Lauren Pick

A Model Checker Using IC3

Computer Science Tripos – Part II

Homerton College

April 28, 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

The original aims of the project were to implement the basic IC3 algorithm as part of a model checker written in Haskell. This model checker should be able to solve several small example hardware models correctly.

Work Completed

I implemented the basic IC3 algorithm as part of a new model checker, which involved implementing several additional components: the AIGER parser, MiniSat interface, and hardware model representation. As an extension, I implemented other variants of the IC3 algorithm. I benchmarked the variants on fourteen handwritten examples and fifty-six examples from the Hardware Model Checking Competition and compared results with those for the IC3 reference implementation. The implementation of the basic IC3 algorithm and its ability to solve the handwritten examples meets the project's goals; the implementation of other variants and their ability to solve additional examples exceeds these goals.

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-checking algorithm. 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.

Software and hardware bugs can be costly. The Pentium FDIV hardware bug cost an estimated $475 million [33], and the Ariane 5 software bug cost approximately $370 million [21]. The cost of such bugs catalyzed the adoption of new techniques for developing systems that would help reduce the number of or mitigate the effects of residual bugs. In particular, the Pentium FDIV bug encouraged the use of formal verification in hardware design throughout the semi-conductor industry.

Formal methods is an umbrella term that refers to any application of mathematics or logic in the improvement of the design or implementation of hardware or software systems. One such technique falling under the "formal methods" banner is formal verification, which focuses on proving properties of systems. For example, type checking provides a widely-used form of software verification, proving that programs are well-typed. The formal verification technique of model checking has been used to verify that circuits correctly implement the SRT algorithm that the Pentium processors with the FDIV bug did not [18].

Given a model and a specification of a system, a model checker will check whether or not the system satisfies the specification. Model checking is fully automated, unlike formal verification techniques that employ Hoare Logic or proof assistants, which require user guidance.

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 [19].

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 mitigating the effects of the state explosion problem. By representing states and the transition relation between them as logical formulas (details of which are provided in Section 2.4), symbolic model checking allows sets of states to be represented efficiently as logical formulas involving state variables instead of as an explicit list of each individual state in the set [31].

Symbolic model checking was originally invented for use with ordered binary decision diagrams (BDDs), data structures that provide an efficient representation of propositional formulas. For a particular variable ordering, a unique BDD represents each formula (and all equivalent formulas). 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 facilitates the model checking of systems with larger numbers of states than could be handled by explicit-state model checking [31].

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 [11].

An alternative to BDD-based symbolic model-checking techniques are SAT-based techniques, which use procedures for solving the Boolean satisfiability problem and, unlike BDD-based methods, do not use the canonical representations of propositional formulas. Such techniques include bounded model checking (BMC) [11], k-induction [34], and the IC3 algorithm that is the focus of this project.

1.2 The IC3 Algorithm

The IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) algorithm (also called Property-Directed Reachability [22]) is a state-of-the-art, SAT-based model-checking algorithm for proving the safety properties (i.e. properties that must hold in all reachable states) of hardware [15]. In this section, I outline some of the merits of the IC3 algorithm, with a description of IC3 following in Section 2.7.

The first implementation of the algorithm `ic3` written by Aaron Bradley placed third in the 2010 Hardware Model Checking Competition (HWMCC'10) [7], 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.

The IC3 algorithm has advantages when compared to other model checking techniques such as k-induction and BMC that allow it to prove more properties more efficiently. Unlike k-induction, the IC3 algorithm discovers new invariants based on the safety property that it is trying to establish of the system. As a result, these invariants are more relevant to proving the property than those found by k-induction [16]. 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 [16].

While the IC3 algorithm was designed for model checking safety properties of hardware, it has been applied to model checking more elaborate properties expressing temporal constraints (e.g. LTL and CTL properties) and model checking software [16, 17]. Results for more recent developments of software verification techniques based on IC3 suggest that these techniques are competitive with established software verification methods [13].

1.3 Project Aims

The main aim in this project is to implement a basic form of the IC3 algorithm in Haskell that can correctly check several small examples. Additionally, the project is intended to provide an opportunity for me to learn and use Haskell, to understand formal methods and especially model checking more deeply, and to put into practice software engineering and design techniques.

The project aims have been achieved through the completion of the following:

- I gained background knowledge about Haskell, the IC3 algorithm, and software development tools widely used in industry, such as Git (Chapter 2).
- I implemented the components of a model checker, including several variants of the IC3 algorithm (Chapter 3).
- I empirically evaluated the model-checking capabilities of the implementations, and attempt to draw some conclusions from my empirical data (Chapter 4).

I provide a summary of the completed work and ideas for future work in Chapter 5. With reference to the project aims submitted in my project proposal, I have met and exceeded all aims.

Chapter 2

Preparation

This chapter describes the knowledge gained and plans made while preparing to begin writing code for the project. The preparation for the project involved learning purely functional programming language Haskell (Section 2.1), an analysis that distinguishes the main components of the project (Section 2.2), choosing and learning how to use the necessary tools for implementing the components of the project (Sections 2.3, 2.5, 2.6), and gaining the necessary knowledge about the symbolic representation of hardware models (Section 2.4) and the IC3 algorithm (Section 2.7) to adequately complete my project. Throughout this chapter and subsequent ones, I collective refer to the model checker variants implemented as part of this project as *MC*.

2.1 Haskell

All of the code for the project, with the exception of the C wrapper for the MiniSat interface (described in Section 3.2) is implemented in the purely functional programming language Haskell [6]. I briefly explain some features of the language that will make the rest of the report easier to follow. I assume a working knowledge of Standard ML.

Functions Unlike in Standard ML, there is no keyword in Haskell for defining functions. Instead, Haskell functions are defined as a series of equations. Otherwise, Haskell function definitions are similar to Standard ML function definitions with the `fun` keyword omitted, though curried functions are more common in Haskell than Standard ML. Anonymous functions in Haskell are also defined similarly as they are in Standard ML, using `\` to bind variables analagously to Standard ML's `fn` keyword. Similarly to Standard ML, Haskell performs pattern matching for function definitions and `case` expressions. The pattern languages Standard ML and Haskell are identical.

Types Haskell is a strongly- and statically-typed language. Haskell type constructors and datatype declarations are largely similar to those in Standard ML. A new Haskell datatype is declared using the `data` keyword, an analogue of Standard ML's `datatype` keyword. Haskell also provides record syntax for creating new record types, allowing components of a product type to be named.

Though Haskell compilers perform type inference, type signatures can be (and, in cases where type inference cannot resolve ambiguities, must be) provided. For example, the type signature for the `prove` function in `IC3.hs`, a curried function that takes a `Model` and returns a function that takes a `Lit` and returns a `Bool`, is `prove :: Model -> Lit -> Bool`. Type signatures may begin with constraints specifying that a polymorphic type variable occuring in the type signature must be an instance of a certain type class.

Haskell's type classes facilitate ad hoc polymorphism [26]. Functions specified within the definition of a type class must be supported for any type that is an instance of that type class. Haskell compilers are able to automatically provide instances of standard type classes for some types. For example, a datatype that is an instance of a type class is the `Lit` datatype in the `AigModel` module in `Parser/AigModel.hs`:

```
data Lit = Var Word | Neg Word | Boolean Bool deriving (Show, Eq, Ord)
```

The `Lit` datatype is an instance of the `Show`, `Eq`, and `Ord` type classes, with the instances being automatically derived. The `Show` class contains the `show` function (and related functions) that converts instances to `String`, the `Eq` class contains operators that allow equality testing of instances for equality, and `Ord` class contains comparison operators that allow instances to be ordered.

Haskell's `Monad` class encompasses composable structures that describe computations. These computations may have side effects, and Haskell programs use instances of the `Monad` class to achieve side effects such as I/O, which is achieved using the `IO` monad.

Monadic values can be created with the function `return :: Monad m => a -> m a`, which takes a Haskell value of some type `a` and returns an instance `m` of the `Monad` class containing that value. The `>>=` infix operator ('bind') allows computations to be composed. As expected given its type signature `(>>=) :: Monad m => m a -> (a -> m b) -> m b`, the function takes a Monadic value containing a computation that produces a value of type `a` and a function that takes values of type `a` and returns monadic values, and returns the result of applying the function to the value. The `>>` infix operator, which has type signature `(>>) :: Monad m => m a -> m b -> m b`, also allows computations to be composed, but the output of the first action is ignored by the second.

To avoid unwieldy code resulting from composing several computations with `>>=`, Haskell provides syntactic sugar in the form of `do`-notation, which allows monadic computations to be written in an imperative fashion.

For example, the `addClause'` function in `Minisat/Minisat.hs` could be written as follows:

```
addClause' solver clause =
  newMinisatVecLit >>=
  \veclit ->
    addToVecLit veclit clause >>
    addMinisatClause solver veclit >>
    deleteMinisatVecLit veclit >>
    return solver
```

Using do-notation results in more readable code:

```
addClause' solver clause =
  do veclit <- newMinisatVecLit
     addToVecLit veclit clause
     addMinisatClause solver veclit
     deleteMinisatVecLit veclit
     return solver
```

Values inside the IO instances of the Monad class can be extracted using unsafePerformIO at the expense of guaranteed type safety; for pure computations, the use of unsafePerformIO does not compromise the type safety of the program.

Lists and Tuples List types and values in Haskell are denoted using square brackets, e.g., a list of Lits has type [Lit]. Lists can be appended using the ++ infix operator, and elements can be prepended to lists using the : infix operator. For example, 1:[2,3,4] ++ [5,6] gives the Haskell equivalent to 1::[2,3,4] @ [5,6] in Standard ML, both of which result in the value [1,2,3,4,5,6].

Tuples in Haskell are syntactically the same as in Standard ML, but the Standard ML product type a * b * c has Haskell type (a, b, c) as its analogue.

Modules A Haskell program consists of modules, which organize code. Modules (or just selected functions from modules) can be imported into other modules, which is how library functions can be used. Modules can be referred to (e.g. in import statements) by names that incorporate where they are stored in the directory structure. For example, the Model module in file Model/Model.hs can be referred to as Model.Model.

2.2 Requirements Analysis

The model checker *MC* requires a way of taking input models and, since the IC3 algorithm uses a SAT solver as an internal subcomponent, 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 *MC* to be run on examples from the Hardware Model Checking Competition (HWMCC), since this is the format used to specify examples in the competitions. I chose the MiniSat SAT solver to allow for better comparison of *MC* with Aaron Bradley's reference implementation of IC3 (*IC3ref*)[14]. Because MiniSat is the solver used by *IC3ref*, using it as the solver for *MC* removes the choice of SAT solver as a variable to consider when comparing performance, reducing confounding factors.

