Solving RERS 2014 problems with IC3

H.N. Hindriks
h.n.hindriks@student.utwente.nl

April 18, 2015

1 Introduction

This report covers the individual project performed by H.N. Hindriks, under supervision of T. van Dijk and J.C. Van der Pol, which took place during the 3rd quartile of 2014-2015.

IC3 is a rather new algorithm for model checking, developed by Fabio Somenzi and Aaron Bradley [2]. The algorithm relies on providing a SAT solver (relatively) small queries in order to construct either an inductive proof or a counterexample to induction. The algorithm is described in more detail by Bradley [1].

The RERS challenge (which stands for Rigorous Examination of Reactive Systems), is an international contest currently held annualy. During the contest participants attempt to find bugs in reactive systems of varying complexity using testing and model checking tools.

The aim of the project was to acquire a better in-depth knowledge of the IC3 algorithm, while also experiencing the challenges encountered when model checking real-world-like programs.

Contents

2 Background

2.1 SAT

The IC3 algorithm uses a SAT solver to learn more about the system under verification. Before introducing the IC3 algorithm, it makes sense to briefly explain some basic principles on SAT solving.

An arbitrary propositional logic formula is called *satisfiable* when there exists an assignment of all the literals in the formula which satisfy the formula (also referred to as a model). This means that substituting the literals with their respective truth assignments results in the formula to be evaluated to $\top$.

This problem is known as the *Satisfiability* problem (or just shorthand SAT). A formula for which such a truth assignment exists is called satisfiable. Likewise, a formula for which such a truth assignment does not exist, is called unsatisfiable.

A SAT solver is a piece of software, which is able to decide whether a formula is satisfiable. When this is the case, most SAT solvers also return a proof in the form of a model.

2.2 CNF

A propositional logic formula is in Conjunctive Normal Form when it consists of a conjunction of clauses. Every propositional logic formula can be converted to an equivalent formula in CNF. See section 3.5 for some examples of formulae in CNF.

2.3 Tseitin transformation

Any formula in propositional logic can be converted to CNF by using DeMorgan's law and the distributivity laws (of $\wedge$ and $\vee$). However, this might create formulae which are exponential in size when compared to the original formula.

Using the Tseitin transformation, it is possible to convert an arbitrary formula to a formula in CNF which is not equal, but equisatisfiable to the original. This means that the new formula is satisfiable iff the original formula is satisfiable. The resulting formula is linear in size when compared to the original formula.

Furthermore, the Tseitin transformed formula also has a 1-to-1 correspondence with satisfiable assignments to the original formula. This means that a satisfiable assignment of the converted formula is easily transformed to a satisfiable assignment of the original formula.

The Tseitin transform introduces new literals for each operator. Each operator is viewed as a logic gate, consisting of some inputs, and a single output. For each operator, a literal is introduced to represent the output of a gate modelling that operator.

Given two inputs A, B and an output C we define the Tseitin transformation on the $\wedge$ and $\vee$ operations as follows [5]:

$$A \wedge B \approx (A \wedge B = C) \Rightarrow (\neg A \vee \neg B \vee C) \wedge (A \vee \neg C) \wedge (B \vee \neg C)$$
$$A \vee B \approx (A \vee B = C) \Rightarrow (A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg B \vee C)$$

When the above transformation is recursively applied on a formula, it is converted to a formula in CNF which is $\mathcal{O}(n)$ in size compared to the original formula.

2.4 IC3

IC3, which stands for 'Incremental Construction of Inductive Clauses for Indubitable Correctness', is a Model Checking algorithm which is able to construct inductive proofs or counterexamples for reachability problems.

The algorithm has three cubes as input:

- I, the initial state
- $T(s, s')$, the transition relation
- P, the property which should hold in all states

In the following sections, a single apostrophe will denote the 'next' state of literals, as defined by T.

The algorithm returns as output either one of the following:

- An *inductive strengthening*, a frontier set for which $F \wedge T \Rightarrow F'$ and $F \Rightarrow P$ hold.
- A *counterexample trace*, leading to a state where $\neg P$ holds.

