Lauren Pick

A Model Checker Using IC3

Computer Science Tripos – Part II

Homerton College

April 1, 2016

Proforma

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

Original Aims of the Project

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

Work Completed

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

Special Difficulties

None.

Declaration

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

Signed [signature]

Date [date]

Contents

Chapter 1

Introduction

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

One formal verification technique is model checking. Given a model of the system and a specified property, a model checker will automatically check whether all reachable states in the system satisfy this property.

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

A brief description of symbolic model-checking and SAT-based model-checking follows to provide further context for this project, which focuses on implementing the IC3 algorithm, a SAT-based model-checking algorithm.

1.1 Symbolic Model Checking

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

The initial approach to the model checking problem involved explicitly considering each reachable state in the model. Symbolic model checking arose as a method of mitigating the effects of the state explosion problem to some extent. By representing states and the transition relation between them as boolean expression, symbolic model checking allows sets of states to be represented efficiently as boolean expressions involving state variables as opposed to an explicit list of each individual state in the set [11].

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

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

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

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

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

1.2 The IC3 Algorithm

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

As in k-induction, properties are proved through induction: the algorithm considers reachable sets of states k steps away from the initial state until reaching a fixed point. As with later variants of k-induction, the IC3 algorithm also discovers new invariants if the initial assumptions are not strong enough to prove the desired property, but unlike k-induction, the safety property guides the discovery of the invariants. As a result, the discovered invariants are more relevant for proving the safety property [5].

Furthermore, IC3 does not unroll the transition relation as k-induction or other BMC-

based methods do, but instead considers at most one step of the transition relation at a time, leading to smaller, simpler SAT queries. As a result, IC3 requires less memory than BMC-based methods in practice [5].

1.3 Further Work

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

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

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

1.4 Project Aims

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

Chapter 2

Preparation

2.1 Requirements Analysis

The model checker requires a way of taking input models and also requires a way to solve SAT queries. I chose the AIGER format for representing the hardware models and the MiniSat SAT solver for answering SAT queries, resulting in a need for an AIGER parser and an interface to MiniSat in Haskell.

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

In summary, the required components are as follows:

- AIGER parser
- MiniSat interface
- Transition system representation
- IC3 algorithm

2.2 Tools Used

The tools used in the project and what they were used for are as follows:

- Git was used for version control, with GitHub providing backup storage
- Haddock was used for documentation
- HUnit was used for testing
- Criterion was used for benchmarking
- Cabal was used for package/dependency management
- The `hsc2hs` preprocessor eased writing Haskell interfaces to C code

- The Aiger Utilities' parser for comparison with and as an alternative to the one developed as part of the project
- The `bliftoaig` and `aigtoaig` tools eased the specification of models by converting from human-readable BLIF and ASCII AIG formats to the binary AIG format
- The MiniSat SAT solver was used for handling SAT queries

2.3 Model Specification

I used both the AIGER format and Berkeley Logic Interchange Format (BLIF) to specify some of the example hardware models. The hardware models that the model checker accepts as input are specified using the AIGER format, which is the format used to specify hardware models in the Hardware Model Checking Competitions; however using the AIGER format to specify larger models was cumbersome, so such models were specified using BLIF and converted to AIGER format using the AIGER library's `bliftoaig` tool.

The model checker takes as input models formatted using either the ASCII or binary AIGER formats described below.

2.3.1 AIGER

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, NOT gates, and latches.

The AIGER format has both an ASCII and binary version, where the ASCII format is more flexible and human readable, and the binary version is more compact. The Hardware Model Checking Competition's examples are in the latter of the two.

A new version of the AIGER format is currently under development, with examples from HWMCC'14 using the new version.

Old version

All AIGER files 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

Variables are represented in AIGER with even integers greater than 1, with 0 representing the constant value *False* and 1 representing *True*. An odd integer value $i + 1$ represents the negated value of the literal represented by i. In other words, a variable named x is represented by $x \times 2$, which gives the positive literal x, and $x \times 2 + 1$ gives the negative literal $\neg x$.