Given that the model checking algorithm deals with transition systems (discussed in Section 2.4), 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 [4] was used for managing the project's code, and GitHub [5], a widely-used hosting service for Git repositories, was used to keep backups of the code. The previous versions maintained by the system proved useful in the development of the code, and branching and merging capabilities were useful for organizing different variations of the model checker. I used Git submodules, which allow the inclusion of other Git projects within another project, 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).

Haddock The Haddock [30] 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 the Haskell package database Hackage.

HUnit HUnit [28] is a framework for writing unit tests in Haskell based on the JUnit framework [10] for unit testing in Java. HUnit tests can 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 gives a `TestCase` that tests that function `isEven` returns `True` when called with parameter 12:

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

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

Criterion Criterion [3] 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 HTML. The `.tpl` file can be configured such that Criterion can, e.g., output benchmark sample results to a CSV file, as the `.tpl` for benchmarking this project was configured.

Cabal Cabal [2] is the standard package and dependency management system for Haskell, where a package may be a library or a complete piece of executable software. A `.cabal` file in the root directory of a project specifies information about the Cabal package, such as its version and dependencies, The `.cabal` file may contain several 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 for MiniSat and the MiniSat wrapper 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 [29] 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 Haskell source (`.hs`) file that can then be compiled with a Haskell compiler and run.

HLint The HLint tool [32] is a linting tool that suggests improvements to Haskell source code to improve the style of the code. The incorporation of HLint suggestions in this project resulted in simpler, more readable code.

AIGER Utilities Several tools provided in AIGER Utilities [1] were used in this project. The AIGER parser provided was used for comparison with and as an alternative to the parser developed as part of this project.

The Aiger Utilities' tools to convert between formats for specifying hardware models eased the specification of new models that would be compatible with the model checker implementations, which accept only AIGER-formatted inputs. In particular, I used the `bliftoaig` tool to convert circuits specified using the Berkeley Logic Interchange Format to circuits specified using the binary AIGER format, and the `aigtoaig` tool, to convert between the ASCII and binary AIGER formats.

MiniSat MiniSat [23, 24] 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. I assume basic knowledge of propositional logic throughout.

Logic A variable is a propositional symbol that can be assigned Boolean values *True* or *False*. A *literal* is defined as being either an atom a (which can be a variable or Boolean value) or its negation $\neg a$.

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 cube 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 de Morgan'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 de Morgan'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. Any propositional formula can be easily converted to an equivalent CNF form.

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:

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

In a similar fashion, for a formula X involving only current-state variables, the formula X' is the formula X where each current-state variable v has been replaced by the corresponding next-state variable 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. Relative inductiveness plays a key role in the IC3 algorithm.

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 $\{a \in \mathit{Full\ Assignment} | a \wedge b \text{ is satisfiable}\}$.

For a logical formula B, a B *state* is a state that is in the set of states represented by B. 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 $1 \leq i < k$, where T represents the transition relation of the transition system.

2.5 Model Specification

I used both the AIGER format and Berkeley Logic Interchange Format (BLIF) to specify fourteen example hardware models. The models that the model checker accepts as input are specified using the AIGER format; however because using the AIGER format to specify larger models was cumbersome, I specified some models using BLIF and converted them to AIGER format using the Aiger Utilities' `bliftoaig` tool. Given that one of the project's components is an AIGER parser, I describe the AIGER format in more depth.

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, either of which can be used as inputs to this project's model checker. 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 format. Another example is that AND gates' inputs can occur in any order in the ASCII format, but the binary format encodes AND gates under the assumption that inputs' indices are in ascending order. The binary version's assumptions on component ordering allow the format to be more compact. The HWMCC examples use the binary format.

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

I provide a description of how variables are represented in all AIGER formats followed by a description of the old ASCII version of the AIGER format, which is sufficient to understand the handwritten examples written directly in AIGER format. Descriptions of the other formats can be found in Appendix A.

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 literal $\neg x$ 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.

Any index above 1 represents a literal. For all such indices, even indices represent positive literals, and odd indices represent negative literals. The representation thus allows the least significant bit of an index to give the sign of a literal and a single bitwise right shift to find the variable name for the literal.

Old ASCII 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 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

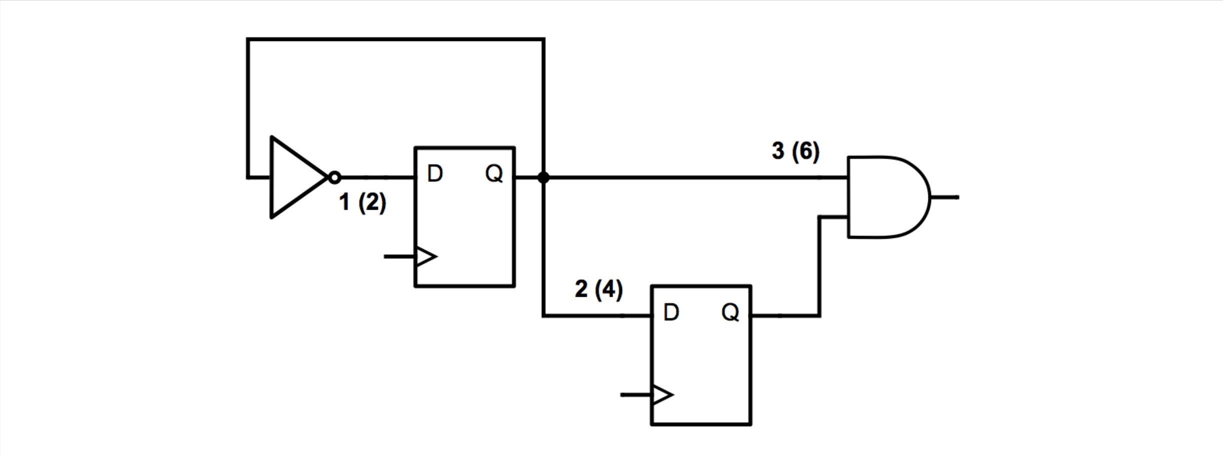

Figure 2.1: The circuit represented in `examples/simple3.aag`.

the positive literal for their variable name and the two indices that specify their input values.

For example, the circuit in Figure 2.1 is represented as follows:

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

The circuit has no inputs, two latches with indices 2 and 4 and one AND gate with index 6 that takes the outputs of the two latches as inputs. The output of the whole circuit is the output of the AND gate.

2.6 MiniSat

MiniSat is the SAT solver used by the *MC* implementations. The creation of a working interface to MiniSat required some background knowledge of how MiniSat works. The MiniSat code explained in this section is in C++.

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.

MC uses instances of `SimpSolver`, a subclass of the `Solver` class that does simplification and returns full assignments, providing more useful results for the queries that IC3 makes.

2.7 The IC3 Algorithm

I now describe the basic IC3 algorithm and some of its extensions, which are necessary to understand the following chapters. 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 reachable $\neg P$ state. The pseudocode in Figure 2.2 gives an overview of the basic IC3 algorithm, which I refer to in the following 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 reachable by the transition system in at most i steps from the initial state (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 the *frontier*.

The *initiation query* $I \Rightarrow P$ on line 2 checks that the property holds in the initial state I. This query is run once at the start of the algorithm for the desired property. If it fails (i.e. if it is *False*), then the algorithm terminates, as a state in which $\neg P$ holds is reachable in 0 steps. If it 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

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

then 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* algorithm 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 in Figure 2.3 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.

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 CTIs, 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.

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

Figure 2.3: Pseudocode for proving negated CTIs.

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). Because the number of clauses in each frame F_i decreases monotonically as i increases and the frame F_0 can only have finitely many states, the algorithm always terminates.

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.

```
Function mic(cls,i):
    foreach literal l in cls do
        subcls := cls \ {l}
        if down(subcls, i) then
            cls := subcls
    return cls
Function down(cls, i):
    if ¬(I ⇒ cls) then return False
    if F_i ∧ cls ∧ T ⇒ cls' then return True
    p := F_i ∧ t state such that F_i ∧ t ∧ p ⇒ ¬t'
    cls := cls ∩ p
    return down(cls,i)
```

Figure 2.4: The algorithm for finding the minimal inductive subclause. Clauses are assumed to be passed by reference.

The checking phase (described by *down* in Figure 3.3, which takes a clause $cls = \neg s$ and a depth i as arguments) performs the normal queries for determining whether the subclause is inductive relative to F_i: 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 $u \Rightarrow t$.

The formula $F_i \wedge t \wedge T \Rightarrow t$ not holding 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 $t \cap \neg p$.

An improvement to the generalization provided by finding minimal inductive subclauses in this way incorporates the use of counterexamples to generalization [27].

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.

The reason the query $F_k \wedge T \wedge c \Rightarrow c'$ is unsatisfiable might be that 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

This chapter describes my implementation of the *MC* model checkers. The implementation can be broken up into four main components: the AIGER parser (Section 3.1), the MiniSat interface (Section 3.2), the hardware model representation (Section 3.3), and the model checker (Section 3.4). The implementation of the AIGER parser, MiniSat interface, and *Basic* version of the IC3 algorithm satisfy the project aims to implement these components. As extensions, I have implemented additional variants of the IC3 algorithm.

The variants of the model checker component differ in overall structure, the finding of CTIs, the way that propagation is performed, and the way that CTIs are inductively generalized. The variants and the differences among them are given in the following table:

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

I now briefly explain the motivations for the deviations from the *Basic* implementation. The details of these deviations are given in Section 3.4. The “Priority Queue” alteration was based on observations that using priority queues to keep track of proof obligations is more efficient than the simple recursive implementation of the IC3 algorithm [22, 25]. The “Smaller CTIs” alteration was based on observations that it is better to find predecessors to counterexamples that describe sets of several states rather than singleton states [25]. The “Subsumed Clauses” alteration was based on the observation that subsumed clauses slow down the SAT-solver and should be eliminated to achieve better performance [22]. The “Generalization with CTGs” alteration was based on the description of an algorithm that improves upon the basic inductive generalization algorithm [27].

In this chapter, I provide an explanation of the implementation of each of the four main components in turn.

3.1 Parser

The relevant files for this section can be found in the `Parser` directory. The parser component parses ASCII or binary-formatted AIGER files and assumes that the new format is used (because the new format is backward compatible) . Justice properties and fairness constraints are not handled by *MC*, so the parser ignores them.

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

3.1.1 Model

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

Literals are represented by `Lits` (defined in Section 2.1), which store decoded versions of AIGER indices. The `Lit` datatype in `Parser.AigModel` has the following constructors:

- `Boolean`, which takes a `Bool` argument;
- `Var`, which takes a `Word` argument; and
- `Neg`, which takes a `Word` argument.