2.4.1 Invariants

After applying Bounded Model Checking for $k = 0$ and $k = 1$, the following four invariants hold for the duration of the IC3 algorithm [1]:

$$I \Rightarrow F_0 \tag{1}$$
$$F_i \Rightarrow F_{i+1} \qquad 0 \leq i < k \tag{2}$$
$$F_i \Rightarrow P \qquad 0 \leq i \leq k \tag{3}$$
$$F_i \wedge T \Rightarrow F'_{i+1} \qquad 0 \leq i < k \tag{4}$$

These four invariants define frontier sets $F_0 \dots F_k$. A single frontier set F_i overapproximates all states from the transition system reachable within i steps or less.

During the execution of the IC3 algorithm, the value of k is increased, and consequently more more frontier sets are introduced. The algorithm also strengthens the frontier sets when spurious (unreachable) bad states are found.

2.4.2 Initialization

The algorithm is initialized by setting up an environment in which the aforementioned invariants hold. This is done by proving that the following formulae hold (using BMC):

$$I \Rightarrow P' \qquad k = 0$$
$$I \wedge T \Rightarrow P' \qquad k = 1$$

When one of the aforementioned formulae is disproven, the algorithm can immediately return a counterexample trace with a length of either 0 or 1. When the formulae hold, the algorithm initializes the following values:

$$k = 1$$
$$F_0 = I$$
$$F_1 = P$$

The IC3 algorithm keeps track of all remaining proof obligations using a priority queue. These proof obligations are generated during the execution of the algorithm. The proof obligations are removed from the queue in the order of dependency, resolving some proof obligations might resolve other proof obligations.

2.4.3 Main algorithm

The main part of the algorithm loops until either an inductive strengthening, or a counterexample trace is found.

The algorithm starts by testing whether F_k is an inductive strengthening by checking the unsatisfiability of $F_k \wedge T \Rightarrow P'$ (queried as $\text{sat}(F_k \wedge T \wedge \neg P')$).

When this is not the case, the SAT solver returns a state $s \in F_k$, from which $\neg P$ can be reached ($s \wedge T \Rightarrow \neg P'$). The state s is added to the proof obligation queue, and the refining phase of the algorithm is initiated.

2.4.4 Refining phase

The refining phase of the IC3 algorithm consists of a loop which refines the frontier sets in order to eliminate spurious counterexample traces. The loop runs until all counterexamples have been eliminated, or a real counterexample is found.

For each proof obligation we have a level $l \in \mathbb{Z}$ and a bad state $s \in F_l$, for which it holds that $s \wedge T \Rightarrow \neg P'$. Even though we have $s \in F_l$, it might be the case that s is actually unreachable, meaning that F_l is too weak.

In order to prove this, the algorithm attempts to find the largest (highest index) frontier set F_{ind} for which $F_{ind} \wedge \neg s \wedge T \Rightarrow \neg s'$ is satisfiable.

When F_{ind} is not found, it means that s is reachable from I, and a counterexample trace is returned. In the case that F_{ind} is found, we attempt to refine $F_{ind} \dots F_k$ by strenghtening the frontier sets.

2.4.5 Strenghtening

From this point, we have a frontier set F_i from which the bad state s cannot be reached. We first invoke the MIC algorithm (described in section 2.4.7) with parameters $\neg s, F_i$, to obtain a minimal inductive subclause c (for which $F_i \wedge c \wedge T \Rightarrow \neg s'$ holds). The clause c is then added to all frontier sets $F_0 \dots F_i$.

We now verify whether we have resolved the initial counterexample. When this is the case, we continue with the algorithm by increasing k. However, when the counterexample is not yet resolved, this must mean that a predecessor state $t \in F_i$ exists, form which s is reachable ($t \wedge T \Rightarrow s$). We query the SAT solver to obtain such a state by finding a model for $F_i \wedge T \wedge s'$. We add this state to the proof obligation queue with priority i, and return to the beginning of the refining loop.

2.4.6 Increasing k

When the refining phase ends, this means that all counterexamples were refuted by strenghtening all involved frontier sets. We know that $\neg P$ is unreachable within k steps.

We now increase the value of k, and initialize the new F_k with P. However, we also retain our previously gathered knowledge about the transition system by *propagating clauses*. Recall the second invariant ($F_i \Rightarrow F_{i+1}$) , we can also express it as $Clauses(F_i) \subseteq Clauses(F_{i+1})$. This means that for all frontier sets $F_0 \dots F_{k-1}$, we must take every clause c for which $c \in F_i \wedge c \notin F_{i+1}$ holds, and add c to F_{i+1}.

