Lauren Pick

A Model Checker Using IC3

Computer Science Tripos – Part II

Homerton College

April 15, 2016

Proforma

Name:	**Lauren Pick**
College:	**Homerton College**
Project Title:	**A Model Checker Using IC3**
Examination:	**Computer Science Tripos – Part II, 2016**
Word Count:	
Project Originator:	Lauren Pick
Supervisors:	Dr Dominic Mulligan, Dr Ali Sezgin
Supporting Supervisor:	Prof Alan Mycroft

Original Aims of the Project

Describe the original aims of the project, i.e. summarize information from the "Substance and Structure" and "Success Criteria" sections of the project proposal.

Work Completed

Describe the work completed as part of project, including extensions.

Special Difficulties

None.

Declaration

I, Lauren Pick of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

Chapter 1

Introduction

This project focuses on implementing the IC3 algorithm, a SAT-based model checker. To provide context for the project, I provide a brief introduction to formal verification and model checking, followed by a discussion of symbolic model checking and SAT-based model checking. I then highlight some of the important features of the IC3 algorithm and its more recent uses.

Formal verification is the use of mathematics and logic to prove properties of hardware and software systems. Formal verification techniques are employed in several domains where the correctness of a system is crucial, such as in hardware design and aviation. Guaranteeing that there are no errors in hardware designs before manufacturing begins can help avoid the high costs associated with needing to remanufacture the hardware if the design needs to be corrected. Additionally, the ability to prove that certain kinds of errors do not occur are important for ensuring safety in safety-critical systems such as those in airplanes and medical devices.

One formal verification technique is model checking. Given a model of the system and a specified property, a model checker will check whether all reachable states in the system satisfy this property. Unlike formal verification techniques that employ Hoare Logic and theorem provers that require user guidance, model checking is a fully automated verification technique.

In addition to being automated, because model checkers prove properties by considering reachable states, when a model checker discovers that a property does not hold, the model checker has necessarily discovered a reachable state that violates the property. A counterexample trace can, in such cases, be provided, giving greater insight into why the property does not hold for that model.

1.1 Symbolic Model Checking

All model-checking approaches suffer from limitations on the size of the systems they can model check in practice as a result of the state explosion problem: the number of states in a system can be (and often is) exponential in the number of state variables [7].

The initial approach to the model checking problem involved explicitly considering each reachable state in the model. Symbolic model checking arose as a method of miti-

gating the effects of the state explosion problem to some extent. By representing states and the transition relation between them as boolean expression, symbolic model checking allows sets of states to be represented efficiently as boolean expressions involving state variables as opposed to an explicit list of each individual state in the set [11].

Symbolic model checking was originally invented for use with ordered binary decision diagrams (BDDs), data structures that provide an efficient representation of propositional formulas. A unique BDD represents each logical formula given a particular variable ordering, and an implementation that only stores each BDD once and uses pointers appropriately can result in less space being used. The efficiency of BDDs in storing propositional formulas allowed for the model checking of systems with larger numbers of states than could be handled by explicit-state model checking [11].

The efficiency of BDD representations relies on choosing an appropriate ordering, which can be computationally expensive, and in some cases, there is no such ordering that results in a space-efficient BDD [1].

An alternative to BDD-based symbolic model-checking techniques are SAT-based techniques, which do not use the canonical representations of propositional formulas as BDDs and make use of procedures for solving the boolean satisfiability problem. Such techniques include bounded model checking (BMC), which proves properties by finding counterexamples of certain lengths [1]; k-induction, which proves properties inductively [12]; and the IC3 algorithm that is the focus of this project.

Many modern SAT-based model checkers are based on BMC, which begins at the initial state of the transition system and searches for paths of length k from the initial state that violate the negated property. If no path of length k is found, BMC increments k and searches again. BMC is effective at finding counterexamples, but for some large systems, BMC is incomplete unless it is allowed to reach the point at which k is the maximal path length.

The k-induction algorithm is similar to BMC in its unrolling of the transition relation to consider paths of length k, but it also incorporates induction. At each depth k, the algorithm asserts that the desired property holds at each state before the final state.

1.2 The IC3 Algorithm

The IC3 algorithm (also called PDR [9]) is a SAT-based model-checking algorithm for proving the safety properties (i.e. properties that must hold in all reachable states) of hardware. The first implementation of the algorithm `ic3` placed third in the 2010 Hardware Model Checking Competition (HWMCC'10), a competitive event that receives model checker and benchmark submissions from industry and academia. Its performance at HWMCC'10 generated interest in the algorithm, and since then, several variants of the algorithm have been developed.

As in k-induction, properties are proved through induction: the algorithm considers reachable sets of states k steps away from the initial state until reaching a fixed point. As with later variants of k-induction, the IC3 algorithm also discovers new invariants if the initial assumptions are not strong enough to prove the desired property, but unlike

k-induction, the safety property guides the discovery of the invariants. As a result, the discovered invariants are more relevant for proving the safety property [5].

Furthermore, IC3 does not unroll the transition relation as k-induction or other BMC-based methods do, but instead considers at most one step of the transition relation at a time, leading to smaller, simpler SAT queries. As a result, IC3 requires less memory than BMC-based methods in practice [5].

1.3 Further Work

While IC3 algorithm is for model checking safety properties of hardware, there are applications of the algorithm in model checking LTL and CTL properties and model checking software [5].

The FAIR model-checking algorithm, which model checks ω-regular properties, makes use of a safety model checker such as IC3, and IICTL, which model-checks CTL formulas [10], makes use of both a safety model checker such as IC3 as well as a fair cycle finder such as FAIR.

Other work generalizes IC3 to use an underlying SMT solver rather than a SAT solver, and this generalization has been used to check control-flow graphs of programs [6]. More recently, Johannes Birgmeier, Aaron Bradley, and Georg Weissenbacher have introduced CTIGAR, a method of abstraction-refinement based on IC3's counterexamples to induction rather than the counterexamples in CEGAR, which experiments suggest is competitive with CEGAR-based techniques for software verification [3].

1.4 Project Aims

This project's main aim is to implement a basic form of the IC3 algorithm in Haskell that should be able to correctly check several small examples. Additionally, the project is meant to give me an opportunity to learn and use Haskell, to understand formal methods and especially model checking more deeply, and to put into practice software engineering techniques.

I have successfully implemented not only a basic form of the IC3 algorithm, which can correctly check several examples from the hardware model checking competition in addition to the small hardware models I initially set out to check, but also several variants of the algorithm.

Chapter 2

Preparation

The preparation for the project involved learning Haskell, distinguishing the main components of the project, choosing and learning how to use the necessary tools for implementing the components of the project, and gaining the necessary knowledge about the IC3 algorithm.

2.1 Haskell

The majority of the code for the project is implemented in the functional programming language Haskell. I explain many of the features of the language by comparison with Standard ML.

Types Similarly to Standard ML, Haskell compilers perform type inference, but type signatures can be (and, in some cases where type inference cannot resolve ambiguities, must be) provided. For example, the type signature for the `prove` function in the model checker, which is a curried function that takes a `Model` and a `Lit` and returns a `Bool`, implementation is given in figure 2.1.

```
prove :: Model -> Lit -> Bool
```

Figure 2.1: The type signature for the `prove` function.

Haskell type constructors work similarly to Standard ML type constructors.

Lists The model checker implemented as part of this project makes extensive use of lists. List types in Haskell are denoted using square brackets, i.e. a list of `Lit`s has type `[Lit]`. Lists can be appended using the `++` infix operator, and elements can be prepended to lists using the infix operator `:`. For example, the Haskell equivalent to the Standard ML expression `1::[2,3,4] @ [5,6]` is `1:[2,3,4] ++ [5,6]`.

Modules A Haskell program consists of modules, which are used to organize code. Modules (or just selected functions from modules) can be imported into other modules, which is how library functions can be used.

2.2 Requirements Analysis

The model checker requires a way of taking input models and also requires a way to solve SAT queries. I chose the AIGER format for representing the hardware models and the MiniSat SAT solver for answering SAT queries, resulting in a need for an AIGER parser and a Haskell interface to MiniSat. The choice of the AIGER format allows the model checker to be run on examples from the Hardware Model Checking Competition, since this is the format used to specify examples in the competitions. I chose the MiniSat SAT solver because it is the SAT solver used by Aaron Bradley's reference implementation of IC3 that this implementation is to be compared against.

Given that the model checking algorithm deals with transition systems (discussed later), the implementation also requires a representation of transition systems, which should correspond to the input hardware model. A further requirement is the implementation of the IC3 algorithm itself.

The main required components are thus the AIGER parser, MiniSat interface, transition system representation, and IC3 algorithm implementation.

2.3 Tools Used

I used a variety of tools to employ software engineering practices, such as version control and testing, and to otherwise ease the development of the project's code.

Git The Git version control system was used for managing the project's code. The previous versions maintained by the version control system proved useful in the development of the code, and branching and merging capabilities were useful for working on and organizing different variations of the model checker. Git submodules, which allow the inclusion of other Git projects within another project, were used to include MiniSat within the project, enabling easier acquisition of project dependencies (i.e., MiniSat can be obtained by running `git submodule init` after running `git clone` to clone the repository).

GitHub, a widely-used hosting service for Git repositories, was used to keep backups of the code.

Haddock The Haddock documentation tool for Haskell was used to generate documentation for the code. Haddock automatically generates documentation in several formats (e.g. HTML) from annotated Haskell code. It is commonly used to document Haskell code, being used for most packages available on Hackage.

HUnit HUnit is a framework for writing unit tests in Haskell based on the JUnit framework for unit testing in Java. HUnit allows tests to be specified by using functions that return the `Assertion` type to write `TestCase`s. For example, the `assertBool :: String -> Bool -> Assertion` function takes a `String` that gives an error message and a `Bool` value, and raises an exception (with the error message) if the

`Bool` is not `True`, so the following expression would give a `TestCase` that tests that function `isEven` returns `True` when called with parameter 12:

```
TestCase (assertBool "Error: (isEven 12) results in False" (isEven 12))
```

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

Figure 2.2: The HUnit `Test` datatype

The `Test` datatype in HUnit allows `Test`s to be grouped and built up hierarchically. Tests that have been assembled into a singly tree can then be treated as a test suite, and the whole tree of unit tests can be run.