`Booleans` represent the Boolean values corresponding with AIGER indices 0 and 1, `Var` represents the positive literal of the variable whose name is given by the `Word` it takes as an argument, and `Neg` represents the negative literal of the variable whose name is given by the `Word` it takes as an argument. Variable names are adjusted (by subtracting 1) so that they start at 0. For example, the index 3 read from an AIGER file is parsed to `Neg 0`; the odd index 3 indicates that it is a negative literal of the variable 1, and subtracting by 1 gives the new variable name 0.

Latches and AND gates are represented using three-element `[Lit]`s. 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 example, the latch from Figure 2.1 represented by `2 3` is parsed to `[Var 0, Neg 0, Boolean False]`. 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.

For example, the AND gate from Figure 2.1 represented by `6 2 4` is parsed to `[Var 2, Var 0, Var 1]`.

The full AIGER representation for the circuit in Figure 2.1 is as follows:

```
Model { numVars = 3
      , numInputs = 0
      , latches = [ [Var 0, Neg 0, Boolean False]
                  , [Var 1, Var 0, Boolean False] ]
      , outputs = [Var 2]
      , ands = [ [Var 2,Var 0,Var 1] ]
      , bad = []
      , constraints = []}
```

3.2 MiniSat Interface

The SAT solver for *MC* is MiniSat. Because the Haskell Foreign Function Interface (FFI) 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.

In the C wrapper, 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. For instance, the following function is in `Minisat/CSolver.cpp` is a wrapper for the `addClause` in the `Solver` class:

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

The `result` struct to `Minisat/CSolver.h` allows a single function call to return all the results of a SAT query. The struct contains an indication of query satisfiability and pointers to the model and conflict vector (if any) of the `Solver`:

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

The function `solveWithAssumps`, a wrapper for the version of `solve()` that takes an assumption vector as an argument, returns a pointer to a `result` struct rather than just whether or not the query was satisfiable. The Haskell interface uses the Haskell FFI and 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. All calls to C functions must occur inside of `IO` monads, but having the interface functions return `IO` monads means that any functions that use the interface functions to get values from MiniSat would need to perform all their computations inside monads as well, requiring them to be written imperatively. 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 *MC* representation of literals was necessary.

I chose to represent a variable with name x taken from the nonnegative integers with the MiniSat variable $2 \times x$ and its corresponding next-state variable x' with MiniSat variable $(2 \times x) + 1$. The model returned in the `result` struct points to an array of three-valued boolean (the normal boolean values including an *undefined* value) `lbool` values indexed by MiniSat variable indices. To extract models, the array values are accessed, and for each *True* or *False* value the corresponding positive or negative literal for that index is added to the `Lit` list representing the model.

3.3 Hardware Models

3.3.1 Representation

Literals and Clauses

The `Lit` data structure in `Model.Model` represents 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 `Lits` 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:

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

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` list of clauses that specify latches and clauses that specify AND gates capture the transition relation T. The literal that gives the safety property is given by `safe`.

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 a `Parser.AigModel.Model` and outputs a `Model.Model.Model`. As mentioned before, the `Model.Model.Lit` 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` because Boolean values are not used in these representations.

Latches

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

Consider a given `Parser.AigModel.Latch [l, n, init]` representing the latch with output variable l (represented by `l`), next-state n (represented by `n`) 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.

Generating 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.

Generating next-state clauses proceeds similarly. By the semantics of a latch, the clauses generated for the next state should have a conjunction logically equivalent to $n \Leftrightarrow l'$. 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, if n is not a Boolean value, 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 $n \Leftrightarrow l'$, 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:

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

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 `[out, in1, in2]`, representing the AND gate with output variable a (represented by `out`), 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. By the semantics of AND gates, the current-state clauses generated should have a conjunction logically equivalent to $a \Leftrightarrow i_1 \wedge i_2$ and the next-state clauses generated should have a conjunction logically equivalent to $a' \Leftrightarrow i_1' \wedge i_2'$. To achieve the necessary logical equivalence for the next-state clauses, the generated current-state clauses are primed.

If both i_1 and i_2 are Booleans (corresponding to both `in1` and `in2` using the `Boolean` constructor for `Parser.AigModel.Lits`), 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, 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 in the explanation for generating clauses for latches.

If neither of i_1 or i_2 are Booleans, 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 $a \Leftrightarrow i_1 \wedge i_2$:

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

It follows from distribution and application of de Morgan's laws that the original double implication is equivalent to the CNF formula corresponding 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 variants of the IC3 algorithm: the most basic variant (*Basic*), a variant that improves upon *Basic* by discovering smaller CTIs (*BetterCTI*), and a variant that improves upon *BetterCTI* by considering subsumed clauses (*BetterPropagation*).

1 **Function** $prove(M, P)$**:**

2   **if** $\neg(I \Rightarrow P)$ **then return** *False*

3   **return** $prove'(M, P, I, \text{nil})$

4 **Function** $prove'(M, P, F_k, [F_0, \ldots, F_{k-1}])$**:**

5   **if** $F_k \wedge T \Rightarrow P'$ **then**

6   **return** $pushFrame(F_k, \emptyset, M, P, [F_0, \ldots, F_{k-1}])$

7   **else**

8   **let** $cti = nextCTI(F_k \wedge T \Rightarrow P')$,

9         $(result, [G_0, \ldots, G_{k-1}], G_k) = proveNegCTI((i, x, I, T), \text{cti}, k-1)$ **in**

10   **if** *result* **then**

11   **let** $(fixed, [H_0, \ldots, H_{k-1}, H_k]) = propagate([G_0, \ldots, G_{k-1}, G_k])$ **in**

12   **if** *fixed* **then return** *True*

13   **else return** $prove'(M, P, H_k, [H_0, \ldots, H_{k-1}])$

14   **else return** *False*

15 **Function** $pushFrame(F_{k-1}, F_k, M, [F_0, \ldots, F_{k-2}])$**:**

16   **let** $(fixed, G_k) = push(F_{k-1}, F_k)$ **in**

17   **if** *fixed* **then return** *True*

18   **else return** $prove'(M, P, G_k, [F_0, \ldots, F_{k-2}, F_{k-1}])$

Figure 3.1: General structure of the algorithm implementation in Haskell. The transition relation T is acquired from the model M

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

I describe the overall structure shared by all the variants except the *PriorityQueue* implementation, and then describe the implementation details of smaller components of the algorithm and how they differ across variants. A separate description of the *PriorityQueue* implementation follows.

3.4.1 Overall structure

The general structure of the algorithm in the implementations is similar to the structure given in Figure 2.2; however, there are 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 3.1 that outlines the general structure shared by all the implementations of the model checker except *PriorityQueue* and compare this structure with Figure 2.2 (see Section 3.4.8 for *PriorityQueue*).

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 (line 1) makes an initiation query (line 2), and, if it succeeds, calls *prove'* (line 3), which corresponds to a recursive version of the main while loop in line 5 of Figure 2.2. Here, *proveNegCTI* (line 9) does not correspond just to the *proveNegCTI* algorithm in Figure 2.2 but rather to the while loop that contains that function.

Because functions in Haskell are pure, the assumption made in Figure 2.2 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 Figure 3.1 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 (line 15) instead where appropriate.

In Figure 2.2, each iteration of the algorithm calls the propagation phase (line 14 in Figure 2.2); however, 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 unnecessary work. The modified algorithm is such that that when the consecution query succeeds, it calls the *pushFrame* function (line 6) 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 2.2, calls the *propagate* function (line 11) to handle the updates to the frames made by *proveNegCTI*.

The *propagate* function carries out the propagation phase of the algorithm. 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.

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 implementation of the query $I \Rightarrow P$ for a frame I and a clause P makes use of the fact that $I \Rightarrow P$ is holds iff $\neg P \wedge I$ is unsatisfiable. The resulting implementation in `IC3.hs` is the following:

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

3.4.4 Consecution

Similarly to the initiation query, the consecution query must be expressed in CNF. All variants make use of the fact that $F_k \wedge T \Rightarrow P'$ holds iff $\neg P' \wedge F_k \wedge T$ is unsatisfiable to yield the following implementation:

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

3.4.5 Counterexamples to Induction

CTIs 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`.

Basic

In the *Basic* implementation, `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 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 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 3.1 (lines 15, 11) and Figure 3.4 (lines 7, 10) 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 `f`. If it is, then the clause can be added to `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 removing the clauses in `f` that are subsumed by other clauses, which is done by all 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 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.

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 [25], and all implemented versions of generalization (i.e. all the `inductiveGeneralization` function implementations) approximate the MIC with a call to the function `generalize`.