Now all invariants hold, and we attempt to see whether we have found an inductive strengthening (a fixpoint, where $\exists i.F_i \wedge T \Rightarrow F_{i+1}$). This however, is evaluated syntactically in favor of semantically (by querying the SAT solver), by checking whether $\exists i.Clauses(F_i) = Clauses(F_{i+1})$. When such an inductive strenghtening is found, it is returned, and the algorithm is finished.

2.4.7 The Minimal Inductive (sub)Clause algorithm

The Minimal Inductive Clause algorithm, to which we will refer as the MIC algorithm, plays a very important role in the IC3 algorithm. The MIC algorithm takes a negated counterexample to induction (the negated cube of literals becomes a single clause), and attempts to reduce its size. The resulting clause is minimal, and inductive (reachable) to a given frontier set.

The algorithm takes as input:

- I, the initial state
- T, the transition relation
- s, a counterexample to induction
- F_i, with $i = 0 \dots k$, a frontier set for which $F \wedge T \rightarrow \neg s'$ is satisfiable

Note that it is not nessecary to implement the MIC algorithm, the negation of the counterexample is already inductive. However, not doing so effectively only removes a single state from the search. While, in many cases, the MIC algorithm will remove many more states from the search.

The simplest algorithm (after the 'algorithm' described in the previous paragraph), is documented as the *down* algorithm, which behaves as described in algorithm 1.

Informally, the *down* algorithm drops a literal from $\neg s$, tests whether $\neg s$ remains inductive, and if so, repeats the process.

The resulting clause is then minimal (no more literals can be dropped), and is used afterwards by the IC3 algorithm to strengthen the frontier sets.

2.4.8 Generating Counterexamples

A counterexample is a trace to the reachable bad state. A trace can be seen as a sequence of states. As IC3 can only deal with propositional logic formulae, 'state' refers to the CNF representation of that state.

As mentioned in section 2.4.5, elimination of a bad state s may depend on other bad states t from which s is reachable. This induces to a chain of dependent bad states. As soon as (part of) this chain is reachable from I, the chain is a counterexample trace.

Algorithm 1 The down algorithm

```
down(I,T,F_i,s){
  q = ¬s;
  reduced = false;
  do{
        reduced = false;
        //attempt to drop a literal from q
        for(i = 0...|q| − 1 && !reduced){
              l_i = q[i];
              q̂ = q \ l_i
              //test initiation
              if(!sat(I ∧ ¬q̂)){
                    //test whether q̂ is inductive
                    if(sat(F_i ∧ q̂ ∧ T ⇒ q̂′)){
                          reduced = true;
                          //restart the algorithm on q̂
                          q = q̂;
                    }
              }
        }
  }while(reduced);
  return q;
}
```

3 Implementation

3.1 Propositional Logic Formulae and CNF

IC3 relies on using logic formula for representing transition systems and properties. To use these formula internally, we designed two systems for representing propositional logic. A system to represent arbitrary propositional logic formulae, and a system to represent formulae in conjunctive normal form. Furthermore, both systems contain code to convert either system to an equisatisfiable formula in the other system.

The first system supports formulae consisting of the $\neg$,$\vee$,and $\wedge$ operators. It is implemented (by an object oriented structure) as a binary tree. It comes with convenience methods which also implement the $\rightarrow$ and $\leftrightarrow$ operators. Negation of the entire formula is also supported.

The second system represents formulae which are in CNF. A single formula is implemented as a cube, a set clauses. A clause is implemented as a set of literals. For example, the formula $(p \wedge (q \vee \neg r))$ is represented as $\{\{p\}, \{q, \neg r\}\}$.

3.1.1 Conversion

The (trivial) conversion from formulae in CNF system to formulae in PLF preserves equality ($\Leftrightarrow$). However, the conversion from the first to the second system is implemented by multiple methods.

In order to perform conversion of the two formula representation systems, the aforementioned algorithms for performing conversion to CNF had to be developed. Both the equivalence conversion and the Tseitin transormation algorithms were implemented. The literals intro-

duced by the Tseitin transformation are flagged as such, to enable the code which parses the result of the SAT solver to discard these variables.