Criterion Criterion is a library for performing benchmarking in Haskell. Criterion can output benchmarking results in any format specified in the `.tpl` template format, and by default outputs to HTML. The `.tpl` file can be configured such that Criterion will, for example, output benchmark sample results to a CSV, as the `.tpl` for benchmarking this project was configured.

Cabal Cabal is the standard package and dependency management system for Haskell, where a package may be either a library or a complete piece of software. Information about a Cabal package, such as its version and dependencies, is specified through a `.cabal` file in its root directory. The `.cabal` file may contain several different sections, such as a `library` section, describing the modules in the package that should be exposed in the library provided by the package or an `executable` section, which has fields for specifying the Haskell file containing the `Main` module and for specifying other Haskell files used by the program. The `.cabal` file for this project also uses the `Test-Suite` section to allow the HUnit test for the project to be run in a standard way (by running `cabal test` in the root directory of the package) and the `Benchmark` section to allow the benchmarking program to be run in a standard way as well (by running `cabal bench` in the root directory of the package).

Cabal also uses a Haskell file `Setup.hs` to give further information about how to build the package. For example, the `Setup.hs` file for this project compiles the C and C++ code needed for the project before Cabal attempts to build the rest of the project, so the files necessary for linking are already present.

The use of Cabal enables the project to be built easily on different platforms, since Cabal provides a standard method for building the package that works across platforms.

hsc2hs The `hsc2hs` preprocessor eases the writing of Haskell bindings to C code by enabling the programmer to write a `.hsc` file containing macros that the preprocessor can expand to, e.g., pointer offsets. The `hsc2hs` expands the macros in a `.hsc` file to produce a `.hs` file that can then be compiled with a Haskell compiler and run.

HLint The HLint tool suggests improvements for Haskell source code to improve the style of the code. The incorporation of HLint suggestions in the project resulted in simpler, more readable code.

Aiger Utilities Several tools provided in Aiger Utilities were used in the project. The AIGER parser provided was used for comparison with and as an alternative to the one developed as part of this project.

The Aiger Utilities' tools to convert between formats for specifying hardware models eased the specification of new hardware models that would be compatible with the model checker, which accepts only AIGER-formatted inputs, and that would be compatible with Aaron Bradley's reference implementation of IC3, which takes only the binary format of AIGER as input. In particular, the tools of use were the `bliftoaig` tool, which converts a circuit specified using the Berkeley Logic Interchange Format to a circuit specified using the binary AIGER format, and the `aigtoaig` tool, which converts between the ASCII and binary AIGER formats.

MiniSat MiniSat is a SAT solver implemented in C++ that solves boolean satisfiability problems posed in conjunctive normal form. Further details are given in section 2.6.

2.4 Symbolic Representation

Symbolic model checkers rely on the representation of the underlying system as a transition system, which describes the behavior of the system as one-step transitions between states. Transition systems and states are themselves defined using propositional logic formulas.

I give a brief review of concepts in logic before formally defining transition systems and explaining how propositional logic formulas represent states.

Logic Unless otherwise specified, a variable is a propositional symbol, and can be assigned to boolean values *True* or *False*. A *literal* is defined as being either a variable v or its negation $\neg v$.

The concepts of a cube and a clause are useful in describing the representation of states as logical formulas. A *cube* is defined to be a conjunction of literals and may be represented as the set of literals that occur in it. Similarly, a *clause* is a disjunction of literals that may also be represented as the set of literals that occur in it.

Given a cube c, a d is a *subcube* of c (written $d \subset c$) iff the set of literals in d are a subset of the set of literals in c. Similarly, given a clause c, a clause d is a *subclause* of c (also denoted $d \subset c$) iff the set of literals in d are a subset of the literals in c.

Through the application of deMorgan's laws, the negation of a cube is a clause and vice-versa. In particular, a cube $C = l_0 \wedge \ldots \wedge l_n$ has negation $\neg C = \neg(l_0 \wedge \ldots \wedge l_n)$, which is logically equivalent, by deMorgan's laws, to the formula $\neg l_0 \vee \ldots \vee \neg l_n$. Similarly, a clause $D = l_0 \vee \ldots \vee l_n$ has a negation that is logically equivalent to $\neg l_0 \wedge \ldots \wedge \neg l_n$. It follows that the clause obtained by negating a cube is specified by the set obtained by

negating each literal in the cube set and that the cube obtained by negating a clause is specified by the set obtained by negating each literal in the clause.

A propositional formula is in *conjunctive normal form* (CNF) iff it is a conjunction $\bigwedge_i D_i$ of disjunctions D_i of literals (i.e. clauses). A set of clauses can be interpreted as the CNF formula resulting from the conjunction of the clauses.

Transition Systems A *transition system* is a tuple (i, x, I, T) consisting of a set of input variables i, state variables x, an initial set of states represented by the logical formula $I(x)$ and a transition relation represented by the logical formula $T(i, x, x')$, where x' is the set of next-state variables.

For each state variable v, v' denotes the corresponding next-state variable. For example, a transition relation that states that all variables that are currently *True* should become *False* in the next state is as follows:

$$T(i, x, x') = \bigwedge_{v \in x} (v \Rightarrow \neg v').$$

Given transition relation (i, x, I, T), a logical formula C is, by definition, *inductive relative* to another logical formula F if both $I \Rightarrow C$ and $F \wedge C \wedge T \Rightarrow C'$ hold.

States A single state of the transition system (or a singleton set containing that state) is specified through the assignment of all variables in the transition system to boolean values, where a complete assignment is represented as a cube such that every variable appears in the formula exactly once. An incomplete assignment of variables in the transition system is a cube such that at least one variable in the transition system does not appear in the cube. Such an assignment c specifies the set of cubes $\{a \in \mathit{Full\ Assignment} | c \subset a\}$, where *FullAssignment* is the set of complete assignments to the variables in the transition system. More generally, any logical formula b involving the variables in the transition system gives the set of states $\{a \in \mathit{Full\ Assignment} | a \wedge b \text{ is satisfiable}\}$.

A state that is in the set of states represented by a logical formula B is referred to as a B state. A set of states s is said to be reachable in k steps of the transition relation iff there exist states $s_0, \ldots, s_k$ such that s_0 is an I state and $s_i \wedge T \Rightarrow s_{i+1}$ for $1 \leq i < k$.

2.5 Model Specification

I used both the AIGER format and Berkeley Logic Interchange Format (BLIF) to specify the example hardware models that were not taken from the Hardware Model Checking Competition. The hardware models that the model checker accepts as input are specified using the AIGER format; however using the AIGER format to specify larger models was cumbersome, so such models were specified using BLIF and converted to AIGER format using the Aiger Utilities' `bliftoaig` tool.

Inputs to model checker are models formatted using either the ASCII or binary versions of the AIGER format. The AIGER format provides a method of specifying hardware modeled as And-Inverter Graphs with latch elements providing single clock-tick delays: all circuits are modeled as a graph of nodes consisting only of AND gates, inverters, and

latches, where the latches behave like D flip-flops, outputting the value of the current input at the next clock tick.

The AIGER format has both an ASCII and a binary version. The ASCII format is more flexible and human readable, imposing fewer constraints on the ordering of components within the input file. For example, an AND gate with variable name 20 may be specified before an AND gate with variable name 11 in the ASCII format, but AND gates must be specified in ascending order of their variable names in the binary AIGER format. Another example is that AND gates' inputs can be specified in any order in the ASCII format, but the binary format encodes AND gates under the assumption that inputs are specified in ascending order. The the binary version's assumptions on component ordering allows the format to be more compact. The Hardware Model Checking Competition's examples use the binary format.

A new version of the AIGER format is currently under development, with examples from HWMCC'14 using the new version. The AIGER parser implemented as part of this project handles both the old and new versions of the AIGER format.

AIGER Variables A variable's name in AIGER format is a positive integer. Variables themselves are not represented directly in AIGER format; instead, nonnegative numbers are used to represent literals. I will refer to these nonnegative numbers as indices.

For any variable named x, the index for positive literal x is given by $2 \times x$, and the index for negative literals $\neg x$ is given by $2 \times x + 1$, i.e. a function to map from variable names x and a boolean value b giving the sign of the literal would be as follows:

$$index(x, b) = \begin{cases} 2x & \text{if } b \\ 2x + 1 & \text{otherwise} \end{cases}$$

The indices 0 and 1 are used to represent the constant boolean values *False* and *True*, respectively.

It is apparent that any index above 1 represents a literal and that for all such indices, all even indices represent positive literals, and all odd indices represent negative literals. The representation thus allows an implementation use the least significant bit of an index to be used to find the sign of a literal, and a single bitwise right shift to find the variable name for the literal.

Old version All AIGER files in the old version begin with a header of the form

```
V M I L O A
```

where

- `V` can take on values `aag`, specifying that the file is in the ASCII format or `aig`, specifying that the file is in the binary format.
- `M` gives the maximum index of a variable.
- `L` gives the number of latches.

- `O` gives the number of outputs.
- `A` gives the number of two-input AND gates.

The different components are specified after the header in the order that their counts are given in the header.

In the ASCII version of the format, inputs are specified by giving the index that represents the positive literal for its corresponding variable name, and outputs are specified similarly as single indices (that may represent booleans or literals of any sign).

Latches have initial value 0 (i.e., *False*) and are specified by giving the index representing the positive literals for their corresponding variable name followed by the index for their next-state value. AND gates are specified by giving the indices that represent the positive literal for their variable name and the two indices that specify their input values.

```
aag 3 0 2 1 1
2 3
4 2
6
6 2 4
```

Figure 2.3: A circuit specified in the old ASCII AIGER format

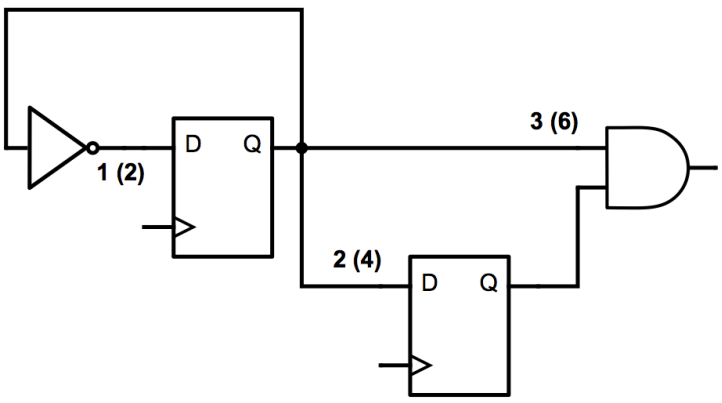

Figure 2.4: The circuit represented by figure 2.3, where the variable name for each component is directly to the left of that component, with the corresponding index in parenthesis.