Simple

The simplest approximation for a MIC attempts to drop each literal in turn and checks that the resulting clause c satisfies formulas $I \Rightarrow c$ and $F_k \wedge c \wedge T \Rightarrow c'$ as the original clause did. If the resulting clause satisfies both 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`.

```
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 3.2: The `inductiveGeneralization` Haskell function that approximates the *mic* algorithm.

This corresponds to the algorithm described in Figure 3.3, but where *down* simply checks for the relative inductiveness of the subclause and does not attempt to expand it.

Minimal Inductive Subclauses and Counterexamples to Generalization

The more elaborate version of generalization implements the full (but limited in number of attempts) *mic* algorithm with *down* modified to handle 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 for the aforementioned efficiency reasons; 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

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

Function $down(cls, i)$**:**
  if $\neg(I \Rightarrow cls)$ **then return** *False*
  if $F_i \wedge cls \wedge T \Rightarrow cls'$ **then return** *True*
  ctg:= model extracted from SAT query $F_i \wedge cls \wedge T \Rightarrow cls'$ **if** $I \Rightarrow \neg ctg$ and $F_i \wedge \neg ctg \wedge T \Rightarrow \neg ctg'$ **then**
    $j := 0$
    while $F_j \wedge \neg ctg \wedge T \Rightarrow \neg ctg$ **do** $j := j + 1$
    $generalizedNegCTG := mic(\neg ctg, j)$
    $F_j := F_j \cup \{generalizedNegCTG\}$
    return $down(cls, i)$
  else
    $p := F_i \wedge t$ state such that $F_i \wedge t \wedge p \Rightarrow \neg t'$
    $cls := cls \cap p$
    return *down(cls,i)*

Figure 3.3: The algorithm for the version of *down* that handles CTGs.

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 CTI 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 $k \leq i$. I discuss the way that the implementation achieves this priority ordering later.

Unlike the other variants, 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 to prove whichever property has the highest priority in the queue, i.e. fulfill the proof obligation with the highest priority (this proof obligation is the one returned by $dequeue(queue)$ in line 6 of 3.4).

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

Function *prove*(M, P)**:**
  if $\neg(I \Rightarrow P)$ **then return** *False*
  let *queue* = queue containing proof obligation $(\neg P, 1)$ **in**
    return *fulfillObligations*$(M, [I], queue)$

Function *fulfillObligations*$(M, [F_0, \ldots, F_k], queue])$**:**
  let $((s, i), q) = dequeue(queue)$ **in**
    if $F_{i-1} \wedge T \Rightarrow \neg s'$ **then return** *pushFrame*$(M, [F_0, \ldots, F_k], q, (s, i))$
  else let $cti = nextCTI(F_{i-1} \wedge T \Rightarrow \neg s')$ **in**
    if $I \Rightarrow \neg cti$ **then**
      let $(fixed, [G_0, \ldots, G_k], d) = propagate([F_0 \cup \{\neg cti\}, F_1, \ldots, F_k], \neg cti)$
      in
        if *fixed* **then return** *True*
        return *fulfillObligation*$(M, [G_0, \ldots, G_k], (generalize(\neg cti, d), d))$
    else return *False*

Figure 3.4: General structure of the algorithm implementation in *PriorityQueue*.

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 with 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

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

where $q = enqueue((s, i+1), queue)$, 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. This approach 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 with success. 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 triple type `(Int, Int, Clause)`. The `Obligation` triple `(i,r,c)` consists of the the depth `i`, a rank `r` for deciding the ordering of proof obligations at the same depth within the priority queue, and the clause `c` representing $\neg s$. The function implementing *fulfillObligations* is named `proveObligations`.

The priority queue is represented by a `MinQueue` (the minimal element has the highest priority) of `Obligation`s.

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

This chapter discusses the evaluation of the model checker implementations. I describe the solving capabilities of the *MC* variants (Section 4.1) and how the project aim of being able to solve several solve examples has been met, the output of the model checker (Section 4.2.1), how benchmarks were taken (Section 4.2.2. I then compare the variants' performance with each other (Section 4.3) and with the reference implementation *IC3ref* (Section 4.4).

To evaluate the implementations, I ran all variants on fourteen handwritten examples and over one hundred examples taken from the Hardware Model Checking Competitions spanning four years [7, 8, 9]. I chose examples from HWMCC'10 that had relatively short (under 2 second) solving times for `ic3`'s in the competition. Examples were also chosen from HWMCC'11 that did not overlap with HWMCC'10 examples. To avoid having as much overlap between examples as those from HWMCC'10 and HWMCC'11, I skipped HWMCC'12 and took examples with relatively short solving times for some of the solvers from HWMCC'13 [9].

If the elapsed time for attempting to solve an example took longer than ten minutes, that attempt was considered to have timed out. The parameterizable number of failed attempts at dropping literals in the `inductiveGeneralization` functions was set to three, and the parameterizable number of CTGs that each generalization attempt in the *CTG* implementation will handle was also set to three, matching the values for these parameters used by *IC3ref*.

4.1 Correctness

The *MC* implementations have passed the HUnit tests for parsing AIGER files, making MiniSat queries, and model checking. The parser tests involve comparing the output of the two parsers and checking that they match, the MiniSat tests involve checking that results from MiniSat queries (including model and conflict vectors), and the model-checking tests involve checking the results for specific functions in `IC3.hs` such as `consecution`.

The variants could solve all the handwritten examples and fifty of the HWMCC examples within ten minutes correctly, and some variants were able to solve additional examples without timing out. All solutions reported solutions within the time limit agree

```
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`.

with those given by *IC3ref*, providing evidence for the correctness of the solutions given by *MC*.

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, 77 of which are AND gates. To provide context for the typical number of variables in the small examples, the following table provies the number of variables in the handwritten examples involving a two-bit counter (`counters2.aig`), a three-bit counter (`counters3.aig`), and a four-bit counter (`counters4.aig`):

Example	Number of Variables
`counters2.aig`	14
`counters3.aig`	45
`counters4.aig`	79

For examples solved without timing out, the largest (in terms of number of variables) unsafe example for which the variants gave a solution was `bj08goodbakerycyclef7.aig` with 19900 variables, and the largest safe example for which the variants gave a solution was `pdtvsar8multip26.aig` with 7174 variables.

4.2 Empirical Analysis

4.2.1 Output Format

All model checker implementations print the string `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 provides 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 total number of queries made. The *CTG* implementation also reports the number of CTGs found. Sample output for the *CTG* implementation on example `counters3.aig` is given in Figure 4.1.

4.2.2 Benchmarking

I took performance benchmarks both for the *MC* variants and for two configurations of *IC3ref*, where the configurations differ in whether CTG-handling is enabled or not. For each variant, forty benchmarking samples were taken for each example that the variant could solve within the time limit.

The collected data consists of execution time, 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. These measurements can be found in Appendix B.

Out of the 50 HWMCC examples, 47 did not require finding any CTIs; for these examples, the *Basic*, *BetterCTI*, *BetterPropagation*, and *CTG* implementations give similar results.

4.3 Performance Impact of Variations

Profiling 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. In this section, I will discuss the impact that different variants of the model checker have on the size and number of SAT-solver queries and performance.

Smaller Counterexamples to Induction The *BetterCTI* implementation exhibits consistently better performance than the *Basic* implementation for examples that require finding at least one CTI. In such cases, discovering CTI clauses with fewer literals leads, as expected, to a smaller average number of literals per clause (as seen in Figure 4.2), which suggests smaller SAT queries.

As mentioned previously, because the *BetterCTI* implementation uses CTIs that encompass sets of states rather than single states, when a negated CTI is proven at a depth k, several states have been shown to be unreachable within k steps of the transition relation from the initial state. Dealing with sets of states rather than single CTI states should allow the *BetterCTI* implementation to deal with fewer CTIs in some cases (because the CTI sets of states may encompass several CTI states), leading to fewer queries. Benchmark results agree with these expectations; for examples that require finding more than one CTI, *BetterCTI* finds fewer CTIs and makes fewer queries. For example, the *Basic* variant finds 59 CTIs and makes 414 queries to solve `shortp0.aig`, but the *BetterCTI* variant only finds 3 CTIs and makes 49 queries, an order of magnitude difference.

The improvement of finding more general CTIs enabled the *BetterCTI* variant of the implementation (and all other implementations that include finding smaller CTIs) to solve six more examples (`counterp0.aig`, `counterp0neg.aig`, `pdtvishuffman7.aig`, `pdtvismiim3.aig`, `6s318r.aig`, `srg5ptimo.aig`) than the *Basic* version without timing out.

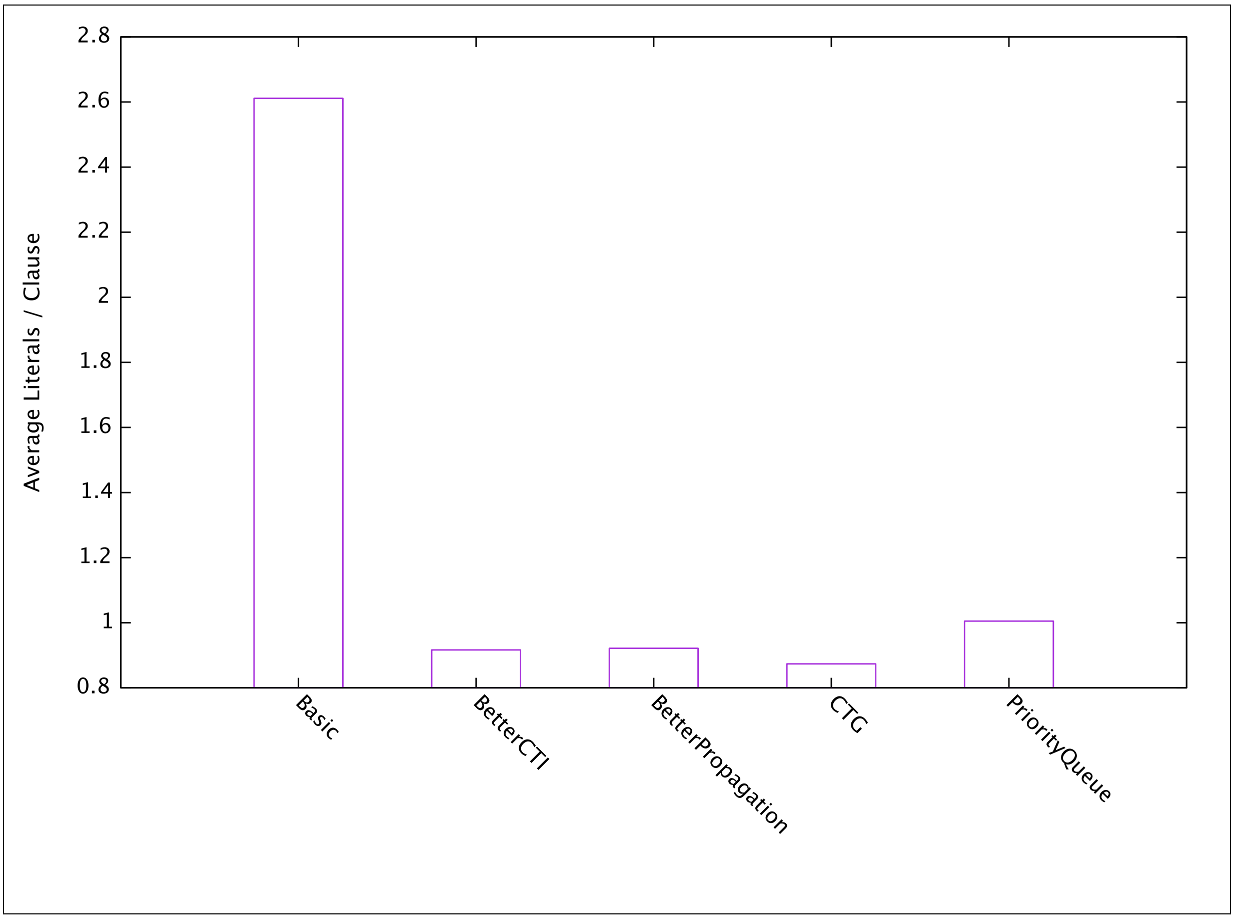

Figure 4.2: Average literals per clause averaged over the fourteen handwritten examples and fifty Hardware Model Checking Competition examples.

Propagation Removing subsumed clauses also results in better performance for several examples. While the performance impact that the improvement has is less drastic than the improvement of *BetterCTI* over *Basic*, the *BetterPropagation* version performs considerably better than the *BetterCTI* version on the `counters3.aig` and `counters4.aig` examples in particular (as seen in Figure 4.3), where the adjustments allow the algorithm to prove the safety properties using fewer queries (as seen in Figure 4.4). Even for examples such as `pdtvismiim3.aig`, where *BetterPropagation* makes more queries than *BetterCTI*, *BetterPropagation* manages to perform better than *BetterCTI* because it makes smaller queries.

Counterexamples to Generalization The *CTG* variation that deals with CTGs performs 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, which require making additional queries on each call to the `inductiveGeneralization` function.