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 (greater than 2) that represents its corresponding variable name, and outputs are specified similarly as single indices. Latches initially begin with value 0 and are specified by giving the index representing their corresponding variable name followed by the integer that represents the literal giving their next-state value. AND gates are specified by giving their indices and their two inputs literals' integer representations.

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

Figure 2.1: A circuit specified in the old ASCII AIGER format

For example, figure 2.1 specifies a circuit with no inputs, two latches with indices 2 and 4 and one AND gate 6 that takes the outputs of the two latches as inputs and whose output is the single output of the circuit.

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

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

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

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

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

and

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

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

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

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

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

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

New version

The new AIGER format begins with a header of the form

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

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

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

Variables are represented in the same manner as the old format and are specified after the header in the same order that the counts are given in the header with the exception of AND gates, which occur at the end of the file. Latches' initial values can also be specified now with an additional `0` or `1` after the next-state literal's index. If the initial value is omitted, the initial value is assumed to be 0 as in the old version of the format. The header can also be truncated after giving the number of AND-gates if the remaining counts are all zero, allowing any parser for the new AIGER format to be backwards-compatible.

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

2.4 The IC3 Algorithm

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

2.4.1 Transition Systems

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

A single state of the system (or a singleton set containing that state) is specified through the assignment of all state variables in x.

2.4.2 Frames

As with other inductive approaches to model checking, the IC3 algorithm maintains a set of k frames $F_0, \ldots, F_k$, where each frame F_i is a set of clauses whose conjunction represents an overapproximation of the set of states that reachable by the transition system in i steps (so, for example, F_0 is just the initial state set I).

If $F_i = F_{i+1}$ holds for any i at any point, then a fixed point has been found, and the algorithm terminates.

2.4.3 Checking Properties

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

The consecution query $F_k \wedge T \Rightarrow P'$ for a frame F_k is used to check whether the property P necessarily holds in the next frame.

The algorithm extends its set of frames $F_0, \ldots, F_k$ with a new frame F_{k+1} by running the consecution query. If the consecution query is successful, then the new frame F_{k+1} with clause $\{P\}$ can be added. All clauses in frame F_k are then propagated to F_{k+1}: For each clause $C \in F_k$, if $F_k \wedge T \Rightarrow C'$ holds, then C' is added to 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 a *counterexample to induction* (CTI) state.

The algorithm aims to refine the approximation F_k of the set of states reachable in at least k steps by finding a clause c that holds at depth k such that $F_k \wedge c \wedge T \Rightarrow P'$ holds:

Such a c must be such that for the counterexample to induction s,

$$c \Rightarrow \neg s,$$

with an obvious choice being $c = \neg s$, which can be found easily using a SAT solver. In practice, it is better to generalize $\neg s$ and choose a smaller c, which can allow for a set of several states to be eliminated in F_k rather than just eliminating s. In particular, the choice of c that maximises the number of states eliminated in F_k is the minimal inductive subclause of $\neg s$ relative to frame F_k.

2.4.4 Minimal Inductive Subclauses

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

$$F_k \wedge T \wedge c \Rightarrow c'$$

holds.

The minimal inductive subclause can be found by dropping each literal in s in turn and checking if the resulting clause is inductive relative to F_k:

Function *mic(s,k)*:
    foreach *literal l in s* **do**
        c := s \ {l}
        if *c is inductive relative to frame at depth k* **then**
            s := c
        end
    end
    return s
end

Finding the minimal inductive subclause can be computationally expensive, so it is often approximated in practice.

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

2.4.5 Counterexamples to Generalization

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

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

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

Chapter 3

Implementation

The implementation can be broken up into four main components: the AIGER parser, the MiniSat interface, the hardware model representation, and the model checker.