For example, figure 2.3 specifies a circuit with no inputs, two latches with indices 2 and 4 and one AND gate 6 that takes the outputs of the two latches as inputs and whose output is the single output of the circuit.

Binary version In the binary version of the format, variable indices are assumed to occur in increasing order. Since each literal must be defined, this allows for the omission of the variable indices when defining inputs or latches.

Inputs are not explicitly listed: the input variables are inferred based on the value of `I`. Similarly, latches are specified by only listing their next-state literals' representations.

The binary format also assumes that AND gates occur in order of their variable indices and additionally assumes that inputs to an AND gate will have already been defined before that AND gate. These assumptions allow AND gates to be represented by two differences that tend to be small in practice:

For an AND gate specified in the old format by `lhs rhs0 rhs1`, where inputs `rhs0` and `rhs1` have been ordered such that `rhs0` $\geq$ `rhs1`, define

$$\delta_0 = \texttt{lhs} - \texttt{rhs0}$$

and

$$\delta_1 = \texttt{lhs} - \texttt{rhs1}$$

The values δ_i are then represented with the following binary encoding, giving a more compact representation for AND gates than the ASCII version of the format:

For 7-bit words $w_0, \ldots w_n$ with

$$\delta_i = w_0 + 2^7 w_1 + \ldots + 2^{7n} w_n,$$

δ_i is represented as the sequence of $n+1$ bytes $b_0, \ldots, b_n$, where

- for $0 \leq k < n$, b_k is the byte obtained by setting the most significant bit to 1 and the rest of the bits to w_k, and
- b_n is the byte obtained by setting the most significant bit to 0 and the rest of the bits to w_n

New version The new AIGER format begins with a header of the form

`V M I L O A B C J F`

where `V`, `M`, `I`, `L`, `O`, `A` are as in the old format, and

- `B` gives the number of "bad state" properties
- `C` gives the number of invariant constraints
- `J` gives the number of justice properties
- `F` gives the number of fairness constraints

The "bad state" properties allow for the specification of properties for a model checker to prove are unreachable separately from the outputs; in the old AIGER format, such properties had to be specified as outputs. The invariant constraints allow for the specification of properties that are true at all states up to and including the state where the "bad state" is found. Justice and fairness constraints are not included in the model used by the model checker and will not be explained further.

Components are specified after the header in the same order that their counts are given in the header with the exception of AND gates, which occur at the end of the file.

Latches' initial values can also be specified now with an additional `0` or `1` after the next-state literal's index. The initial value may also be given as the index of the latch itself, in which case the latch is considered to be uninitialized. If the initial value is omitted, the initial value is assumed to be 0 as in the old version of the format. The header can also be truncated after giving the number of AND-gates if the remaining counts are all zero, allowing any parser for the new AIGER format to be backwards-compatible.

The new version of the format is otherwise the same as the old format.

2.6 MiniSat

To solve a SAT query, MiniSat creates an instance of a `Solver` object, which contains a set of variables, sets of clauses, and possibly a model or a conflict vector. The set of variables in the `Solver` gives all the variables that may appear in a SAT query, the set of clauses forms the SAT query, and the model or conflict vector gives further information about the last SAT query made.

In addition to a `Var` type for representing variables, MiniSat has a `Lit` type for representing literals, and MiniSat represents sets of clauses as `vec<Lit>`s, vectors of literals. The set of clauses in a `Solver` together represent a CNF query, so that if the `Solver`'s `solve()` function is called, the resulting `bool` indicates whether the query is satisfiable or not. The `solve()` function is overloaded so that it may also take an assumption `vec<Lit>*` as an argument. The literals in the assumption vector must hold in addition to the CNF query formed by the `Solver`'s clauses: if the `Solver`'s clauses form some CNF query C and `solve(assumps)` is called, where `assumps` points to the the assumption vector containing all the literals in some set A, then the SAT query is $C \wedge \bigwedge_{l \in A} l$.

If at least one query to a `Solver` has been made, then if the last query to the `Solver` was satisfiable, the `Solver`,provides a `model` pointer to a set of satisfying variable assignments for that query, or, if the last query to the solver was not satisfiable, the `conflict` pointer to a set of literals from the assumption vector provided in the query that contributed to the query being unsatisfiable.

If there has been at least one query made of the `Solver` object, and the query was satisfiable, the `Solver`'s `model` variable points to a set of variable assignments for that SAT query. If there has been at least one query made of the `Solver` object, and the query was unsatisfiable, the `Solver`'s `conflict` variable points to a set of literals that contains the assumed literals that caused the query to be unsatisfiable.

This model checker uses instances of `SimpSolver`, a subclass of the `Solver` class that does simplification and returns full assignments.

2.7 The IC3 Algorithm

Given a hardware model (i.e. a finite-state transition system (i, x, I, T)) and a safety property P, IC3 aims either to prove inductively that P holds at all reachable states from the initial state or to find a $\neg P$ state that can be reached. The pseudocode in 2.5 gives

```
1  Function prove((i, x, I, T), P):
2      if ¬(I ⇒ P) then return False
3      F_0 := I
4      k := 0
5      while True do
6          if F_k ∧ T ⇒ P' then
7              create frame F_{k+1} initialized to ∅
8              k := k + 1
9          else
10             while ¬(F_k ∧ T ⇒ P') do
11                 cti := nextCTI(F_k ∧ T ⇒ P')
12                 if proveNegCTI((i, x, I, T), cti, k − 1) then F_k := F_k ∪ {¬cti}
13                 else return False
14         for i = 0 to k − 1 do
15             F_{i+1} := F_{i+1} ∪ {c ∈ F_i | F_i ∧ T ⇒ c'}
16             if F_i = F_{i+1} then return True
```

Figure 2.5: An overview of the IC3 algorithm. Frames are assumed to be passed by reference.

an overview of the basic IC3 algorithm, which I will refer to in my explanation of the algorithm.

The IC3 algorithm maintains a set of $k+1$ frames $F_0, \ldots, F_k$, where each frame F_i is a set of clauses whose disjunction represents an overapproximation of the set of states that reachable by the transition system in at most i steps (so, for example, F_0 is just the initial state set I, as seen in the pseudocode). The deepest frame F_k in the set of frames is called the frontier frame.

The initiation query $I \Rightarrow P$ on line 2 is used to check that the safety property holds in the initial state I. This query is run once at the start of the algorithm for the desired safety property. If it fails (i.e. if it is *False*), then the algorithm terminates, as an error state in which $\neg P$ holds is reachable in 0 steps. If the query succeeds, then the algorithm proceeds to its main loop.

The main loop of the algorithm is the while-loop beginning on line 5. The algorithm only exits the loop when it has determined whether or not the safety property holds at all reachable states in the model.

The consecution query $F_k \wedge T \Rightarrow P'$ on line 6 is is used to check whether the property P necessarily holds in the next frame. If it succeeds (i.e., if it is *True*), then IC3 creates a new frontier frame F_{k+1}. If a consecution query $F_k \wedge T \Rightarrow P'$ fails, then that means that there is an F_k state that is a predecessor of the $\neg P'$ state, i.e. there is an F_k state s and a $\neg P'$ state v with $T(i, s, v')$. The state s is called a *counterexample to induction* (CTI) state.

The algorithm aims to refine the approximation F_k of the set of states reachable in

at most k steps by showing that all states that are reachable in at most k steps are $\neg s$ states.

The call to *nextCTI* in line 11 finds the counterexample to induction state s. The call to *proveNegCTI* on line 12 attempts to prove that $\neg s$ is inductive relative to F_{k-1}, so that all F_k states are necessarily $\neg s$ states, so $\neg s$ can be added to the set of F_k clauses.

The *proveNegCTI* function works similarly to the while loop on line 10. For as long as the query $F_k \wedge \neg s \wedge T \Rightarrow s'$ is unsuccessful, the algorithm extracts a counterexample to induction cube and calls *proveNegCTI* to show that the counterexample is inductive relative to frame F_{j-1} so that the negated counterexample can be added to frame F_j. If the shallowest possible depth $j = 0$ is reached, then *proveNegCTI* fails and returns *False*.

The *proveNegCTI* pseudocode does not explicitly check for $I \Rightarrow \neg s$. An explicit check is unnecessary because if $I \Rightarrow \neg s$ does not hold, then eventually *proveNegCTI* will be called recursively with $j = 0$ and the attempt to show that $\neg s$ is relatively inductive to F_j fails.

```
1 Function proveNegCTI((i, x, I, T), s, j):
2     if j = 0 then return False
3     while ¬(F_j ∧ ¬s ∧ T ⇒ ¬s') do
4         cti := nextCTI(F_j ∧ ¬s ∧ T ⇒ ¬s')
5         if proveNegCTI((i, x, I, T), cti, j − 1)) then F_j := F_j ∪ {¬cti}
6         else return False
```

If $\neg c$ cannot be proven to hold at k steps of the transition relation from the initial state, i.e. the state s is in the actual set of states reachable in k steps from the initial state, then a $\neg P$ state is reachable in $k+1$ steps from the initial state; the safety property does not hold, and the algorithm terminates (line 13).

Because there may be several counterexamples to induction, it is necessary to perform the consecution query again (line 10). If it fails again, the process of finding the new counterexample to induction state(s) d and trying to prove that $\neg d$ holds at depth k repeats. Upon the success of the consecution query, the algorithm moves to the propagation phase.

Pushing a clause c from a frame F_i to frame F_{i+1} refers to the act of setting $F_{i+1} := F_{i+1} \cup \{c\}$. A clause c can be pushed from a frame F_i to the next frame F_{i+1} if the consecution query $F_i \wedge T \Rightarrow c'$ holds. The propagation phase of the algorithm goes through the set of frames $F_0, \ldots, F_k$, and, for every F_i with $0 \leq i < k$ (line 14), pushes all the clauses that it can from F_i to F_{i+1} (line 15).