Similar results can be found in the performance of *IC3ref* with basic generalization and improved (CTG-using) generalization on the same examples: for these small examples, the reference implementation performs better overall with CTG-handling disabled.

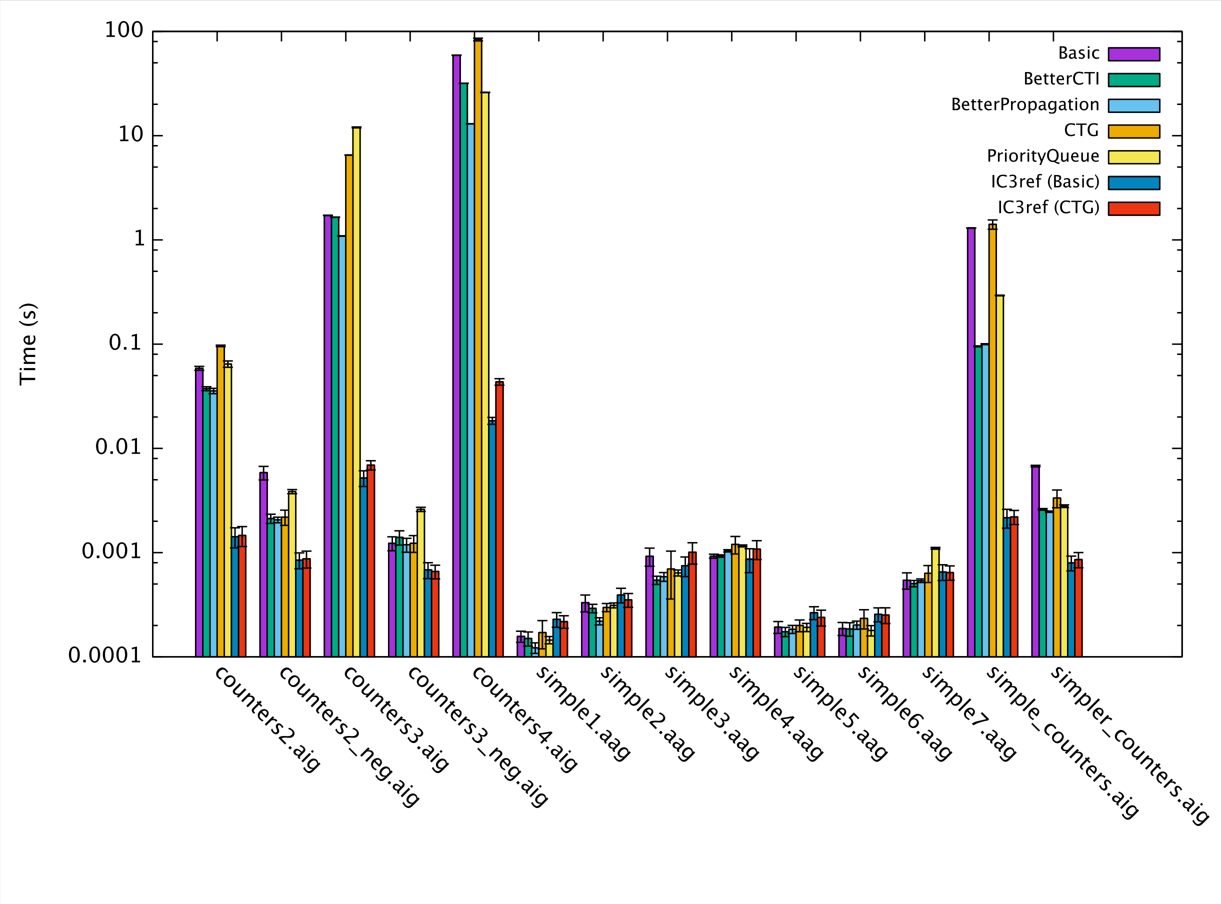

Figure 4.3: Benchmark results for the fourteen handwritten examples on a log scale.

Priority Queues The *PriorityQueue* implementation generally does not perform as well as the implementations that do not use priority queues (with the exception of *CTG*). This result disagrees with findings that implementations of the IC3 algorithm that use priority queues are more efficient than simple recursive implementations [22, 25]. There are two features of the variant in particular that may account for its worse performance: accumulating CTIs and early generalization.

One of the performance advantages of the *PriorityQueue* implementation is that CTIs do not need to be rediscovered [22]: 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.

The *PriorityQueue* implementation performs inductive generalization for each CTI only once, though, when that CTI's first proof obligation is first enqueued. Rediscovering CTIs would allow the CTIs to be generalized relative to later frames as well, rather than only to the first frame relative to which the negated CTI is inductive. Not generalizing CTIs relative to later frames may lead to *PriorityQueue* making larger queries and explain the variant's higher average number of literals per clause.

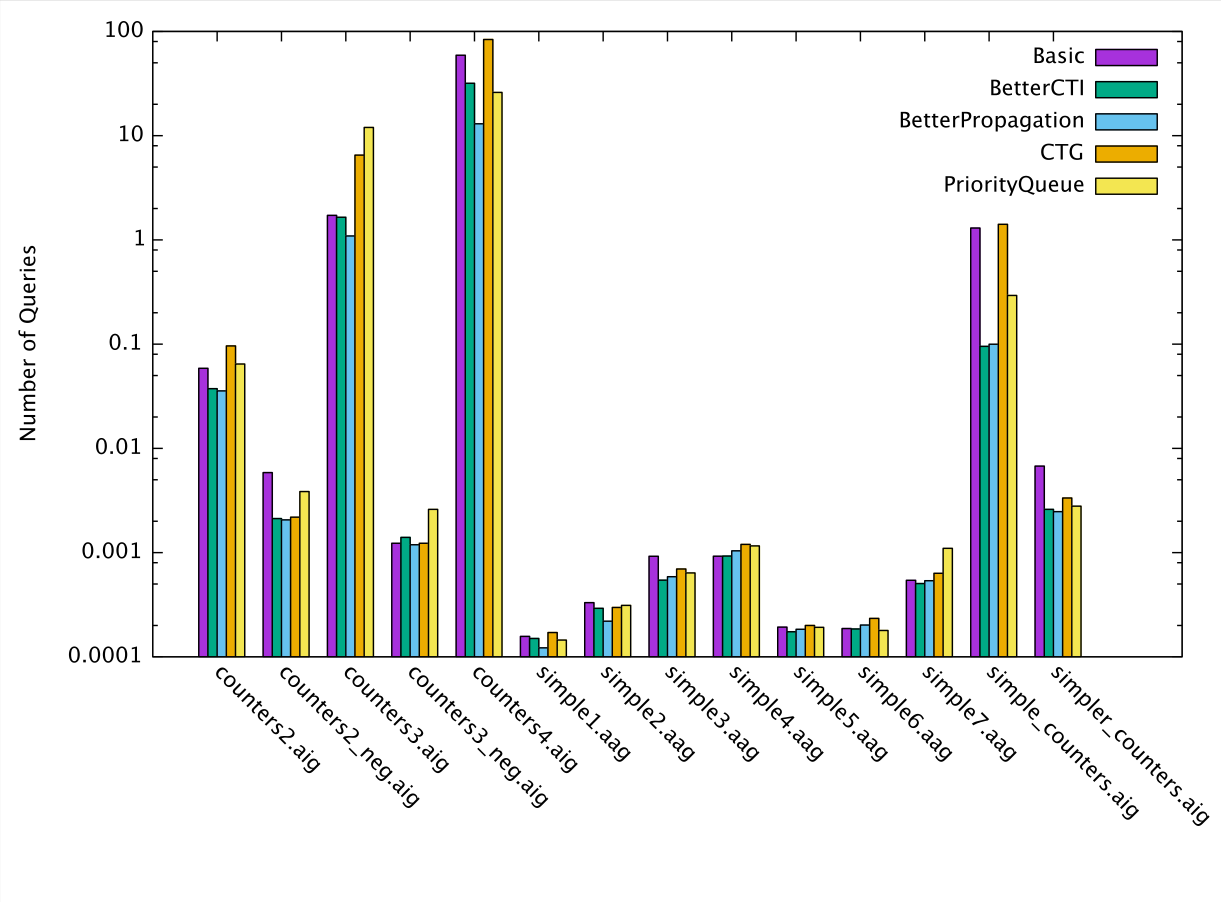

Figure 4.4: Number of queries for each variant run on the fourteen handwritten examples on a log scale.

4.4 Reference Implementation

The performance of this project's model checker implementations is, for all except very small examples (e.g. `simple1.aag`), worse compared to the average performance of *IC3ref* (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 `Models` 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 implementations instead use 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 `SimpSolvers` directly. In contrast, the Haskell implementations 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 [23].

Overall Structure

The reference implementation uses a priority queue, but handles proof obligations differently than the *PriorityQueue* implementation does because *IC3ref* does not attempt to reduce the number of generalization attmpts or prevent the rediscovery of CTIs.

For a CTI s that prevents the fulfillment of a proof obligation at depth i, the reference implementation enqueues proof obligation $(s, i-1)$ 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 *IC3ref* as the Haskell implementations, probably as a result of aforementioned differences. 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, so CTIs need to be rediscovered. The efficiency of *IC3ref* otherwise compensates for the performance disadvantage of rediscovering CTIs. 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

This chapter summarizes the work done and goals met for this project. Following the summary in Section 5.1, I give suggestions for further extensions to the project in Section 5.2.

5.1 Summary

The IC3 algorithm provides a new way to perform SAT-based symbolic model checking of safety properties of hardware, and the performance of its initial implementation `ic3` in HWMCC'10 resulted in the development of several variants of and extensions to the algorithm.

This project aims to implement a basic version of the IC3 algorithm in Haskell with the necessary parser and SAT-solver interface, with the goal of having the model checker be able to check small example hardware models. These goals have been achieved.

I described the implementation of the different components required by the project in Chapter 3. In addition to the basic version of the IC3 algorithm, I implemented several variants of *MC* (also described in Chapter 3). The implementations of the model checker can model check not only the fourteen handwritten examples but also models from the Hardware Model Checking Competitions with thousands of variables.

In chapter 4, I evaluated the different model checker variants by comparing their performance on examples from the Hardware Model Checking Competition. The *Basic* implementation correctly solved fourteen small handwritten examples, validated against the reference implementation of IC3, meeting the project goal of being able to check small examples. The benchmark results show that finding smaller CTI clauses results in a dramatic improvement in performance, with the best-performing implementation *betterPropagation* finding smaller CTI clauses and removing subsumed clauses from frames, though none of the variants performs as well as *IC3ref*.

5.2 Further Extensions

Instead of the extensions mentioned in the initial project proposal, I elected to implement other variants of the model checker and examine their effects on the model checker's performance. As a result, interfacing with different SAT solvers and implementing lazy abstraction-refinement remain as future work.