3.2 SAT solvers

3.2.1 Logic2CNF

At the beginning of the project, no formulae representing test cases were avaiable in CNF, and in order to test the semantics of IC3, a SAT solver was needed which supported arbitrary formulae. Logic2CNF is a fork of Minisat, which supports arbitrary formulae. Internally it applies the Tseitin transform algorithm, and drops the newly introduced literals from the output. It is able to find all models for a given formula, but also supports searching up to a specified maximum.

3.2.2 SAT4J

SAT4J is a Java implementation of the Minisat SAT solver. It only supports formulae which are in CNF. SAT4J is able to find a single model of a formula, but it is also able to find all models.

Furthermore, it conserves the behaviour of Minisat to report unsatisfiablity whenever a trivially unsatisfiable formula is provided to it (e.g.: $p \wedge \neg p \Leftrightarrow \bot$). This enables the SAT solver to return a result early.

3.3 Problem solving architecture

The IC3 algorithm requires a very specific input format. To facilitate the bootstrapping process (e.g. feeding problems to the algorithm), the `ProblemSet` class was created. This class is used as a container for the formulae describing the transition system, and formulae describing (possibly multiple) properties. The class also contains a shortcut to automatically check all defined properties in the problem set.

To run the IC3 algorithm on an arbitrary transition system for a set of properties, the following steps must be performed:

1. Obtain a transition system.
2. Obtain properties to be checked on that transition system.
3. Convert the transition system and properties to a representation in propositional logic.
4. Collect the specification of the transition system in a ProblemSet.
5. Instantiate the IC3 class with a SAT solver.
6. Execute `check(ic3)` on the ProblemSet.

3.4 The IC3 algorithm

The IC3 algorithm was implemented in Java. The algorithm itself consists of a two classes of approximately 300 lines of code. In order to improve readability of the code, the IC3 algorithm is split up in multiple methods as shown below:

- `check()`, runs the IC3 algorithm on a given transition system and property.
- `findInductiveFrontier()`, finds the largest inductive frontier given a proof obligation and $F_0 \dots F_k$.
- `strenghten()`, strengthens $F_1 \dots F_{ind+1}$ given an inductive frontier , uses the MIC algorithm for more efficient strenghtening.
- `propagateClauses()`, propagates earlier discovered clauses forward
- `hasFixpoint()`, checks whether a set of frontier sets contains a fixpoint, that is, whether $Clauses(F_i) = Clauses(F_{i+1})$ for some i.
- `MIC()`, obtains a minimal inductive clause given a bad state and a frontier set from which it is not inductive.
- `down()`, tests whether a clause is inductive relative to a given frontier set [1]

3.5 Testcases

Three testcases were designed to test the IC3 implementation, the behaviour of the algorithm was compared with the described behaviour (if any).

Somenzi and Bradley provide two in-depth examples in [4].

The first example (implemented in `IC3WMIM_1.java`) provides the following parameters:

$$\begin{aligned} I =& (\neg x_1 \wedge \neg x_2) \\ T(x, x') =& (x_1 \vee \neg x_2 \vee x_2') \wedge (x_1 \vee x_2 \vee \neg x_1') \wedge (\neg x_1 \vee x_1') \wedge (\neg x_1 \vee \neg x_2') \wedge (x_2 \vee \neg x_2') \\ P =& (\neg x_1 \vee x_2) \end{aligned}$$

These formulae above describe a system which contains a bad state which is unreachable.

The second example (implemented in `IC3WMIM_2.java`) from [4] was defined differently than was shown in the provided image. The following formulae were formed according to Figure 2 in [4], which describes a transition system with a reachable bad state.

$$\begin{aligned} I =& (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \\ T(x, x') =& (\neg x_2 \vee x_1') \wedge (\neg x_3 \vee x_2') \wedge (\neg x_1' \vee x_2) \wedge (\neg x_2' \vee x_3) \\ P =& (\neg x_1 \vee \neg x_2 \vee \neg x_3) \end{aligned}$$

In order to better test the generation of counterexample traces, a third test case (implemented in `ReachableBadState.java`) was constructed, which consisted of a linear path to a bad state.