If $F_i = F_{i+1}$ holds for any i at any point, then a fixed point has been found: frames at any greater depth than i will continue to be the same as F_i, since all the the clauses in F_i and therefore F_{i+1} can be pushed. Because F_i contains the safety property P as one of its clauses, this means that P holds in all reachable states from the initial state, and the algorithm terminates (line 16).

2.7.1 Inductive Generalization

After showing that a negated CTI state $\neg s$ is relatively inductive to a frame F_i and adding the clause $\neg s$ to frame F_{i+1}, the state s is eliminated from the approximation F_{i+1} of the set of states reachable in at most $i + 1$ steps. An improvement can be made by generalizing s to a set of several states c rather than a single state, and treating c as the CTI. If $\neg c$ is successfully proven to be relatively inductive to F_i, then adding it to frame F_{i+1} eliminates several states (i.e., all c states) at once rather than only s. Because the cube c is chosen so that $\neg c \Rightarrow \neg s$, at least one CTI state has been removed from F_{i+1}, and because c contains several states, it is possible that several CTIs may have been removed from F_{i+1} by adding the clause $\neg c$ to it. The process of finding such a cube c is referred to as *generalization*, and the best such c is the one such the $\neg c$ is the minimal inductive subclause for F_i and $\neg s$.

2.7.2 Minimal Inductive Subclauses

The *minimal inductive subclause* for a frame F_i and a clause $\neg s$ that is inductive relative to F_i (i.e. $F_0 \Rightarrow s'$ and $F_i \wedge T \wedge s \Rightarrow s'$) is a clause $\neg c$ whose literals are the smallest subset of the literals in $\neg s$ such that $\neg c$ is also inductive relative to F_i.

The minimal inductive subclause can be found by dropping each literal in $\neg s$ in turn and checking the resulting clause.

The checking phase (described by *down* in figure 3.10) performs the normal queries for determining whether the subclause is inductive relative to F_k: for a subclause $t = \neg s \ \{l\}$ found by dropping literal l from $\neg s$, it checks that $I \Rightarrow t$ and $F_i \wedge t \wedge T \Rightarrow t$ both hold.

If both formulas hold, then the literal l can be dropped from $\neg s$. If only the formula $F_i \wedge t \wedge T \Rightarrow t$ fails to hold, then it is possible that expanding the set of states in t by removing some of the literals in t would result in a clause that is inductive relative to F_i. If $I \Rightarrow t$ fails to hold, then removing any literals in t to obtain a subclause $u \subset t$ would still result in the query $I \Rightarrow u$ failing, since it is the case that $u \Longrightarrow t$.

The failure of $F_i \wedge t \wedge T \Rightarrow t$ to hold indicates that there is a predecessor to a $\neg t$ state that is a $F_i \wedge t$ state. This predecessor state p can be extracted from the SAT query for $F_i \wedge t \wedge T \Rightarrow t$ in the same way that CTIs are found. The clause t can then be expanded to the clause $t \cap \neg p$ formed by taking the common literals in t and $\neg p$. The checking

phase then repeats, checking the expanded clause $t \cap \neg p$.

```
1  Function mic(cls,i):
2      foreach literal l in cls do
3          subcls := cls \ {l}
4          if down(subcls, i) then
5              cls := subcls
6      return cls
7  Function down(cls, i):
8      if ¬(I ⇒ cls) then return False
9      if F_i ∧ cls ∧ T ⇒ cls' then return True
10     p := F_i ∧ t state such that F_i ∧ t ∧ p ⇒ ¬t'
11     cls := cls ∩ p
12     return down(cls,i)
```

Figure 2.6: The algorithm for finding the minimal inductive subclause. Clauses are assumed to be passed by reference.

An improvement to the generalization provided by finding minimal inductive subclauses in this way incorporates the use of counterexamples to generalization.

Counterexamples to Generalization

Checking if a subclause $\neg c = s \setminus \{l\}$ of a clause $\neg s$ is inductive relative to a frame F_i involves checking if $F_i \wedge T \wedge \neg c \Rightarrow \neg c'$ holds. If the implication does not hold, then $\neg c$ is not inductive relative to F_k. In the original method of generalization described above, this means that $\neg s$ cannot be generalized to $\neg c$, and generalization proceeds without dropping l.

It could be the case that the reason that the query $F_k \wedge T \wedge c \Rightarrow c'$ is unsatisfiable because F_k is too broad an approximation, similarly to why a consecution query at F_k might fail. As with consecution queries, discovering a new clause that can be added to F_k may allow the queries that check for relative induction to succeed, and the discovery of this clause can be directed by a counterexample extracted from the SAT solver after the query for $F_k \wedge T \wedge c \Rightarrow c'$.

The counterexample state in this case is called a *counterexample to generalization* (CTG), and proving the negated CTG to be true at frame F_k allows s to be generalized to c.

Chapter 3

Implementation

The implementation can be broken up into four main components: the AIGER parser, the MiniSat interface, the hardware model representation, and the model checker. I implemented several different variants of the model checker component that differ in overall structure, the finding of CTIs, the way that propagation is performed, and the way that CTIs are inductively generalized. The different variants and the differences among them are given in figure 3.1.

	Priority Queue	Smaller CTIs	Subsumed Clauses	Basic Generalization	Generalization with CTGs
Basic				✓	
BetterCTI		✓		✓	
BetterPropagation		✓	✓	✓	
PriorityQueue	✓	✓	✓	✓	
CTG		✓	✓		✓

Figure 3.1: A summary of the different model-checker variants implemented.

I provide an explanation of the implementation of each of the four main components in turn.

3.1 Parser

The parser component parses both ASCII or binary-formatted AIGER files and assumes that the new format is used (because all old format AIGER files are also new instances of the new format). The justice properties and fairness constraints are ignored by the parser, as they are not used by the model checker.

Both the `Parser.AigerParser` module that implements the parser in Haskell and the `Parser.AigerTools` module that calls the Aiger Utilities' parser's functions parse the AIGER file into the `Model` data structure in `Parser.AigModel`, which stores the components specified in the AIGER file.

3.1.1 Model

More specifically, the `Model` data structure stores the number of variables and the number of inputs. It also stores as a list of literals the outputs, bad states, and invariant constraints. The data structure also stores latches and AND gates as lists of lists of literals. I discuss the representation of literals, latches, and AND gates below.

Literals are represented by a `Lit` data structure in the `Parser.AigModel` module, which stores decoded versions of AIGER format indices: The `Lit` data structure has a constructor `Boolean` that takes a `Bool` argument for representing the boolean values that correspond with AIGER indices 0 and 1, a `Var` constructor that takes a `Word` for representing a positive literal, and a `Neg` constructor that takes a `Word` for representing a negative literal. Variable names are adjusted (by subtracting 1) so that they start at 0.

```
data Lit = Var Word | Neg Word | Boolean Bool
```

Figure 3.2: The `Lit` data structure in *Parser.AigModel*

For example, the index 3 read from an AIGER file is parsed into `Neg 0`: the odd index 3 indicates that it is a negative literal of the variable named 1, and subtracting by 1 gives the new variable name 0.

Latches and AND gates are represented using three-element `Lit` lists. For latches, the first element gives the variable name of the latch (as a positive literal), the second gives the next-state literal, and the final element gives the initial state of the latch. For AND gates, the first element gives the variable name of the AND gate (as a positive literal), and the next two elements give the literals whose values are taken as inputs to the AND gate.

3.2 MiniSat Interface

MiniSat serves as the SAT solver for this implementation of the IC3 algorithm. Because the Haskell Foreign Function Interface cannot interface with C++ directly, the interface to the MiniSat SAT solver is composed of a C wrapper for the relevant MiniSat functions and classes and a Haskell interface to the C wrapper.

Much of the C wrapper is straightforwardly as follows: every MiniSat class is replaced with a C type, and every MiniSat function is replaced with a function with an `extern C` function that calls the MiniSat C++ function, as in figure 3.3.

```
extern "C" int addMinisatClause (Minisat::SimpSolver* solver,
                                 Minisat::vec<Minisat::Lit>* ps) {
  return solver -> addClause (*ps);
}
```

Figure 3.3: The C wrapper function for `addClause` from `CSolver.cpp`

I also added an additional `result` struct to allow for all the results of a SAT query to be returned from a single function call. The struct contains not only the result of the

SAT query, but also pointers to the model and conflict vector (if any) of the `Solver`. The wrapper function `solveWithAssumps` for the version of `solve()` that takes an assumption vector as an argument and returns a pointer to a `result` struct rather than just whether or not the query was satisfiable.

```
struct result {
  unsigned solved;
  unsigned modelSize;
  unsigned conflictSize;
  minisatLbool* model;
  litptr* conflict;
} res = {0, 0, 0, 0, 0};
```

Figure 3.4: The `result` struct from `CSolver.h`

The Haskell interface makes use of the Haskell FFI as well as the `hsc2hs` preprocessor for handling the `result` struct.

Using just the Haskell FFI for calling the C functions does not provide a sufficient abstraction for use by the rest of the model checker. I wrote further functions to allow for a more natural interface to MiniSat, making use of `unsafePerformIO` to have the functions return values outside the `IO` monad.

Many of the functions and datatypes in the interface are analogous to functions and structs in the C wrapper and C++ implementation of MiniSat. For example, the `Solver` datatype is an analogue to the MiniSat `Solver` object, and itself contains a pointer to an instance of a MiniSat `Solver` object. Similarly, functions such as `solveWithAssumps` work analogously to the C wrapper's `solveWithAssumps`, returning a `Result` that contains whether or not the query was satisfiable and the model or conflict vector (if any).

The information kept in a `Result` is taken directly from the `result` returned by the C Wrapper functions. I used the `hsc2hs` preprocessor to help handle pointer offsets when unmarshalling from the C struct. Beyond straightforward unmarshalling, some additional work to convert from the MiniSat representation of literals to the model checkers representation of literals was necessary.

3.3 Hardware Models

3.3.1 Representation

Literals and Clauses

The `Lit` data structure in `Model.Model` gives the representation for literals in the model checker. The `Var` constructor gives positive current-state (unprimed) literals, the `Neg` constructor gives negative current-state literals, and the `Var'` and `Neg'` constructors respectively give positive and negative next-state (primed) literals. A clause is represented with type `Clause`, where each `Clause` is a list of the `Lit`s in the clause.