Implementing interfaces with different SAT solvers may allow more examples to be solved efficiently. Aaron Bradley notes that the performance of the IC3 algorithm is considerably affected by the behavior of the SAT solver it uses [16]; even if each SAT query takes the same amount of time, the algorithm's performance may still vary if the SAT solver behaves even slightly differently. Because the choice of SAT solver may affect which examples can be solved efficiently, allowing the model checker to use different SAT solvers may increase the number of examples that can be solved.

Abstraction-refinement is a technique used in verification to mitigate the effects of the state explosion problem. Abstraction removes irrelevant details of the model, and if the abstraction is found to be too coarse at some point during verification, refinement can add necessary details of the model back into the abstraction. Yakir Vizel, Orna Grumberg, and Sharon Shoham introduced a abstraction-refinement scheme that is compatible with the IC3 algorithm [35]. The implementation of this modified algorithm achieved significant speedups compared with the original IC3 algorithm implementation `ic3`, and a variant of the model checker in Haskell that uses this abstraction-refinement scheme may exhibit a similar improvement.

Bibliography

[1] AIGER. http://fmv.jku.at/aiger/.

[2] Cabal. http://www.haskell.org/cabal/.

[3] Criterion. http://www.serpentine.com/criterion.

[4] Git. https://git-scm.com.

[5] GitHub. https://github.com.

[6] Haskell. https://www.haskell.org.

[7] HWMCC'10. http://fmv.jku.at/hwmcc10/.

[8] HWMCC'11. http://fmv.jku.at/hwmcc11/.

[9] HWMCC'13. http://fmv.jku.at/hwmcc13/.

[10] Junit. http://junit.org.

[11] 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.

[12] 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.

[13] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (ctigar). In *Computer Aided Verification (CAV)*, pages 831–848. Springer International Publishing, 2014.

[14] Aaron Bradley. IC3ref. https://github.com/arbrad/IC3ref.

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

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

[17] Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *Computer Aided Verification (CAV)*, pages 277–293, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[18] Edmund M Clarke, Manpreet Khaira, and Xudong Zhao. Word level model checking-avoiding the Pentium FDIV error. In *Design Automation Conference (DAC)*, pages 645–648. IEEE, 1996.

[19] 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.

[20] 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.

[21] Mark Dowson. The ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84–, March 1997.

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

[23] Niklas Eén and Niklas Sörensson. MiniSat. `http://minisat.se/MiniSat.html`.

[24] Niklas Eén and Niklas Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *Theory and Applications of Satisfiability Testing (SAT)*, 2005.

[25] Alberto Griggio and Marco Roveri. Comparing different variants of the IC3 algorithm for hardware model checking. In *Design and Implementation of Formal Tools and Systems (DIFTS)*, 2014.

[26] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *Programming Languages and Systems (ESOP)*, pages 241–256, 1994.

[27] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. Better generalization in IC3. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 157–164, 2013.

[28] Dean Herington. HUnit. `http://hackage.haskell.org/package/HUnit`.

[29] Marcin Kowalczyk. hsc2hs. `http://hackage.haskell.org/package/hsc2hs`.

[30] Simon Marlow. Haddock. `http://www.haskell.org/haddock/`.

[31] 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.

[32] Neil Mitchell. HLint. `http://hackage.haskell.org/package/hlint`.

[33] Vaughan Pratt. Anatomy of the Pentium bug. In *Theory and Practice of Software Development (TAPSOFT)*, pages 97–107, 1995.

[34] 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.

[35] 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

AIGER Format

This appendix describes the binary version of the old AIGER format by comparison with the ASCII version and the new version of the AIGER format by comparison with the old format.

Binary version The binary version of the format assumes that variable indices occur in increasing order. Since each literal must be defined, this assumption allows for variable indices to be omitted 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 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 $\texttt{rhs0} \geq \texttt{rhs1}$, define

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

and

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

The values δ_i are 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 negated safety properties 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 occur 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 be truncated after giving the AND-gate count if all remaining counts are zero, allowing parsers for the new AIGER format to be backwards-compatible.

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

Appendix B

Benchmark Results

In the following tables, "Average Literals per Clause" is abbreviated as "ALC," and "Standard Deviation" is abbreviated as "SD."

B.1 Basic

Name	Frames	ALC	CTIs	Queries	Mean Time (s)	SD of Time(s)
counters2.aig	5	5.72262	11	211	0.05861	0.00259
counters2_neg.aig	2	2.00000	1	31	0.00585	0.00086
counters3_neg.aig	1	1.00000	0	2	0.00123	0.00019
simple1.aag	0	0.00000	0	1	0.00016	0.00002
simple2.aag	1	1.00000	0	3	0.00033	0.00006
simple3.aag	3	1.00000	0	10	0.00092	0.00018
simple4.aag	2	1.00000	0	7	0.00092	0.00004
simple5.aag	0	0.00000	0	1	0.00019	0.00003
simple6.aag	0	0.00000	0	1	0.00019	0.00003
simple7.aag	1	1.00000	0	2	0.00054	0.00010
simpler_counters.aig	2	2.33333	2	44	0.00676	0.00012
bj08aut1.aig	1	1.00000	0	6	0.00866	0.00035
bj08aut5.aig	1	1.00000	0	6	0.02659	0.00040
bj08goodbakerycyclef7.aig	1	1.00000	0	2	0.56561	0.00302
neclaftp5001.aig	5	1.00000	0	98	0.34280	0.00143
neclaftp5002.aig	5	1.00000	0	98	0.34098	0.00198
pdtvisblackjack0.aig	1	1.00000	0	107	4.93599	0.02093
pdtvisblackjack1.aig	1	1.00000	0	107	4.94018	0.01704
pdtvisblackjack2.aig	1	1.00000	0	107	4.94647	0.01987
pdtvisblackjack3.aig	1	1.00000	0	107	4.97260	0.04372
pdtvisblackjack4.aig	1	1.00000	0	107	4.99169	0.01656
pdtvisbpb1.aig	5	1.00000	0	227	4.44190	0.01190
pdtvisgray0.aig	3	1.00000	0	16	0.00482	0.00056

pdtvisgray1.aig	3	1.00000	0	16	0.00397	0.00050
pdtvisheap04.aig	5	1.00000	0	80	1.26738	0.00431
pdtvisheap07.aig	5	1.00000	0	80	1.26758	0.00429
pdtvisheap11.aig	5	1.00000	0	80	1.26100	0.01465
pdtvishuffman2.aig	11	1.00000	0	505	6.97372	0.10439
pdtvishuffman5.aig	0	0.00000	0	1	0.01429	0.00042
pdtvisrethersqo3.aig	0	0.00000	0	1	0.01127	0.00083
pdtvistictactoe00.aig	4	1.00000	0	70	0.82004	0.00389
pdtvistictactoe01.aig	0	0.00000	0	1	0.01251	0.00075
pdtvistictactoe03.aig	0	0.00000	0	1	0.01254	0.00076
pdtvistictactoe04.aig	0	0.00000	0	1	0.01267	0.00087
pdtvistictactoe05.aig	0	0.00000	0	1	0.01254	0.00085
pdtvistictactoe06.aig	0	0.00000	0	1	0.01256	0.00087
pdtvistictactoe07.aig	0	0.00000	0	1	0.01267	0.00093
pdtvistictactoe08.aig	0	0.00000	0	1	0.01269	0.00085
pdtvistictactoe09.aig	0	0.00000	0	1	0.01252	0.00080
pdtvistictactoe11.aig	4	1.00000	0	70	0.84943	0.01721
pdtvistictactoe12.aig	4	1.00000	0	70	0.83579	0.01043
pdtvistwo0.aig	2	1.00000	0	59	0.31650	0.00550
pdtvistwo1.aig	2	1.00000	0	59	0.30160	0.00407
pdtvisvending03.aig	6	1.00000	0	87	1.17120	0.01330
pdtvisvending06.aig	6	1.00000	0	87	1.16194	0.00973
pdtvisvsar02.aig	7	1.00000	0	538	16.55710	0.37338
pdtvisvsar18.aig	7	1.00000	0	538	16.84172	0.12671
shortp0.aig	3	28.96396	59	414	0.76302	0.00649
shortp0neg.aig	2	1.00000	1	20	0.02395	0.00100
srg5ptimoneg.aig	2	1.00000	1	53	0.22922	0.00383
texasifetch1p1.aig	7	1.00000	0	172	1.48502	0.01351
texasifetch1p3.aig	7	1.00000	0	172	1.48269	0.01277
viselevatorp1.aig	5	1.00000	0	81	1.27678	0.01560
6s40p1.aig	0	0.00000	0	1	0.51010	0.00271
6s40p2.aig	0	0.00000	0	1	0.53924	0.04520
bobmiterbm1or.aig	0	0.00000	0	1	0.04803	0.00122
bobsynth00neg.aig	0	0.00000	0	1	0.25376	0.00724
bobtuint06.aig	0	0.00000	0	1	0.03255	0.00073
pdtpmstwo.aig	2	1.00000	0	200	2.37140	0.14902
pdtvsar8multip24.aig	7	1.00000	0	805	94.31110	2.31667
pdtvsar8multip26.aig	7	1.00000	0	805	96.83664	1.36990