3.1 Parser

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

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

3.1.1 Model

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

Literals are represented by a `Lit` data structure in `Parser.AigModel`, which instead of representing a positive literal x as $2 \times x$ and negative literal $\neg x$ as $2 \times x + 1$, represents positive literal x as `Lit` $x - 1$ and negative literal $\neg x$ as `Neg` $x - 1$. The subtraction by 1 allows variable indices to start at 0 rather than 1, since starting at 1 is unnecessary because the `Lit` data struture also contains the separate `Bool` constructor for representing the values `True` and `False`.

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

3.2 Minisat Interface

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

3.2.1 Relevant MiniSat Functions

To solve a SAT query, MiniSat creates an instance of a `Solver` object, which contains a set of variables, sets of clauses represented by `VecLit` instances that must be satisfied each SAT query, a model and a conflict vector. The `solveWithAssumps` function makes a SAT query that must satisfy all of the clauses in the `Solver` as well as all the literals supplied in the assumption `VecLit`.

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

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

3.2.2 C Wrapper

Much of the C wrapper is straightforward: 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. I also added an additional `result` struct to allow for the results of a SAT query to be returned from a single function call. The struct contains not only the result of the SAT query, but also pointers to the model and conflict vector (if any) of the `Solver`. The wrapper function for `solveWithAssumps` returns a pointer to a `result` struct rather than just whether or not the query was satisfiable.

3.2.3 Haskell Interface

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

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

Many of the functions and datatypes in the interface are analogous to functions and structs in the C wrapper and C++ implementation of MiniSat. For example, the `Solver` datatype is an analogue to the MiniSat `Solver` object, and itself contains a pointer to an instance of a MiniSat `Solver` object. Similarly, functions such as `solveWithAssumps`

work analogously to the C wrapper's `solveWithAssumps`, returning a `Result` that contains whether or not the query was satisfiable and the model or conflict vector (if any).

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

3.3 Hardware Models

3.3.1 Representation

Literals

The `Lit` data structure in `Model.Model` gives the representation for literals in the model checker. The `Var` constructor gives positive current-state (unprimed) literals, the `Neg` constructor gives negative current-state literals, and the `Var'` and `Neg'` constructors respectively give positive and negative next-state (primed) literals.

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`. The inputs i and state variables x in the transition system $T(i, x, I, T)$ are not distinguished, and the total count of variables is kept in `vars`. Clauses that specify the initial state I are kept in `initial`. The transition relation T is captured by `transition`, a list of both clauses that specify latches and clauses that specify AND gates. The literal that gives the safety property is given by `safe`.

3.3.2 Construction

The `Model.Model` module contains functions to convert the `Model` data structure in `Parser.AigModel` into the hardware model representation used by the model checker.

3.4 Model Checking

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

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

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

3.4.1 Overall structure

All versions of the model checker begin with a call to the `prove` function, passing a `Model` and the `Lit` that is the safety property. The `prove` function first creates the frame for the initial state and makes an initiation query. If the query is unsuccessful and returns `False`, then `prove` can just return `False`, since the safety property does not hold in the initial state. Then `prove` calls `prove'`, which has different implementations based on whether the version is recursive or not.

Recursive

In the general structure for the recursive implementations of the model checker, `prove'` takes as arguments the `Model` for the hardware, the `Lit` that gives the safety property, the frontier `Frame` (i.e. the `Frame` that approximates the reachable set at the greatest depth considered so far), and the rest of the `Frame`s in a list ordered by increasing depth.

The `prove'` function makes a consecution query for the safety property and frontier frame `frame`. If the consecution query returns `True`, then `prove'` creates a new frame and calls `pushFrame` which uses the `push` function to push as many clauses as possible from `frame` to the new frame. If `push` indicates that a fixed point has been reached, then `pushFrame` returns `True`, but otherwise `pushFrame` calls `prove'` recursively with the new frontier frame added to the list of frames.

If the consecution query returns `False`, then `prove'` calls `nextCTI` to find a (possibly incomplete, depending on the implementation version) assignment whose current-state literals give a CTI cube. The CTI is extracted from the result of `nextCTI` and passed along to `proveNegCTI` along with the frontier frame, the rest of the frames, and the safety property.