Transition Systems and Safety Properties

The representation of transition systems and the safety property for the model checker to check are both encompassed in the `Model` data structure in `Model.Model`, which serves as the representation of the hardware in the model checker. The inputs i and state variables x in the transition system $T(i, x, I, T)$ are not distinguished, and the total count of variables is kept in `vars`. Clauses that specify the initial state I are kept in `initial`. The transition relation T is captured by `transition`, a list of both clauses that specify latches and clauses that specify AND gates. The literal that gives the safety property is given by `safe`.

```
data Model = Model { vars :: Word
                   , initial :: [Clause]
                   , transition :: [Clause]
                   , safe :: Lit } deriving Show
```

Figure 3.5: The data structure for representing the hardware model in the model checker.

3.3.2 Construction

The `Model.Model` module contains functions to convert the `Model` data structure from the `Parser.AigModel` module into the hardware model representation used by the model checker. In particular, the `toModel` function takes an `Parser.AigModel.Model` and outputs a `Model.Model.Model`. As mentioned before, the `Model.ModelLit` data structure only has constructors for variables and their negations; `Lit`s from the `Parser.AigModel` module are either converted to `Model.Model.Lit`s or, in the case that they use the `Boolean` constructor, are removed from the model during the conversion of the `Latch` and `And` components to `Clause`s in `Model.Model`.

Latches

The `makeLatches` function generates a pair of `Clause` lists for a list of `Parser.AigModel.Latch`es, where the first `Clause` list contains clauses whose conjunction describes the latches' initial values, and the second `Clause` list contains a clauses whose conjunction describes the latches' next-state values.

Consider a given `Parser.AigModel.Latch [latchVar, next, init]` representing the latch with output variable l (represented by `latchVar`), next-state n (represented by `next`) taken from the set of booleans and literals, and initial value (represented by `init`) also taken from the set of booleans and literals. The `makeLatches` function uses the values of l, n, and i to generate `Clause`s that describe the latches' initial values and next-state values.

The generation of the initial value clause of the latch proceeds as follows: if $i = \mathit{True}$, then the singleton clause $\{l\}$ is generated for the initial value list, and if $i = \mathit{False}$, then the singleton clause $\{\neg l\}$ is generated. If i is a literal rather than a boolean value, then the latch is uninitialized and no clauses are generated for its initial value.

The generation of next-state clauses proceeds similarly: if $n = \mathit{True}$, then the singleton clause $\{l'\}$ is generated because the next-state value for the variable is a constant-*True* value, and if $n = \mathit{False}$, then the singleton clause $\{\neg l'\}$ is generated. Otherwise, the next-value clauses generated for l, are $\{l', \neg n\}$ and $\{\neg l, , n\}$. The conjunction of these clauses are, as needed, logically equivalent to $n \Rightarrow l'$, i.e., where $\simeq$ denotes logical equivalence, the following hold:

$$
\begin{aligned}
l' \Leftrightarrow n &\simeq (l' \Rightarrow n) \wedge (n \Rightarrow l') \\
l' \Rightarrow n &\simeq \neg l' \vee n \\
n \Rightarrow l' &\simeq l' \vee \neg n.
\end{aligned}
$$

It follows that the original double implication is equivalent to the the CNF formula that corresponds to the generated clauses:

$$l' \Rightarrow n \simeq (\neg l' \vee n) \wedge l' \vee \neg n.$$

AND gates

The `makeAnds` function generates a single `Clause` list for a list of `Parser.AigModel.And`s, where the conjunction of the clauses in the list describes the relationship between the AND-gate output and the AND-gate inputs.

Consider a `Parser.AigModel.And`, of the form `[andVar, in1, in2]`, representing the AND gate with output variable a (represented by `andVar`), and inputs i_1 and i_2 (represented by `in1` and `in2`). The `makeAnds` function uses the values of a, i_1, and i_2 to generate the appropriate `Clause`s that describe the AND gates' values.

If both i_1 and i_2 are booleans (corresponding to both `in1` and `in2` using the `Boolean` constructor for `Parser.AigModel.Lit`s), then a singleton clause suffices to describe the AND gate. If $i_1 \wedge i_2$ holds, then the singleton clause $\{a\}$ describes the constantly *True* AND gate, and if not, then the singleton clause $\{\neg a\}$ describes the constantly *False* AND gate.

If only one of the inputs (i_1 and i_2) is a boolean value, then the clauses equivalent to $a \Leftrightarrow i$ are generated, where i is the input that is not a boolean value and the clauses to generate for $a \Leftrightarrow i$ are described above in the explanation for generating clauses for latches.

If neither of i_1 or i_2 are boolean values, then the clauses generated are $\{\neg a, i_1\}$, $\{\neg a, i_1\}$, and $\{\neg i_1, \neg i_2, a\}$. The conjunction of these clauses are, as needed, logically equivalent to $a \Leftrightarrow i_1 \wedge i_2$:

$$
\begin{aligned}
a \Leftrightarrow i_1 \wedge i_2 &\simeq (a \Rightarrow i_1 \wedge i_2) \wedge (i_1 \wedge i_2 \Rightarrow a) \\
a \Rightarrow i_1 \wedge i_2 &\simeq \neg a \vee (i_1 \wedge i_2) \\
i_1 \wedge i_2 \Rightarrow a &\simeq \neg(i_1 \wedge i_2) \vee a
\end{aligned}
$$

Distributing $\vee$ over $\wedge$ gives further equivalence

$$\neg a \vee (i_1 \wedge i_2) \simeq (\neg a \vee i_1) \wedge (\neg a \vee i_2),$$

and using deMorgan's laws gives equivalence

$$\neg(i_1 \wedge i_2) \vee a \simeq \neg i_1 \vee \neg i_2 \vee a.$$

It follows that the original double implication is equivalent to the CNF formula that corresponds to the generated clauses:

$$a \Leftrightarrow i_1 \wedge i_2 \simeq (\neg a \vee i_1) \wedge (\neg a \vee i_2) \wedge (\neg i_1 \vee \neg i_2 \vee a).$$

3.4 Model Checking

I have implemented several versions of the recursive IC3 algorithm: the most basic version (*Basic*), a version that improves upon the most basic version by discovering smaller CTIs (*BetterCTI*), and a version that improves upon the version with improved CTIs by considering subsumed clauses (*BetterPropagation*).

I have also implemented a variation of IC3 that uses priority queues (*PriorityQueue*) and a variation that uses CTGs to improve generalization (*CTG*).

3.4.1 Overall structure

The general structure of the algorithm in the implementations is similar to the structure given in figure 2.5; however, there are some small differences that result from implementing the algorithm in a functional language and an adjustment to how the propagation phase is carried out.

To explain the modifications to the structure of the algorithm, I give pseudocode in figure 3.6 that outlines the general structure shared by all the implementations of the model checker except *PriorityQueue* and compare this structure with figure 2.5. I provide an explanation of the variant used in the *PriorityQueue* implementation is provided in section 3.4.8.

Because the implementation of the model checker is in Haskell, the overall structure of the algorithm has been modified to be recursive rather than iterative. The *prove* function makes an initiation query, and, if it succeeds, calls the *prove'* function that corresponds the main while loop in line 5 of figure 2.5 using recursion. The *proveNegCTI* function here does not correspond just to the *proveNegCTI* function in 2.5 but rather to the while loop that contains that function.

Because functions in Haskell are pure, the assumption made in figure 2.5 that function can could modify the set of (passed-by-reference) frames can no longer be made. Instead, the updated values of frames are returned explicitly from the function call in a tuple along with any other values needed from the function call. For example, *proveNegCTI* on line 9 in 3.6 returns not only the result indicating whether or not the negated CTI was proven, but also returns the possibly updated values for the frontier frame G_k and previous frames $G_0, \ldots, G_{k-1}$.

In addition to the necessary language-related modifications to the algorithm, I made a change to how often the full propagation phase is carried out, calling the *pushFrame* function instead where appropriate.

```
1  Function prove(M, P):
2      if ¬(I ⇒ P) then return False
3      return prove'(M, P, I, nil)
4  Function prove'(M, P, F_k, [F_0, ..., F_{k-1}]):
5      if F_k ∧ T ⇒ P' then
6          return pushFrame(F_k, ∅, M, P, [F_0, ..., F_{k-1}])
7      else
8          let cti = nextCTI(F_k ∧ T ⇒ P'),
9          (result, [G_0, ..., G_{k-1}], G_k) = proveNegCTI((i, x, I, T), cti, k − 1) in
10             if result then
11                 let (fixed, [H_0, ..., H_{k-1}, H_k]) = propagate([G_0, ..., G_{k-1}, G_k]) in
12                     if fixed then return True
13                     else return prove'(M, P, H_k, [H_0, ..., H_{k-1}])
14             else return False
15 Function pushFrame(F_{k-1}, F_k, M, [F_0, ..., F_{k-2}]):
16     let (fixed, G_k) = push(F_{k-1}, F_k) in
17         if fixed then return True
18         else return prove'(M, P, G_k, [F_0, ..., F_{k-2}, F_{k-1}])
```

Figure 3.6: General structure of the algorithm implementation. The transition relation T is acquired from the model M

The propagation phase of the algorithm is carried out by the *propagate* function. The actual implementation of the *propagate* function returns type `Maybe [Frame]`, but for the sake of discussing the high level structure of the implementation, it is assumed here to return a pair of a boolean value indicating whether a fixed point has been found while pushing clauses and a list of the updated frames.

While in figure 2.5, the propagation phase is called at each iteration of the algorithm, if the consecution query succeeds and a new frontier frame F_k is added in that iteration, then none of the frames have had any new clauses added to them. As a result, the only frame that modified during the propagation phase is the frame F_k because all the frames before F_{k-1} have already had all possible clauses pushed forward in previous iterations. Similarly, the only way a fixed point would be detected is if $F_{k-1} = F_k$, since all pairs of consecutive frames except (F_{k-1}, F_k) have been checked for equality.

In the case that the consecution query succeeds, considering pairs of frames other than (F_{k-1}, F_k) is unecessary work. The modified algorithm is such that that when the consecution query succeeds, it calls the *pushFrame* function that checks only a single pair of frames (which also makes the recursive call to *prove*). When the consecution query fails, the adjusted algorithm, like the original in figure 2.5, calls the *propagate* function to handle the updates to the frames made by *proveNegCTI*.