B.2 BetterCTI

Name	Frames	ALC	CTIs	Queries	Mean Time (s)	SD of Time (s)
counters2.aig	5	2.14667	9	117	0.03739	0.00163
counters2_neg.aig	2	1.00000	1	11	0.00212	0.00021
counters3.aig	14	4.77747	84	1067	1.64834	0.00713
counters3_neg.aig	1	1.00000	0	2	0.00140	0.00022
counters4.aig	20	7.41489	740	7238	31.82758	0.09015
simple1.aag	0	0.00000	0	1	0.00015	0.00002
simple2.aag	1	1.00000	0	3	0.00029	0.00003
simple3.aag	3	1.00000	0	10	0.00054	0.00005
simple4.aag	2	1.00000	0	7	0.00093	0.00003
simple5.aag	0	0.00000	0	1	0.00017	0.00002
simple6.aag	0	0.00000	0	1	0.00018	0.00003
simple7.aag	1	1.00000	0	2	0.00050	0.00004
simple_counters.aig	3	1.53333	4	59	0.09522	0.00108
simpler_counters.aig	2	1.00000	1	17	0.00260	0.00005
bj08aut1.aig	1	1.00000	0	6	0.00879	0.00018
bj08aut5.aig	1	1.00000	0	6	0.02692	0.00053
bj08goodbakerycyclef7.aig	1	1.00000	0	2	0.56773	0.00299
neclaftp5001.aig	5	1.00000	0	98	0.35171	0.00227
neclaftp5002.aig	5	1.00000	0	98	0.34885	0.00172
pdtvisblackjack0.aig	1	1.00000	0	107	5.00528	0.02396
pdtvisblackjack1.aig	1	1.00000	0	107	5.01390	0.01967
pdtvisblackjack2.aig	1	1.00000	0	107	5.01015	0.01743
pdtvisblackjack3.aig	1	1.00000	0	107	5.02559	0.01869
pdtvisblackjack4.aig	1	1.00000	0	107	5.00053	0.01562
pdtvisbpb1.aig	5	1.00000	0	227	4.43402	0.01480
pdtvisgray0.aig	3	1.00000	0	16	0.00493	0.00033
pdtvisgray1.aig	3	1.00000	0	16	0.00398	0.00046
pdtvisheap04.aig	5	1.00000	0	80	1.27575	0.06627
pdtvisheap07.aig	5	1.00000	0	80	1.27072	0.00582
pdtvisheap11.aig	5	1.00000	0	80	1.26885	0.00558
pdtvishuffman2.aig	11	1.00000	0	505	7.05033	0.02206
pdtvishuffman5.aig	0	0.00000	0	1	0.01441	0.00049
pdtvisrethersqo3.aig	0	0.00000	0	1	0.01165	0.00096
pdtvistictactoe00.aig	4	1.00000	0	70	0.83900	0.00810
pdtvistictactoe01.aig	0	0.00000	0	1	0.01269	0.00075
pdtvistictactoe03.aig	0	0.00000	0	1	0.01276	0.00088
pdtvistictactoe04.aig	0	0.00000	0	1	0.01276	0.00094
pdtvistictactoe05.aig	0	0.00000	0	1	0.01273	0.00092
pdtvistictactoe06.aig	0	0.00000	0	1	0.01262	0.00083
pdtvistictactoe07.aig	0	0.00000	0	1	0.01285	0.00097

pdtvistictactoe08.aig	0	0.00000	0	1	0.01269	0.00088
pdtvistictactoe09.aig	0	0.00000	0	1	0.01285	0.00096
pdtvistictactoe11.aig	4	1.00000	0	70	0.85572	0.00833
pdtvistictactoe12.aig	4	1.00000	0	70	0.86233	0.02433
pdtvistwo0.aig	2	1.00000	0	59	0.32304	0.00296
pdtvistwo1.aig	2	1.00000	0	59	0.30760	0.00584
pdtvisvending03.aig	6	1.00000	0	87	1.19141	0.01042
pdtvisvending06.aig	6	1.00000	0	87	1.18719	0.02535
pdtvisvsar02.aig	7	1.00000	0	538	16.59943	0.09582
pdtvisvsar18.aig	7	1.00000	0	538	16.86875	0.06693
shortp0.aig	3	1.77778	3	49	0.11769	0.00225
shortp0neg.aig	2	1.00000	1	21	0.02666	0.00120
srg5ptimoneg.aig	2	1.00000	1	54	0.24075	0.00658
texasifetch1p1.aig	7	1.00000	0	172	1.52927	0.01564
texasifetch1p3.aig	7	1.00000	0	172	1.52518	0.01259
viselevatorp1.aig	5	1.00000	0	81	1.31455	0.01441
6s40p1.aig	0	0.00000	0	1	0.50789	0.00254
6s40p2.aig	0	0.00000	0	1	0.50274	0.00295
bobmiterbm1or.aig	0	0.00000	0	1	0.04754	0.00100
bobsynth00neg.aig	0	0.00000	0	1	0.27934	0.01959
bobtuint06.aig	0	0.00000	0	1	0.04026	0.00657
pdtpmstwo.aig	2	1.00000	0	200	2.32845	0.12187
pdtvsar8multip24.aig	7	1.00000	0	805	98.40888	1.53823
pdtvsar8multip26.aig	7	1.00000	0	805	98.98892	1.46676
6s318r.aig	2	1.00000	1	673	28.77465	1.03627
counterp0.aig	4	5.39063	20	254	0.48499	0.00247
counterp0neg.aig	4	5.60938	22	282	0.44950	0.00226
pdtvishuffman7.aig	5	1.01539	9	365	5.52972	0.05578
pdtvismiim3.aig	12	1.36531	7	677	10.78374	0.04258
srg5ptimo.aig	3	4.76389	23	260	0.90162	0.01239

B.3 BetterPropagation

Name	Frames	ALC	CTIs	Queries	Mean Time (s)	SD of Time (s)
counters2.aig	5	2.22	9	115	0.03565	0.00211
counters2_neg.aig	2	1.00	1	11	0.00206	0.00013
counters3.aig	12	4.95	76	887	1.09227	0.00417
counters3_neg.aig	1	1.00	0	2	0.00119	0.00018
simple1.aag	0	0.00	0	1	0.00012	0.00001
simple2.aag	1	1.00	0	3	0.00022	0.00002

simple3.aag	3	1.00	0	10	0.00059	0.00006
simple4.aag	2	1.00	0	7	0.00104	0.00003
simple5.aag	0	0.00	0	1	0.00018	0.00002
simple6.aag	0	0.00	0	1	0.00020	0.00002
simple7.aag	1	1.00	0	2	0.00054	0.00002
simple_counters.aig	3	1.79	7	79	0.09987	0.00100
simpler_counters.aig	2	1.00	1	17	0.00247	0.00004
bj08aut1.aig	1	1.00	0	6	0.00876	0.00013
bj08aut5.aig	1	1.00	0	6	0.02700	0.00054
bj08goodbakerycyclef7.aig	1	1.00	0	2	0.57022	0.00462
neclaftp5001.aig	5	1.00	0	98	0.34549	0.00165
neclaftp5002.aig	5	1.00	0	98	0.34268	0.00198
pdtvisblackjack0.aig	1	1.00	0	107	4.95965	0.02993
pdtvisblackjack1.aig	1	1.00	0	107	4.96841	0.03178
pdtvisblackjack2.aig	1	1.00	0	107	4.99453	0.03150
pdtvisblackjack3.aig	1	1.00	0	107	5.00022	0.02670
pdtvisblackjack4.aig	1	1.00	0	107	4.95597	0.01815
pdtvisbpb1.aig	5	1.00	0	227	4.42824	0.01753
pdtvisgray0.aig	3	1.00	0	16	0.00498	0.00033
pdtvisgray1.aig	3	1.00	0	16	0.00418	0.00041
pdtvisheap04.aig	5	1.00	0	80	1.28119	0.06245
pdtvisheap07.aig	5	1.00	0	80	1.26690	0.00705
pdtvisheap11.aig	5	1.00	0	80	1.27137	0.00751
pdtvishuffman2.aig	11	1.00	0	505	7.06688	0.03908
pdtvishuffman5.aig	0	0.00	0	1	0.01468	0.00081
pdtvisrethersqo3.aig	0	0.00	0	1	0.01223	0.00086
pdtvistictactoe00.aig	4	1.00	0	70	0.84192	0.01036
pdtvistictactoe01.aig	0	0.00	0	1	0.01325	0.00062
pdtvistictactoe03.aig	0	0.00	0	1	0.01321	0.00065
pdtvistictactoe04.aig	0	0.00	0	1	0.01324	0.00077
pdtvistictactoe05.aig	0	0.00	0	1	0.01320	0.00074
pdtvistictactoe06.aig	0	0.00	0	1	0.01317	0.00062
pdtvistictactoe07.aig	0	0.00	0	1	0.01319	0.00065
pdtvistictactoe08.aig	0	0.00	0	1	0.01318	0.00078
pdtvistictactoe09.aig	0	0.00	0	1	0.01337	0.00088
pdtvistictactoe11.aig	4	1.00	0	70	0.86201	0.01066
pdtvistictactoe12.aig	4	1.00	0	70	0.85858	0.01042
pdtvistwo0.aig	2	1.00	0	59	0.32532	0.00581
pdtvistwo1.aig	2	1.00	0	59	0.31054	0.00505
pdtvisvending03.aig	6	1.00	0	87	1.18586	0.01930
pdtvisvending06.aig	6	1.00	0	87	1.16866	0.01042
pdtvisvsar02.aig	7	1.00	0	538	16.59315	0.09505

pdtvisvsar18.aig	7	1.00	0	538	16.74249	0.08761
shortp0.aig	3	1.78	3	49	0.11530	0.00237
shortp0neg.aig	2	1.00	1	21	0.02495	0.00117
srg5ptimoneg.aig	2	1.00	1	54	0.23408	0.00235
texasifetch1p1.aig	7	1.00	0	172	1.49728	0.01558
texasifetch1p3.aig	7	1.00	0	172	1.49405	0.01516
viselevatorp1.aig	5	1.00	0	81	1.28970	0.01460
6s40p1.aig	0	0.00	0	1	0.50789	0.00254
6s40p2.aig	0	0.00	0	1	0.50274	0.00295
bobmiterbm1or.aig	0	0.00	0	1	0.04754	0.00100
bobsynth00neg.aig	0	0.00	0	1	0.27934	0.01959
bobtuint06.aig	0	0.00	0	1	0.04026	0.00657
pdtpmstwo.aig	2	1.00	0	200	2.32845	0.12187
pdtvsar8multip24.aig	7	1.00	0	805	98.40888	1.53823
pdtvsar8multip26.aig	7	1.00	0	805	2.32845	0.12187
6s318r.aig	2	1.00	1	673	98.40888	1.53823
counterp0.aig	4	5.67	24	285	0.51207	0.00407
counterp0neg.aig	4	6.13	26	313	0.57826	0.00331
pdtvishuffman7.aig	5	1.12	10	317	4.37308	0.05090
pdtvismiim3.aig	12	1.06	18	811	10.21632	0.07314
srg5ptimo.aig	3	4.28	23	259	0.78126	0.01102