The function `proveNegCTI` attempts to show that the CTI s is unreachable at the current depth by showing that $\neg s$ holds in the current frame. The function checks first that $I \wedge \neg s \wedge s'$ is unsatisfiable (i.e. to check that $I \wedge \neg s \Rightarrow \neg s'$ and therefore that $\neg s$ is inductive relative to I). Then `pushNegCTI` finds the deepest frame (the frame F_k with the largest k) that comes before the frame that contains the counterexample to induction state relative to which $\neg s$ is inductive and adds the clause $\neg s$ to all frames up to that depth.

If this deepest frame F_{k-1} is the frame just before the frame that contains the CTI, then since $\neg s$ is inductive relative to F_{k-1}, $\neg s$ can also be added to F_k, establishing that $\neg s$ is not reachable in k steps. The `pushNegCTI` function in this case yields a triple `(Nothing, acc', f')`, where `acc'` gives a list containing the representation of frames

$F_0, \ldots, F_{k-1}$ after having $\neg s$ conjoined to them, and `f'` contains the representation of frame F_k after having $\neg s$ conjoined to it. The function `proveNegCTI` then checks again if the safety property is inductive relative to the updated F_k. If so, then `proveNegCTI` has successfully proven enough CTIs are unreachable at depth k. Otherwise, `proveNegCTI` calls `nextCTI` to extract the next CTI to prove unreachable at depth k and calls itself recursively with this newly-discovered CTI.

If the deepest frame F_{k-1} to which $\neg s$ is relatively inductive is not the frame just before the CTI-containing frame, then `proveNegCTI` uses `nextCTI` to find a new CTI z that is a counterexample for the property $\neg s$ being relatively inductive to F_k. It then calls itself recursively with this new CTI to prove, and the frames up to F_{k-1} rather than F_k. After updating frames $F_0, \ldots F_{k-1}$ to prove that $\neg z$ holds at F_{k-1}, the call to `negCTI` to prove $\neg s$ is repeated using the updated frames $F_0, \ldots, F_{k-1}$.

If any call to `proveNegCTI` ever reaches the point where the only frame it is called with is F_0, then `proveNegCTI` has failed to prove a property necessary to show that the error state is unreachable, and the model checker can then give the result that the safety property does not hold.

When `proveNegCTI` successfully proves a negated CTI unreachable, `prove'` then propagates clauses through all frames from the inital frame F_0 to the frontier frame and checks for a fixed point. If a fixed point has been found, then the safety property holds at all reachable states and the model checker can return `True`. Otherwise, `prove'` calls itself recursively to prove the safety property at the same depth, but with the updated frames (with negated CTIs conjoined to them).

Priority queues

The priority queue implementation keeps track of what to prove next by using a priority queue of proof obligations instead of through recursive calls that explicitly specify which property to prove at which depth. A proof obligation is a pair (s, i) of a state s that is either a set of bad states or a counterexample to induction and a depth i. When the model checker encounters a proof obligation (s, i) as the highest-priority element of the queue, it must prove $\neg s$ at depth i to fulfill (s, i). The presence of proof obligation (s, i) on the priority queue also indicates that for all values j with $0 \leq j < i$, (s, j) has already been fulfilled.

The implementation represents proof obligations (s, i) using the `Obligation` type, which is a triple `(Int, Int, Clause)` of the depth i, a rank for deciding the ordering of proof obligations at the same depth within the priority queue, and the clause $\neg s$.

In the *PriorityQueue* implementation, `prove'` also takes a `Model` as a parameter but otherwise takes a `MinQueue` (the minimal element has the highest priority) of `Obligations` containing the priority queue with only `Obligation (1, 0, [prop])` in it, and the initial frame. The function `prove'` then invokes `proveObligations`, the main function for the *PriorityQueue* implementation.