3.4.2 Frames

The `Frame` data structure represents frames in all implementations of the model checker. Along with the set of clauses (represented by a list of literals), a `Frame` also includes a `Solver`, which contains at least all the clauses in the frame's set of clauses. The `Solver` may also contain the `transition` clauses for the hardware model.

3.4.3 Initiation

The initiation query $I \Rightarrow P$ is an implication, but a MiniSat `Solver` can only solve queries given in CNF (with an optional assumption cube). As a result, the representation of the query $I \Rightarrow P$ for a frame I and a clause P makes use of the logical equivalence between $I \Rightarrow P$ and $\neg(\neg P \wedge I)$.

The implementation of the initiation query in figure 3.7 makes use of the fact that the formula $\neg(\neg P \wedge I)$ is true iff $\neg P \wedge I$ is unsatisfiable. The `initiation` The implementation also makes use of deMorgan's laws as described in section 2.4 to acquire the assumption cube by negating all the literals in the `prop` clause.

```
initiation :: Frame -> Clause -> Bool
initiation f prop =
  not (satisfiable (solveWithAssumps (solver f) (map neg prop)))
```

Figure 3.7: The initiation query implementation.

3.4.4 Consecution

Similarly to how the implication in the initiation query is converted to an equivalent CNF formula, all implementation variants use the logical equivalence of $F_k \wedge T \Rightarrow P'$ and $\neg(\neg P' \wedge F_k \wedge T)$ along with deMorgan's laws to yield the implementation in figure 3.8.

```
consecution :: Frame -> Clause -> Bool
consecution f prop =
  not (satisfiable (solveWithAssumps (solver f) (map (prime.neg) prop)))
```

Figure 3.8: The consecution query implementation.

3.4.5 Counterexamples to Induction

Counterexamples to induction are found by the `nextCTI` function, which uses results from SAT queries to find a full or partial assignment to the variables in the `Model` from which the CTI can be extracted.

Basic

In the *Basic* implementation of the IC3 algorithm, `nextCTI` asks for a model (i.e. the set of true literals) for the satisfiable query $\neg P' \wedge F_k \wedge T$. The current-state literals then

give a predecessor state (a state from which a $\neg P$ state can be reached in one step of the transition relation) for $\neg P$, i.e., the current-state literals give the CTI. These current-state literals are extracted from the model in the function that called `nextCTI`.

Smaller Counterexamples to Induction

In the all implementations of the algorithm other than *Basic*, `nextCTI` again asks for a model m for the satisfiable query $\neg P' \wedge F_k \wedge T$. The only literals in m that must necessarily be included in the CTI are those current-state literals that result in the unsatisfiability of $m \wedge P' \wedge T$. That is, the current-state literals of any subcube q of m for which $q \wedge P' \wedge T$ holds is also a valid CTI, with the state m being one of the states in the set represented by q.

The conflict vector resulting from querying the SAT solver with $P' \wedge T$ and assumption cube m contains such a q that has only literals relevant to the conflict. This q is then returned to the calling function, which, as in the *Basic* implementation, extracts the current-state literals from q to obtain the CTI.

3.4.6 Propagation

Both the implementation of the *pushFrame* function and the implementation of the *propagate* function in figure 3.6 and figure 3.11 (which describes the structure of the *priorityQueue* implementation) rely on the implementation of the *push* function, which has two variants described below.

Basic

The *Basic* and *BetterCTI* implementations' `push` function, when invoked as `push f model f'` tries to push all clauses in `Frame f` that are not in `Frame f'` to `f'` and results in a pair containing a `Bool` indicating whether a fixed point has been reached (i.e., all clauses could be pushed) and a `Frame` with all the clauses in `f'` and all the clauses in `f` that could be pushed to `f'`. For each clause in `f` that is not in `f'` the `consecution` function is called to see if the clause is inductive relative to the frame represented by `f`. If it is, then the clause can be conjoined to the frame represented by `f'`, and if it is not, then the function must have `False` as the first element in the pair it returns.

Subsumed clauses

The *Basic* and *BetterCTI* implementations' `push` function avoids unnecessary consecution queries by only considering clauses in `f` that are not in `f'`. Further consecution queries may be eliminated by considering the clauses in `f` that are subsumed by other clauses, which is done by all implementation variants other than *Basic* and *betterCTI*.

A clause c *subsumes* a clause c' if the literals in c are a subset of the literals in c'. In this case, $c \Rightarrow c'$ holds, so c' can be removed from the set of clauses. By removing all subsumed clauses c' from a frame before trying to push clauses, the model checker can avoid making the consecution queries that arise from attempts to push those clauses.

```
inductiveGeneralization :: Clause -> Frame -> Frame -> Model -> Word
                        -> Clause
inductiveGeneralization clause f0 fk m = generalize clause f0 fk []
  where
    generalize cs _ _ needed 0 = cs ++ needed
    generalize [] _ _ needed _ = needed
    generalize (c:cs) f0 fk needed k =
      let res = solveWithAssumps
                  (solver (getFrameWith ((cs ++ needed):clauses fk) m))
                  (map (prime.neg) (cs ++ needed))
        if not (satisfiable res) && initiation f0 cs
          then generalize cs f0 fk needed k
          else generalize cs f0 fk (c:needed) ( k - 1 )
```

Figure 3.9: The `inductiveGeneralization` function that approximates the *mic* algorithm.

The versions of `push` that consider subsumed clauses include a call to the function `removeSubsumed` when acquiring the list of clauses to attempt to push. The `removeSubsumed` function takes a list of clauses and removes all clauses in the list that are subsumed by other clauses in the list. The `push` function replaces the frame `f` with a version of `f` with all the subsumed clauses in the frame removed for the rest of the function and proceeds as the basic implementation's `push` function does, returning a triple containing the updated `f` along with the fixed-point `Bool` and updates `Frame f'`.

3.4.7 Inductive Generalization

Finding the minimal inductive subclause (MIC) for a clause is in practice inefficient, and all implemented versions of generalization `inductiveGeneralization` involve approximating the MIC with a call to the function `generalize`.

Simple

The simplest approximation for a MIC involves attempting to drop each literal in turn and checking that the resulting clause c results in the truth of formulas $I \Rightarrow c$ and $F_k \wedge c \wedge T \Rightarrow c'$ that the original clause did. If the resulting clause does result in satisfiable results for both the queries, then the literal can be successfully dropped, but if not, the literal is added to a list `needed` of necessary literals. After a parameterizable number of failed attempts at dropping a literal from the clause or after having attempted dropping all the literals, the `inductiveGeneralization` function that implements this approximation returns the clause resulting from appending the remaining literals in the clause (i.e. the literals that the `generalize` has not tried to drop) with the literals in `needed`.

This corresponds to the algorithm described in figure 3.10, but where *down* simply checks for the relative inductiveness of the subclause and does not attempt to expand it.

```
1  Function down(cls, i):
2      if ¬(I ⇒ cls) then return False
3      if F_i ∧ cls ∧ T ⇒ cls' then return True
4      ctg:= model extracted from SAT query F_i ∧ cls ∧ T ⇒ cls' if I ⇒ ¬ctg and
       F_i ∧ ¬ctg ∧ T ⇒ ¬ctg' then
5          j := 0
6          while F_j ∧ ¬ctg ∧ T ⇒ ¬ctg do j := j + 1
7          generalizedNegCTG := mic(¬ctg, j)
8          F_j := F_j ∪ {generalizedNegCTG}
9          return down(cls, i)
10     else
11         p := F_i ∧ t state such that F_i ∧ t ∧ p ⇒ ¬t'
12         cls := cls ∩ p
13         return down(cls,i)
```

Figure 3.10: The algorithm for the version of *down* that handles CTGs.

Minimal Inductive Subclauses and Counterexamples to Generalization

The more elaborate implementation of generalization implements the full (but limited in number of attempts) MIC algorithm with the *down* function modified so that it deals with CTGs.

The modified *down* algorithm checks, as in the simple approximation for MIC, for the satisfiability of $I \Rightarrow c$ and $F_k \wedge c \wedge T \Rightarrow c'$, where c is the subclause passed to the algorithm. The difference is that *down* does not immediately attempt to expand c if $I \Rightarrow c$ is true and $F_k \wedge c \wedge T \Rightarrow c'$ is not; in this case, the CTG *ctg* is acquired by taking the current literals in the model the SAT solver gives for $\neg c' \wedge c \wedge T \wedge F_k$.

The *down* algorithm then finds the deepest frame F_{j-1} for which $\neg ctg$ is inductive, and attempts to generalize $\neg ctg$ relative to that frame with a recursive call to the *mic* algorithm. The generalization of $\neg ctg$ can then be added to frame F_j, and *down* is called recursively using the updated set of frames.

The implementation of *down* is approximate, the Haskell function `down` that implements the algorithm takes a parameter `r` that limits the number of CTGs that it will handle for each non-recursive call to the implementation of the approximation of the *mic* algorithm.

3.4.8 Priority Queue Variant

The *PriorityQueue* implementation keeps track of what to prove next by using a priority queue of proof obligations instead of through recursive calls that explicitly specify which property to prove at which depth. The implementation of this variant of the algorithm makes use of some of the same functions (e.g. `negCTI` and `push`) as the others, but differs in its overall structure. I first provide a definition of proof obligations, an overview

```
1  Function prove(M, P):
2      if ¬(I ⇒ P) then return False
3      let queue = queue containing proof obligation (¬P, 1) in
4          return fulfillObligations(M, [I], queue)
5  Function fulfillObligations(M, [F_0, ..., F_k], queue]):
6      let ((s, i), q) = dequeue(queue) in
7          if F_{i-1} ∧ T ⇒ ¬s' then return pushFrame(M, [F_0, ..., F_k], q, (s, i))
8      else let cti = nextCTI(F_{i-1} ∧ T ⇒ ¬s') in
9          if I ⇒ ¬cti then
10             let (fixed, [G_0, ..., G_k], d) = propagate([F_0 ∪ {¬cti}, F_1, ..., F_k], ¬cti)
               in
11                 if fixed then return True
12                 return fulfillObligation(M, [G_0, ..., G_k], (generalize(¬cti, d), d))
13         else return False
14
```

Figure 3.11: General structure of the algorithm implementation in *PriorityQueue*.

of the structure of the implementation for this variation of the algorithm, and some implementation details about representing proof obligations and the priority queue.