B.4 PriorityQueue

Name	Frames	ALC	CTIs	Queries	Mean Time (s)	SD of Time (s)
counters2.aig	4	2.94201	6	116	0.06452	0.00453
counters2_neg.aig	2	1.16667	2	22	0.00385	0.00018
counters3.aig	5	6.39181	45	1117	11.99356	0.12766
counters3_neg.aig	1	1.00000	1	5	0.00260	0.00012
counters4.aig	6	8.32578	113	3119	25.97405	0.07161
simple1.aag	0	0.00000	0	1	0.00014	0.00001
simple2.aag	1	1.00000	0	3	0.00031	0.00002
simple3.aag	3	1.00000	0	10	0.00064	0.00004
simple4.aag	2	1.00000	0	7	0.00116	0.00002
simple5.aag	0	0.00000	0	1	0.00019	0.00002
simple6.aag	0	0.00000	0	1	0.00018	0.00002
simple7.aag	1	1.00000	1	5	0.00110	0.00002
simple_counters.aig	3	3.26111	7	90	0.29290	0.00215
simpler_counters.aig	2	1.00000	1	15	0.00279	0.00008
bj08aut1.aig	1	1.00000	0	6	0.00871	0.00010
bj08aut5.aig	1	1.00000	0	6	0.02802	0.00109

bj08goodbakerycyclef7.aig	1	1.00000	1	5	9.65093	0.30866
neclaftp5001.aig	5	1.00000	0	98	0.42721	0.02481
neclaftp5002.aig	5	1.00000	0	98	0.42939	0.02758
pdtvisblackjack0.aig	1	1.00000	0	107	5.27433	0.15706
pdtvisblackjack1.aig	1	1.00000	0	107	5.35653	0.14178
pdtvisblackjack2.aig	1	1.00000	0	107	5.30165	0.18996
pdtvisblackjack3.aig	1	1.00000	0	107	5.40703	0.20540
pdtvisblackjack4.aig	1	1.00000	0	107	5.25143	0.20057
pdtvisbpb1.aig	5	1.00000	0	227	8.91668	0.21769
pdtvisgray0.aig	3	1.00000	0	16	0.00516	0.00040
pdtvisgray1.aig	3	1.00000	0	16	0.00419	0.00039
pdtvisheap04.aig	5	1.00000	0	80	2.12859	0.08487
pdtvisheap07.aig	5	1.00000	0	80	2.18151	0.11410
pdtvisheap11.aig	5	1.00000	0	80	2.18870	0.09766
pdtvishuffman2.aig	11	1.00000	0	505	16.37136	0.42377
pdtvishuffman5.aig	0	0.00000	0	1	0.01701	0.00276
pdtvisrethersqo3.aig	0	0.00000	0	1	0.01225	0.00128
pdtvistictactoe00.aig	4	1.00000	0	70	1.43449	0.09377
pdtvistictactoe01.aig	0	0.00000	0	1	0.01548	0.00352
pdtvistictactoe03.aig	0	0.00000	0	1	0.01387	0.00160
pdtvistictactoe04.aig	0	0.00000	0	1	0.01351	0.00094
pdtvistictactoe05.aig	0	0.00000	0	1	0.01328	0.00080
pdtvistictactoe06.aig	0	0.00000	0	1	0.01353	0.00145
pdtvistictactoe07.aig	0	0.00000	0	1	0.01575	0.00404
pdtvistictactoe08.aig	0	0.00000	0	1	0.01272	0.00027
pdtvistictactoe09.aig	0	0.00000	0	1	0.01459	0.00232
pdtvistictactoe11.aig	4	1.00000	0	70	1.39019	0.07897
pdtvistictactoe12.aig	4	1.00000	0	70	1.41232	0.09649
pdtvistwo0.aig	2	1.00000	0	59	0.52194	0.04320
pdtvistwo1.aig	2	1.00000	0	59	0.51802	0.04900
pdtvisvending03.aig	6	1.00000	0	87	2.05030	0.11085
pdtvisvending06.aig	6	1.00000	0	87	2.08996	0.10079
pdtvisvsar02.aig	7	1.00000	0	538	42.31584	0.96840
pdtvisvsar18.aig	7	1.00000	0	528	41.94855	0.15602
shortp0.aig	3	2.22876	3	64	0.07889	0.00147
shortp0neg.aig	2	1.00000	2	38	0.04734	0.00146
srg5ptimoneg.aig	2	1.00000	2	91	0.39672	0.00995
texasifetch1p1.aig	7	1.00000	0	172	2.64989	0.01620
texasifetch1p3.aig	7	1.00000	0	172	2.66928	0.02123
viselevatorp1.aig	5	1.00000	0	81	1.94468	0.01487
6s40p1.aig	0	0.00000	0	1	0.51492	0.01101
6s40p2.aig	0	0.00000	0	1	0.50456	0.00830

bobmiterbm1or.aig	0	0.00000	0	1	0.04771	0.00099
bobsynth00neg.aig	0	0.00000	0	1	0.25048	0.00213
bobtuint06.aig	0	0.00000	0	1	0.03164	0.00052
pdtpmstwo.aig	2	1.00000	0	200	4.07142	0.04582
pdtvsar8multip24.aig	7	1.00000	0	805	260.41858	4.71720
pdtvsar8multip26.aig	7	1.00000	0	805	258.51646	1.77145
6s318r.aig	2	1.00000	2	804	35.26618	0.85674
pdtvishuffman7.aig	5	2.72774	13	487	12.65960	0.39817
srg5ptimo.aig	3	5.69675	7	281	1.73346	0.03948

B.5 CTG

Name	Frames	ALC	CTIs	CTGs	Queries	Mean Time (s)	SD of Time (s)
counters2.aig	5	1.99	10	8	449	0.09618	0.00159
counters2_neg.aig	2	1.00	1	0	11	0.00219	0.00036
counters3.aig	21	4.39	91	242	15225	6.51102	0.01537
counters3_neg.aig	1	1.00	0	0	2	0.00123	0.00023
counters4.aig	24	5.75	361	683	72170	83.56571	2.59036
simple1.aag	0	0.00	0	0	1	0.00017	0.00005
simple2.aag	1	1.00	0	0	3	0.00030	0.00003
simple3.aag	3	1.00	0	0	10	0.00070	0.00034
simple4.aag	2	1.00	0	0	7	0.00120	0.00023
simple5.aag	0	0.00	0	0	1	0.00020	0.00003
simple6.aag	0	0.00	0	0	1	0.00023	0.00005
simple7.aag	1	1.00	0	0	2	0.00063	0.00012
simple_counters.aig	3	1.00	5	11	1430	1.40536	0.14380
simpler_counters.aig	2	1.00	1	0	16	0.00334	0.00065
bj08aut1.aig	1	1.00	0	0	6	0.00949	0.00077
bj08aut5.aig	1	1.00	0	0	6	0.03042	0.00291
bj08goodbakerycyclef7.aig	1	1.00	0	0	2	0.60639	0.03626
neclaftp5001.aig	5	1.00	0	0	98	0.39299	0.02680
neclaftp5002.aig	5	1.00	0	0	98	0.39532	0.03351
pdtvisblackjack0.aig	1	1.00	0	0	107	5.36670	0.26220
pdtvisblackjack1.aig	1	1.00	0	0	107	5.40568	0.25771
pdtvisblackjack2.aig	1	1.00	0	0	107	5.31258	0.18390
pdtvisblackjack3.aig	1	1.00	0	0	107	5.36394	0.20737
pdtvisblackjack4.aig	1	1.00	0	0	107	5.31176	0.19112
pdtvisbpb1.aig	5	1.00	0	0	227	4.82792	0.21683
pdtvisgray0.aig	3	1.00	0	0	16	0.00565	0.00066
pdtvisgray1.aig	3	1.00	0	0	16	0.00751	0.00271
pdtvisheap04.aig	5	1.00	0	0	80	1.33627	0.06150

pdtvisheap07.aig	5	1.00	0	0	80	1.37171	0.08407
pdtvisheap11.aig	5	1.00	0	0	80	1.37770	0.09415
pdtvishuffman2.aig	11	1.00	0	0	505	7.52349	0.32952
pdtvishuffman5.aig	0	0.00	0	0	1	0.01660	0.00243
pdtvisrethersqo3.aig	0	0.00	0	0	1	0.01162	0.00095
pdtvistictactoe00.aig	4	1.00	0	0	70	0.83136	0.01176
pdtvistictactoe01.aig	0	0.00	0	0	1	0.01277	0.00085
pdtvistictactoe03.aig	0	0.00	0	0	1	0.01280	0.00087
pdtvistictactoe04.aig	0	0.00	0	0	1	0.01256	0.00070
pdtvistictactoe05.aig	0	0.00	0	0	1	0.01269	0.00084
pdtvistictactoe06.aig	0	0.00	0	0	1	0.01273	0.00093
pdtvistictactoe07.aig	0	0.00	0	0	1	0.01283	0.00066
pdtvistictactoe08.aig	0	0.00	0	0	1	0.01294	0.00106
pdtvistictactoe09.aig	0	0.00	0	0	1	0.01274	0.00092
pdtvistictactoe11.aig	4	1.00	0	0	70	0.85269	0.01186
pdtvistictactoe12.aig	4	1.00	0	0	70	0.84277	0.01139
pdtvistwo0.aig	2	1.00	0	0	59	0.31924	0.00617
pdtvistwo1.aig	2	1.00	0	0	59	0.30470	0.00621
pdtvisvending03.aig	6	1.00	0	0	87	1.17390	0.01523
pdtvisvending06.aig	6	1.00	0	0	87	1.16157	0.01636
pdtvisvsar02.aig	7	1.00	0	0	538	16.42766	0.06902
pdtvisvsar18.aig	7	1.00	0	0	538	16.80929	0.08084
shortp0.aig	3	1.78	3	0	57	1.95699	0.03510
shortp0neg.aig	2	1.00	1	0	21	0.02501	0.00124
srg5ptimoneg.aig	2	1.00	1	0	54	0.23244	0.00545
texasifetch1p1.aig	7	1.00	0	0	172	1.48538	0.02581
texasifetch1p3.aig	7	1.00	0	0	172	1.47737	0.02139
viselevatorp1.aig	5	1.00	0	0	81	1.27517	0.01586
6s40p1.aig	0	0.00	0	0	1	0.52503	0.01524
6s40p2.aig	0	0.00	0	0	1	0.51529	0.00512
bobmiterbm1or.aig	0	0.00	0	0	1	0.04958	0.00102
bobsynth00neg.aig	0	0.00	0	0	1	0.25659	0.00158
bobtuint06.aig	0	0.00	0	0	1	0.03285	0.00073
pdtpmstwo.aig	2	1.00	0	0	200	2.18252	0.00804
pdtvsar8multip24.aig	7	1.00	0	0	805	91.90440	0.54409
pdtvsar8multip26.aig	7	1.00	0	0	805	93.45550	0.72848
6s318r.aig	2	1.00	1	0	673	28.21434	0.16409
counterp0.aig	4	10.13	20	85	7286	17.87033	0.73647
counterp0neg.aig	4	10.49	23	130	10834	22.38433	0.67397
pdtvishuffman7.aig	5	1.02	9	9	8443	153.77111	2.98684
pdtvismiim3.aig	12	1.45	4	23	15933	28.40033	0.23470
srg5ptimo.aig	3	4.28	56	240	61482	1.47170	0.01905

Appendix C

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 [12].

Symbolic model checking techniques that rely on SAT solvers provide an alternative to BDD-based approaches. SAT-based approaches include bounded model checking [12] and k-induction [20], but both of these approaches involve unrolling the transition relation, which can lead to long SAT solver queries.

IC3 [15] 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 [20], and the SAT queries that IC3 makes tend to be simpler [16].

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 [15, 16].

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 [35].

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`