Lauren Pick

A Model Checker Using IC3

Computer Science Tripos – Part II

Homerton College

March 19, 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

Model checking attempts to verify that a hardware or software system has certain properties. Given a model of the system and a specified property, a model checker will automatically check whether all states in the system satisfy this property.

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].

A brief description 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

Symbolic model checking was originally invented for use with ordered binary decision diagrams (BDDs), which provide an efficient representation of boolean formulas using canonical forms for the formulas. 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 SAT procedures. 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 HWMCC'10, 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 aims to implement a basic form of the IC3 algorithm in Haskell. The model checker should be able to correctly check several small examples.

Chapter 2

Preparation

2.1 Requirements Analysis

A SAT-based 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 requirements are as follows:

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

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. 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.

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 indices greater than 1, with 0 indexing the constant value *False* and 1 indexing *True*. An odd index $i+1$ represents the negated value of the variable represented by i.

The different components are specified 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 default to 0 (?) and are specified by giving the index representing their corresponding variable name followed by the index that gives their next-state value. AND gates are specified by giving their indices and their two inputs' indices.

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

A circuit specified in the old ASCII AIGER format 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, inputs are not explicitly listed as in the ASCII format, and if the first non-input specified has index n, then there are assumed to be $n-2$ inputs with indices 2 to $n-1$. Additionally, AND-gates ...

New version

The new AIGER format begins with a header of the form

```
V M I L O A B - - -
```

2.3.2 Berkeley Logic Interchange Format

BLIF

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:

```
mic (s, k):
  for literal l in s:
    c := s \ {l}
    if c is inductive relative to frame at depth k
      s := c
  return s
```

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

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

Briefly mention sections in the chapter and the example models written in AIGER and BLIF formats.

3.1 AIGER Parser

Discuss the implementation of the AIGER parser. In particular, mention the handling of both the older and newer AIGER format versions for both the ASCII and binary versions of the format and the representation of AIG models.

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 Transition Systems

3.3.1 Representation

Describe how transition systems and their components are represented.

3.3.2 Construction

Describe how transition systems are constructed given AIG models.

3.4 Model Checking

3.4.1 Initiation

Describe how the step involving the initiation query is implemented.

3.4.2 Consecution

Describe how the step involving the consecution query is implemented.

3.4.3 Counterexamples to Induction

Describe how the implementation discovers counterexamples to induction and proves them unreachable.

3.4.4 Propagation

Basic

Subsumed clauses

3.4.5 Generalization

Simple

Minimal Inductive Subclauses

Counterexamples to Generalization

Chapter 4

Evaluation

4.1 Performance Impact of Improvements

Discuss performance comparison of the naive implementation of the model checker and the final implementation of the model checker.

4.2 Performance Comparison

Discuss performance comparison of final implementation with IC3 reference implementation, taking into account differences between the implementations.

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`