Proof Obligations

A *proof obligation* is a pair (s, i) of a state s that is either a set of bad states or a counterexample to induction and a depth i. When the model checker encounters a proof obligation (s, i) as the highest-priority element of the queue, it must prove $\neg s$ holds for all states reachable in at most i steps of the transition relation to fulfill (s, i).

Overall Structure

The variant of the algorithm used in the *PriorityQueue* implementation relies on a priority queue of proof obligations. When a proof obligation (s, i) is added to the priority queue, it is assigned a priority higher than any proof obligation in the queue (t, j) with $j > i$ and lower than any proof obligation in the queue (u, k) with $k \leq i$. The way that the implementation achieves this priority ordering is discussed later.

Unlike in the other implementations, in the *PriorityQueue* implementation, there is no distinction between the negation of the safety property P and any other property that needs to be proved. The priority queue maintains all the information about which properties need to be proven, and the main recursive *fulfillObligations* function attempts proves whichever property has the highest priority in the queue. In other words, the *fulfillObligations* function always attempts to fulfill the proof obligation with the highest priority in the queue (this proof obligation is the one returned by *dequeue*(*queue*) in line 6 of 3.11).

Whenever a proof obligation (s, i) is fulfilled at a certain depth i, the proof obligation $(s, i + 1)$ is added to the queue. Enqueueing the new proof obligation is valid because s states can reach $\neg P$ states in some number of steps of the transition relation and should therefore not be reachable in any number of steps of the transition relation from the initial state.

In attempting to fulfill a proof obligation (s, i), *fulfillObligations* proceeds generally in the same way as the other variants: if a consecution query succeeds, then *pushFrame* is called, and if not, a CTI is discovered with the intent to prove its negation is inductive relative to frame F_{i-1}.

The structure of the *pushFrame* function is modified to accomodate priority queues and the fact that the pair of frames may not be the pair the greatest possible depth. The *pushFrame* function pushes clauses from frame F_{i-1} to frame F_i (where F_i is not necessarily the frontier frame) and checks for the equality of F_{i-1} and F_i. The recursive call in *pushFrame* is then

$$\mathit{fulfillObligations}(M, [F_0, \ldots, F_{i-1}, G_i, F_{i+1}, \ldots, F_k], q),$$

where $q = \mathit{enqueue}((s, i + 1), \mathit{queue})$, the result of enqueueing the proof obligation for property s at the next depth $i + 1$ in the priority queue *queue*.

When a CTI c for proof obligation (s, i) is discovered the proof obligation $(c, i - 1)$ for proving the negation of the CTI could enqueued before calling calling *fulfillObligations* recursively again, but the implementation employs a different approach that keeps the number of generalization attempts low by generalizing once when the proof obligation for the CTI is enqueued rather than generalizing each time a proof obligation is fulfilled.

The approach employed by the *PriorityQueue* implementation checks that $I \Rightarrow \neg c$, adds $\neg c$ to F_0, and then uses a modified version of *propagate* to push clauses and check for fixed points up to depth $j \leq i$, where j is the greatest value that is less than i such that $\neg c$ is inductive relative to F_{j-1}. If a fixed point is found, then the algorithm can terminate. Otherwise, the clause $\neg c$ is generalized relative to frame F_{j-1} using the simpler approximation for finding MICs, giving clause $\neg d \subseteq c$. The proof obligation (d, j) is then enqueued, and *fulfillObligations* calls itself recursively.

Proof Obligations and Priority Queues

The *PriorityQueue* implementation represents proof obligations (s, i) using the `Obligation` type, which is a triple `(Int, Int, Clause)` of the depth i, a rank for deciding the ordering of proof obligations at the same depth within the priority queue, and the clause $\neg s$. Because of this representation of obligations, the function implementing *fulfillObligations* is named `proveObligations`.

In the *PriorityQueue* implementation, the priority queue is represented by a `MinQueue` (the minimal element has the highest priority) of `Obligations`.

For example, the initial `MinQueue` created after the successful initiation query is given by `singleton (1, 0, [prop])`, which represents the priority queue that contains only `Obligation (1, 0, [prop])`, representing the proof obligation $(\neg P, 1)$, where the clause P is the one that `[prop]` represents.

Chapter 4

Evaluation

The different variants of the model checker were run on fourteen handwritten examples and one hundred examples taken from across HWMCC'10, HWMCC'11, and HWMCC'13 (several examples are common to the competitions from different years). If the elapsed time for attempting to solve an example took longer than ten minutes, the attempt was considered to have timed out. The parameterizable number of failed attempts at dropping literals in the `inductiveGeneralization` functions were set to set to three, and the parameterizable number of CTGs that each generalization attempt in the *CTG* implementation will handle is also set to three.

The handwritten examples served as the "small examples" that the model checker was meant to correctly solve as part of the aims of the project, with the largest (in terms of number of variables) of the handwritten examples, `simple_counters.aig`, having 82 variables. The model checker implementations were not only able to solve all the handwritten examples correctly, but to solve fifty-one of the hundred examples taken from the Hardware Model Checking Competitions within ten minutes as well.

Following a description of the output of the model checker, I discuss the solving capabilities of the model checking implementations and compare the performance of the model checker implementations were compared with each other and with the reference implementation, discussing the possible causes of the differences among implementations.

4.1 Output

The output of all model checker implementations gives the value `True` if the safety property holds (i.e. if a bad state is not reachable from the initial state) and `False` if it does not. The implementations also provide debug output that gives statistics on solving if a nonzero number of frames was required to solve the example. In particular, all variants' outputs give the number of frames, the average number of literals per clause, the number of CTIs found, and the number of queries made for solving that example. The *CTG* implementation also reports the number of CTGs found. Sample output for the *CTG* implementation run on example `counters3.aig` is given in figure 4.1.

```
Number of frames: 21
Average number of literals/clause (not counting transition relation): 4.394657835488733
Number of ctis: 91
Number of ctgs: 242
Number of queries: 15225
True
```

Figure 4.1: Sample output for running the *CTG* implementation on example `counters3.aig`.

4.2 Solving

For the fourteen handwritten examples and fifty-one Hardware Model Checking Competition examples that all the model checker implementations solved without timing out, the implementations' solutions agree with the solutions given by Aaron Bradley's reference implementation, providing evidence for the correctness of the solutions given by the model checker implementations.

The largest (in terms of number of variables) unsafe example that the model checkers gave a solution for without timing out was `bj08goodbakerycyclef7.aig` from HWMCC'10, which has 19900 variables. The largest example for which the safety property holds that the model checkers gave a solution to without timing out was `pdtvsar8multip26.aig` with 7174 variables. Note that the number of variables in an example is only weakly correlated with the amount of time needed to solve the example.

4.3 Benchmarking

Benchmarks were taken for the performance of the different versions of the model checker and the reference implementation. For each example, forty benchmarking samples were taken.

Other than timing data, I also collected data about the number of frames needed to solve an example, the average number of literals per clause, the number of CTIs discovered, the number of SAT-solver queries, and, for the *CTG* implementation, the number of CTGs discovered.

The majority of examples for which the implementations did not time out did not require finding any CTIs; for these examples, the the *Basic*, *BetterCTI*, *BetterPropagate*, and *CTG* implementations give similar results.

4.4 Performance Impact of Variations

Profiling has consistently revealed that functions in the `MiniSat.Minisat` module consume the most time when solving examples, suggesting that the overall performance of the model checker is heavily dependent on the size and number of SAT-solver queries.

Figure 4.2: Average literals per clause averaged over all examples.

Figure 4.3: Average performance averaged over all samples for all examples.

Discovering smaller CTIs leads, as expected, to a smaller average number of literals per clause. This smaller number of literals per clause suggests smaller SAT queries, and *betterCTI* exhibits better performance than the *Basic* implementation for examples that require finding at least one CTI. The *Basic* implementation was able to solve only three unsafe examples (`shortp0.aig`, `shortp0neg.aig`, and `srg5ptimoneg.aig`) from the Hardware Model Checking Competitions and no safe examples from the Hardware Model Checking Competitions that required finding CTIs without timing out, but the *BetterCTI* version of the implementation (and all other implementations that include finding smaller CTIs) solved seven more examples than the *Basic* version without timing out.

Also as expected, removing subsumed clauses results in a smaller number of literals per clause, resulting in the *BetterPropagation* version having slightly better performance than the *BetterCTI* version. While the performance impact that the improvement has is less dramatic than the improvement of *BetterCTI* over *Basic* finding smaller CTIs, there is still a consistent improvement of benchmarked times for examples that require more than one frame.

4.4.1 Counterexamples to Generalization

The *CTG* version that deals with CTGs performs the same as or worse than the *BetterPropagation* version on examples, even in cases where *CTG* reduces the average number of frames per clause, most likely because the examples used are too small for the performance benefits of using CTGs to eliminate more states to overcome the overheads of finding and proving negated CTGs. Similar results can be found in the performance of Aaron Bradley's reference implementation with basic generalization and improved (CTG-using) generalization on the same examples: for these examples, the reference implementation performs better with CTG-handling disabled.

4.4.2 Priority Queues

The advantage of the priority queue implementation is that CTIs do not need to be rediscovered: after a proof obligation (s, i) is enqueued, until the algorithm fails or finds a fixed point, the queue will always contain a proof obligation (s, j) for $j \geq i$. When the proof obligation (s, i) fulfilled at a certain depth i, $(s, i + 1)$ is then enqueued, If s is a CTI for proving a property p at depth $i + 2$ (i.e. proof obligation $(\neg p, i + 2)$), by the time `proveObligations` removes proof obligation $(\neg p, i + 2)$ from the priority queue, $(s, i + 1)$ has already been fulfilled, so the CTI s would not, after its initial discovery, need to be discovered again.

Even if s is not a CTI for proving p at depth $i + 2$, the proof obligations $(s, i + 1)$ would still need to be fulfilled before `proveObligations` attempts to fulfill $(\neg p, i + 2)$.

4.5 Reference Implementation

The average performance of the most efficient implementation of the model checker across all examples is

compared to the average performance of Aaron Bradley's reference implementation (with or without generalization involving CTGs enabled).