The `proveObligations` function removes the minimum (i.e. highest-priority) `Obligation` representing (s, i) from the priority queue and makes the consecution query $F_{i-1} \wedge T \Rightarrow \neg s$. If the consecution query returns `True`, then `proveObligations` invokes

`pushFrame`, which works the same as in the recursive version, except that the recursive call to `prove'` is replaced with a recursive call to `proveObligations` with an updated priority queue that has proof obligation $(s, i + 1)$.

If the consecution query instead returns `False`, then the negated CTI $\neg s$ is found by negating the extracted current-state literals taken from the value given by an invocation of `nextCTI`. The `proveObligations` function checks that $\neg s$ does not hold in the initial state, adds $\neg s$ to frame F_0, and then invokes the `propagate` function to propagate clauses until one of the following occurs:

- The depth i in the proof obligation has been reached
- The clause $\neg s$ can no longer be pushed
- A fixed point has been reached.

In the first case, `proveObligations` enqueues obligation (s, i) with a rank that gives it a priority such that it occurs after all other proof obligations at depth i and calls itself recursively. In the second case, `propagate` gives the greatest j such that $\neg s$ has been shown to hold at j-steps from the initial state (i.e. $\neg s$ is inductive relative to F_{j-1}. Then `proveObligations` enqueues obligation (s, j) with a rank that gives it a priority such that it occurs after all other proof obligations at depth j and calls itself recursively. In the third case, `proveObligations` returns a value containing `True`, since the safety property has been proven to hold at all reachable states.

3.4.2 Frames

The `Frame` data structure represents frames in the IC3 algorithm. 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 conjunctive normal form (CNF), where each disjunct is one of the clauses passed to the solver (assumptions passed to the solver can be taken as a set of single-literal clauses that are also part of the conjunction). As a result, the representation for the query $I \Rightarrow P$ for a frame I and a clause P makes use of the logical equivalence between $I \Rightarrow P$ and $\neg(\neg P \wedge I)$.

Using the fact that the formula $\neg(\neg P \wedge I)$ is true iff $\neg P \wedge I$ is unsatisfiable, and deMorgan's laws gives the following implementation of the initiation query:

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

3.4.4 Consecution

Similarly to how the implication in the initiation query is converted to an equivalent logical expression that renders a CNF query to a MiniSat `Solver` to determine the satisfiability of the query, the consecution query's implementation uses the logical equivalence of $F_k \wedge T \Rightarrow P'$ and $\neg(\neg P' \wedge F_k \wedge T)$. Again, along with deMorgan's laws, this yields the implementation of the consecution query (where it is assumed that the `Frame`'s `Solver` contains the `transition` clauses of the `Model`.

3.4.5 Counterexamples to Induction

When the consecution query $F_k \wedge T \Rightarrow P$ in `prove'` fails, then a call to the function `nextCTI` gives a cube (a conjunction of literals) whose current-state literals give relevant CTI for the query. After, in the recursive version, `proveNegCTI` attempts to prove that the CTI cannot be reached (which may involve further calls to `nextCTI`), and in the *PriorityQueue* version, `proveObligations` enqueues a proof obligation for proving the negated CTI.

Basic

In the *Basic* implementation of the IC3 algorithm, `negCTI` 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 are then extracted from the model in the function that called `negCTI`.

Smaller Counterexamples to Induction

In the other implementations of the algorithm, `negCTI` again asks for a model m for the satisfiable query $\neg P' \wedge F_k \wedge T$. The only literals in m that must necessarily be included in the CTI are those current-state literals that result in the unsatisfiability of $m \wedge P' \wedge T$. That is, the current-state literals of any subcube q of m for which $q \wedge P' \wedge T$ holds is also a valid CTI. 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.

3.4.6 Propagation

Basic

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

conjoined to the frame represented by `f'`, and if it is not, then the function must have `False` as the first element in the pair it returns.

Subsumed clauses

The basic implementation's `push` function avoids unnecessary consecution queries by only considering clauses in `f` that are not in `f'`. Further consecution queries may be eliminated by considering the clauses in `f` that are subsumed by other clauses.

