phase took. More precisely, if the saturation phase does not derive any facts that are useful for the interpolation based model checking procedure, then the time overhead for saturating the system will not be compensated by faster model checking. Also, this time in three cases, the “more optimised” version – i.e. the one that uses saturation – shows extremely poor performance compared to the standard version. Since it is extremely hard to analyse the runs of the algorithms and the complex systems that we use them for, we will dismiss these results in the same manner as before, i.e. by blaming on the random nature of the SAT-solving.

Returning to the positive results, we see that using saturation before invoking the model checking procedure in many cases reduces the run time significantly. In particular, many of the formulas in benchmark suite A are solved much faster using saturation. It could be that these formulas are large in the number of variables, but

---

\(^4\)DPLL stands for Davis, Putnam, Logemann, and Loveland.

\(^5\)A value of \(j = k\) (c.f. Section 4.4) were used for these runs.
Figure 4.3. A comparison of the standard interpolation method to a version that uses saturation as preprocessing.

still easy in some sense (for example, they might be easy to simplify). Such formulas are precisely the kind of formulas that saturation handles very well, which might explain the speed-up observed.
Chapter 5

Conclusions and Future Work

We have described an interpolation and SAT-based procedure for verification of safety properties of sequential systems. The key observation is that resolution refutations of unsatisfiable bounded model checking instances can be exploited to allow unbounded model checking. Intuitively, the interpolants that are extracted from these refutations capture the facts that the SAT-solver considered relevant in proving that a bad state is unreachable within a bounded number of steps. Since modern SAT-solvers, e.g. Chaff [17], are highly optimised and efficient at narrowing an unsatisfiable formula down to relevant facts, this method proves to be extremely fast and robust. The implementation of the interpolation based procedure, that were made as a part of this thesis, confirms this. In the benchmark studies that are accounted for in Section 4.5, it is shown that the interpolation based model checker clearly outperforms a state-of-the-art induction based model checker, on benchmarks that were previously known to be hard for the induction based checker. The benchmark studies also showed that reusing conflict clauses generated from previous iterations proved to be a fairly good optimisation, although the efficiencies gained were not as great as one might have hoped. On a brighter note, using saturation to strengthen the system before invoking the model checking procedure proved to be very efficient in many cases. In some cases the run times were reduced by almost two orders of magnitude. This can be explained by the fact that saturation strengthens the system by putting additional constraints on it, something that will often lead to faster convergence. To our knowledge, this is the first time the interpolation based algorithm has been combined with saturation as preprocessing.

For future work, there are a couple of suggestions that can be made. First, let’s recall the second optimisation of Section 4.4. This optimisation allows us to choose the depth at which to start refuting the final condition, by parameterizing the procedure by a parameter called $j$. For example, setting $j = k$ means we will only refute the final condition at the final step. However, by choosing a value of $j$ other than 1, our model checking procedure loses the guarantee of termination. In a few cases divergence has been observed and this has been corrected by choosing a lower value for $j$. As for future work, it might be interesting to analyse the systems
for which divergence is observed, in order to try and characterise them in some way. For example, it has been noted that in some cases when properties are latched by a sequence of $n$ latches, choosing a value of $j > k - n + 1$ results in divergence of the interpolation based algorithm when trying to prove the properties. The ultimate goal of the analysis would be a fully automatic method for choosing the correct value of $j$ such that the procedure diverges for $j + 1$.

Second, as has been mentioned previously (c.f. Section 4.5), the run times of the interpolation based algorithm depend highly on the kind of resolution proofs that the SAT-solver provides. For one and the same unsatisfiable formula, there exists potentially an enormous amount of resolution proofs, of which only one is provided by the SAT-solver. What we would like is for the SAT-solver to give us the resolution refutation that generates the “best” interpolant. If “best” means the interpolants that give us the fastest run times, it would probably mean the strongest interpolant, because a stronger interpolant is more likely to be an adequate over-approximation of the forward image. Hence a run with stronger interpolants will converge more rapidly. However, forcing the SAT-solver to return resolution refutations that generate stronger interpolants is of course not trivial. Fortunately, there exists some literature that deals with this problem. In [12] an approach that rewrites the resolution proof obtained, rather than changing the SAT-solver, is used. The approach uses simple rewriting rules that pulls resolutions that generate “or” gates in the interpolant circuit towards the hypothesis of the proof, while resolutions that generate “and” gates are pulled towards the conclusion of the proof. This would lead to “or” gates at the inputs to the interpolant circuit, and “and” gates at the output, giving a stronger interpolant. In the best case scenario this interpolant would be in conjunctive normal form. The work that needs to be done is thus to implement and evaluate this rewriting procedure.

Third, it would be interesting to know if, and how, interpolants can be derived from proofs of unsatisfiability other than resolution refutations. For example, is it possible to create an interpolant from a proof by saturation? Also, the interpolation based algorithm needs to be extended to handle some basic arithmetic, in order to be really useful.

Finally, further evaluations of the interpolation based algorithm combined with saturation as preprocessing should be carried out, as the results so far are promising.
Bibliography