The choice of implementation language may account for much of the difference in performance, as the reference implementation in C++ has more control over memory allocations than the implementations in Haskell, which is a garbage-collected language. I mention other differences between the implementations that may explain some of the performance differences below.

Model Representation

The reference implementation differs from this project's implementations in representing the hardware model, which may account for some of the performance differences.

The reference implementation keeps track of which variables are inputs, latches, and AND gates. Each `Model` maintains both the the primed and current values for inputs and latches and keeps a table to memoize the values of AND gates.

As mentioned earlier, when the consecution query $F_k \wedge T \Rightarrow P'$ fails, this corresponds to the CNF query $F_k \wedge T \wedge \neg P'$ being satisfiable, and while a full satisfying assignment s gives a CTI state, it is better to use a set of states $c \subset s$ as a CTI cube, so that several CTI states can be eliminated at once. The `stateOf` function uses the information kept in `Model`s to extract the smaller cube from the model s giving the satisfying assignment for a failed consecution query directly, without further SAT-solver queries. The Haskell implementation instead uses several SAT-solver queries to extract the necessary literals from s.

MiniSat

The reference implementation is more closely coupled to MiniSat's implementation. Because both the reference implementation and MiniSat are in C++, the reference implementation can and does call MiniSat functions and instantiate MiniSat objects, such as `SimpSolver`s directly. In contrast, the Haskell implementation must interact with MiniSat through an interface and suffers from associated overheads, such as those from marshalling data from the data structures returned from the C wrapper for MiniSat into the corresponding Haskell data structures.

The reference implementation also makes use of empirical results to improve the performance of MiniSat queries. For example, in the `stateOf` function, which extracts a model from a failed consecution query (to e.g. find a CTI cube), the set of literals passed to the MiniSat `Solver` are reordered according to an ordering of that was found to be the best choice empirically.

Overall Structure

The reference implementation uses a priority queue, but handles proof obligations differently than the *PriorityQueue* implementation does. For a CTI s that prevents the fulfillment of a proof obligation at depth i, the reference implementation enqueues a CTI $(s, i-1)$ and performs generalization relative to the frame F_{i-1} each time $\neg s$ has been shown to be inductive relative to frame F_{i-1} (generalization does not seem to be as expensive for the reference implementation). The reference implementation also does not enqueue a new proof obligation $(s, i+1)$ each time a proof obligation (s, i) has been fulfilled as the *priorityQueue* implementation does. The safety property is maintained and handled separately from CTIs; its negation is not included as part of a proof obligation placed in the priority queue.

Chapter 5

Conclusion

The project aims to implement a basic version of the IC3 algorithm in Haskell with the necessary parser and SAT-solver interface have been achieved, and the goal of the model checker being able to check small example hardware model has also been reached. Furthermore, several variants of the model checker have been implemented, and the implementations of the model checker can model check not only my handwritten hardware model examples but also models from the Hardware Model Checking Competitions with hundreds of thousands of variables.

5.1 Summary

The IC3 algorithm is a recently-developed SAT-based model-checking algorithm notable for its method of using counterexamples to induction to discover new invariants.

5.2 Further extensions

Rather than implement the extensions mentioned in the initial project proposal, I elected to implement the *CTG* and *PriorityQueue* variants of the model checker and examine their effects on the model checker's performance. As a result, possible further extensions still include those mentioned in the initial project proposal: interfaces to different SAT solvers and abstraction-refinement could be implemented and their effects on the performance of the model checker implementations could be analyzed.

Bibliography

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, pages 317–320, New York, NY, USA, 1999. ACM.

[2] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

[3] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. *Computer Aided Verification (CAV)*, chapter Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR), pages 831–848. Springer International Publishing, 2014.

[4] Aaron Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 70–87, 2011.

[5] Aaron Bradley. Understanding IC3. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 1–14, 2012.

[6] Alessandro Cimatti and Alberto Griggio. *Computer Aided Verification (CAV)*, chapter Software Model Checking via IC3, pages 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[7] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, chapter Model Checking and the State Explosion Problem, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[8] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Andrei Voronkov, editor, *Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, 2003.

[9] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134. IEEE, 2011.

[10] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. *Computer Aided Verification (CAV)*, chapter Incremental, Inductive CTL Model Checking, pages 532–547. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[12] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–125, London, UK, 2000. Springer-Verlag.

[13] Yakir Vizel, Orna Grumberg, and Sharon Shoham. Lazy abstraction and SAT-based reachability in hardware model checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2012.

Appendix A

Project Proposal

Introduction and Description of the Work

Model checking is one way of assessing whether or not a hardware or software system has certain properties. For example, model checkers can be used to check systems for safety properties by finding examples of states that violate the properties or by proving that all states have the properties.

Explicit-state model checking can be infeasible for systems with a large number of states, but symbolic model checking, which represents states and the transition relation between them as boolean expressions, can handle more states. Symbolic model checking initially relied on the efficient representation of boolean expressions through binary decision diagrams (BDDs), but BDDs can still consume a large amount of space, and finding an ordering for BDD variables that keeps the BDDs small can become costly [2].

Symbolic model checking techniques that rely on SAT solvers provide an alternative to BDD-based approaches. SAT-based approaches include bounded model checking [2] and k-induction [8], but both of these approaches involve unrolling the transition relation, which can lead to long SAT solver queries.

IC3 [4] is a more recently developed SAT-based algorithm for the symbolic model checking of safety properties. Instead of unrolling the transition relation and considering entire paths, IC3 maintains a set of frames $F_0, ..., F_k$, where each frame F_i is an overapproximation of the set of states reachable in at most i steps, and considers at most one step of the transition relation from a particular frame at a time. As a result, IC3 can find inductive strengthenings that tend to be smaller and more convenient than those found by BMC-based techniques such as k-induction, which finds strengthenings that are the negations of spurious counterexample paths [8], and the SAT queries that IC3 makes tend to be simpler [5].

This project focuses on implementing a symbolic model checker for verifying safety properties of hardware. The model checker will include a new implementation of the IC3 algorithm in Haskell, which will make use of an existing SAT solver.

Starting Point

I begin the project with some experience programming in Haskell from a summer internship and no experience with model checking or using a SAT solver. I have informally acquired some knowledge about model checking to formulate this project idea.

Substance and Structure of the Project

The project aims to implement a hardware model checker that takes its inputs in AIGER format and queries the MiniSat SAT solver.
The structure of the project can be broken down into the following components:

1. **Parsing AIGER format** The model checker takes its inputs in AIGER format and will, as a result, require an AIGER parser. The AIGER format is fairly simple and hand-coding a parser for it should be suitable.

2. **Interfacing with MiniSat** The model checker will be using the MiniSat SAT solver, so an API that allows the model checker to query MiniSat will be required.

3. **Implementing the IC3 algorithm** The main aspect of the project is the implementation of the IC3 algorithm. The implementation will largely be based on the algorithm as described in [4, 5].

4. **Evaluating the model checker** The model checker will be evaluated by measuring its performance on checking examples. Though the project does not focus greatly on the efficiency of the implementation, it may still be interesting to see how the performance of this IC3 implementation in Haskell compares with other implementations. As a result, benchmarks taken for the model checker will be compared with further benchmarks taken for Aaron Bradley's reference IC3 implementation, which is implemented in C++. Given that the reference implementation takes its inputs in AIGER format and also uses MiniSat, the benchmarks should provide a means to compare the IC3 implementations specifically.

5. **Writing the dissertation**

Possible Extensions

If the aforementioned aspects of the project are completed, carrying out the following extensions could be possible:

- Interfacing with other SAT solvers, and possibly performing additional benchmarking; comparing the performance of the model checker when used with different SAT solvers may be of interest since the performance of IC3 implementations tend to vary considerably depending on the characteristics of the underlying SAT solver.

- Model checking properties of real hardware as a case study.

- Implementing abstraction-refinement as described in [13].

Success Criteria

The project will be a success if the following have been completed:

- The AIGER parser has been implemented.
- The MiniSat interface has been implemented.
- The IC3 algorithm has been implemented.
- The model checker should be able to solve some small examples.

Timetable: Workplan and Milestones

1. **16 October 2015 – 28 October 2015**

   Preliminary reading. Get familiar with the AIGER format, MiniSat and relevant Haskell libraries and tools for implementing the components of the project.

2. **29 October 2015 – 4 November 2015**

   Write an AIGER format parser.
   Milestone: Parser completed. Relevant information from AIGER files can be extracted.

3. **5 November 2015 – 18 November 2015**

   Implement a MiniSat interface.
   Milestone: MiniSat interface completed, enabling the model checker to use MiniSat to solve SAT problems.

4. **19 November 2015**

   Begin implementing the IC3 algorithm.

5. **Michaelmas vacation**

   Continue implementing the IC3 algorithm.

6. **14 January 2016 – 27 January 2016**

   Write progress report. Finish implementation of the IC3 algorithm.
   Milestones: Progress report completed. Working implementation of the model checker completed.

7. **28 January 2016 – 10 February 2016**

   Measure and compare this IC3 implementation's performance and the reference implementation's performance.
   Milestone: Evaluation completed.

8. **11 February 2016 – 11 March 2016**

   Write the main parts of the dissertation.
   Milestone: Finished writing main parts of dissertation: introduction, preparation, implementation and evaluation chapters.

9. **Easter vacation**

   If necessary, use this time for catching up. Otherwise, work on extensions, starting with interfacing with other SAT solvers. Finish writing dissertation.
   Milestones: All implementation and evaluation completed. Draft dissertation completed.

10. **21 April 2016 – 4 May 2016**

    Proofread and edit dissertation as necessary.
    Milestone: Dissertation ready for submission.

11. **5 May 2016 – 13 May 2016**

    Time left for catching up in case any delays have occurred in the completion of any milestones.
    Milestone: Dissertation submitted.

Resources Required

For the project I will mostly make use of my laptop, which runs OS X 10.8. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. If my main computer fails, I will use MCS computers. I will use GitHub for backup and git for revision control.
I will also be using:

- AIGER utilities, available `http://fmv.jku.at/aiger/`
- MiniSat, available `https://github.com/niklasso/minisat`
- Models from the Hardware Model Checking Competition, such as those available `http://fmv.jku.at/hwmcc10/`
- Aaron Bradley's Reference IC3 implementation, available `https://github.com/arbrad/IC3ref`