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

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 Generalization

While the algorithm describes generalization as involving finding the minimal inductive subclause (MIC), finding the MIC for a clause is in practice inefficient, and all implemented versions of generalization `inductiveGeneralization` involve approximating the MIC.

Simple

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

Minimal Inductive Subclauses and Counterexamples to Generalization

The more elaborate implementation of generalization implements the full (but limited in number of attempts) MIC algorithm implementation with the ability to deal with CTGs.

When attempting to drop a literal l from clause c, the `down` function is called on the clause $c \setminus l$ with a list of past frames (which is all the frames except the deepest on the inital call to `down`), a list of current and future frames (which contains only the deepest frame on initial call to `down`), and a limit on the recursion depth `r`.

The `down` function checks, as in the simple approximation for MIC, for the satisfiability of $I \Rightarrow c$ and $F_k \wedge c \wedge T \Rightarrow c'$. The difference is that `down` does not immediately give a value if $I \Rightarrow c$ is true and $F_k \wedge c \wedge T \Rightarrow c'$ is not; in this case, the negated cTG `negCTG` is acquired by taking and negating the current literals in the model the SAT solver gives for $\neg c' \wedge c \wedge T \wedge F_k$.

The `down` function then finds the deepest frame for which `negCTI` is inductive, attempts to generalize `negCTI` relative to that frame with a recursive call (with a decremented value of `r`)

Chapter 4

Evaluation

4.1 Benchmarking

Benchmarks were taken for the performance of the different versions of the model checker and reference implementation on fifteen handwritten examples, fifty examples from HWMCC'10, eleven (twenty) examples from HWMCC'11, and two (ten) examples from HWMCC'13. For each example, forty benchmarking samples were taken, and if the elapsed time for attempting to solve an example takes longer than ten minutes, the attempt is considered to have timed out.

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

4.2 Performance Impact of Variations

The overall performance of the model checker is heavily dependent on the size and number of SAT-solver queries; profiling consistently reveals that functions in the Minisat module consume the most time when solving examples.

4.2.1 Smaller Counterexamples to Induction

Discovering smaller CTIs leads, as expected, to a smaller number of literals per clause and smaller SAT queries, giving consistent performance improvements over the basic CTI-finding implementation for nontrivial examples (i.e. examples that require finding counterexamples). The *BetterCTI* version of the implementation can solve seven more examples than the *Basic* version without timing out.

4.2.2 Propagation

Also as expected, removing subsumed clauses results in a smaller number of literals per clause, resulting in the *BetterPropagation* version having slightly better performance than the *BetterCTI* version.

4.2.3 Counterexamples to Generalization

The *CTG* version that deals with CTGs performs the same as or worse than the *Better-Propagation* version on examples, most likely because the examples used are too small for the performance benefits of using CTGs to eliminate more states to overcome the overheads of finding and proving negated CTGs. Similar results can be found in the performance of the reference implementation with basic generalization and improved (CTG-using) generalization on the same examples: for these examples, the reference implementation performs better with CTG-handling disabled.

4.2.4 Priority Queues

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

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

4.3 Performance Comparison

Chapter 5

Conclusion

5.1 Summary

Summarize the project and accomplishments

5.2 Further extensions

Mention further extensions that could be implemented.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Project Proposal

Introduction and Description of the Work

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

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

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

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

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

Starting Point

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

Substance and Structure of the Project

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

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

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

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

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

5. **Writing the dissertation**

Possible Extensions

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

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

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

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

Success Criteria

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

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

Timetable: Workplan and Milestones

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

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

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

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

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

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

4. **19 November 2015**

   Begin implementing the IC3 algorithm.

5. **Michaelmas vacation**

   Continue implementing the IC3 algorithm.

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

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

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

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

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

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

9. **Easter vacation**

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

10. **21 April 2016 – 4 May 2016**

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

11. **5 May 2016 – 13 May 2016**

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

Resources Required

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

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