[1] The `down()` algorithm can also find smaller inductive clauses based on counterexamples to induction, this is the reason why `MIC` and `down` methods have been split up. However, this optimization has not been implemented.

$$
\begin{aligned}
I =& (\neg A \wedge \neg B) \\
T(x, x') =& (\neg A \vee A') \wedge (\neg A \vee B') \wedge (\neg B \vee B') \wedge (B \vee B') \wedge (C \vee D) \\
& \wedge (\neg B \vee \neg B' \vee A) \wedge (\neg A' \vee \neg B' \vee A) \wedge (\neg A' \vee A \vee B) \\
P =& (\neg A \vee \neg B)
\end{aligned}
$$

3.6 Encoding

Due to time limitations, and errors in the initial encoding design. The design for the documented encoding was not correctly implemented.

3.6.1 Problem structure

The RERS challenges from 2014 come in a variety of complexity. A encoding program was written to encode the problems with the lowest language complexity as propositional logic formulae representing an Initial state, a Transition relation, and a set of properties.

These problems consist of C programs which contain value assignments, comparisons using equality only, and function calls.

Every problem defines a set of possible inputs, and methods which alter the state of the program, according to the given input. In addition, every problem also defines an `errorcheck()` method, which checks for 100 possible errors using assertions. One of the goals of the challenge is to find whether these errors can occur, and what input is needed to trigger them.

3.6.2 Parser implementation

Jeroen Meijer and Jaco van der Pol participated in the RERS 2014 challenge with the LTSmin model checker. They constructed a ANTLR3 parser grammar to generate a parser for the C programs.

The grammar was first converted to ANTLR4, and all additional logic was discarded. ANTLR4 provides tree listener/visitor classes, which can be used to execute logic while walking over the generated abstract syntax tree generated by the generated parser.

3.6.3 Encoding structure

While walking the tree, the parse tree listener obtains the information needed to encode the C program for IC3.

Integers are modelled as 32-bit integers (two's complement). This means that for each variable $2 * 32$ literals are allocated (x and x'). For instance, the first problem declares 43 integers which amount to 2752 literals.

Initial state The initial state is constructed by taking all global variable declarations from the program, parsing the declared initial values, and stating that the bits of the variables are as declared. In addition, a formula is added which represent that any one (but only one) of the input variables must hold. For the first problem, this results in a formula with 1381 literals.

Transition relation Before generating the transition relation, the program first analysis the call structure of the program. The program constructs a dependency graph[2] and then generates propositional logic formulae for all functions in a bottom-up manner. The reason for this being, that formulae for function calls are inlined.

In a function, the following language elements may occur:

- `if-then` statements[3]
- `assignment` statements
- `function` calls

We also make the following assumptions on the code:

1. For each iteration of the program, a single variable is only assigned once
2. The conditions in the if-statements are mutually exclusive

Formulae from multiple statements within the same scope are concatenated using the $\wedge$ operator. This makes these statements interchangeable, which might cause incorrect behaviour, unless the aforementioned assumptions hold.

Assignment statements Before generating code for assignment statements, we need to make sure that we correctly define behaviour when variables are not assigned in a iteration. When we fail to do this, the SAT solver will happily choose a arbitrary value which will quickly lead to incorrect conclusions drawn by the IC3 algorithm. To this end, program introduces new literals for each variable. These literals denote whether a variable was changed.

At the beginning of the transition formula, we add a formula denoting that when a variable is not assigned, its next value must be equivalent to its old value. This formula is of the form:

$$\begin{aligned} T(x, x') &= \ldots \wedge (\neg c_i \rightarrow ((l_i \wedge l_i') \vee (\neg l_i \wedge \neg l_i'))) \wedge \ldots \\ &\Leftrightarrow \ldots \wedge (\neg l_i \vee l_i' \vee c_i) \wedge (l_i \vee \neg l_i' \vee c_i) \wedge \ldots \end{aligned}$$

Now, when we want to denote in our formula that some variable has been assigned, we state that its value has changed, and denote that its future literals should represent its assigned value. For example, to represent that we assign the value 'true' to the boolean variable a we use the following formula:

$$\begin{aligned} T(x, x') &= \ldots \wedge (c_a \rightarrow l_a) \wedge \ldots \\ &\Leftrightarrow \ldots \wedge (\neg c_a \vee l_a) \wedge \ldots \end{aligned}$$

If-then statements If-then statements consist of a condition and a body. If-then statements are encoded as follows:

$$T(x, x') = \ldots \wedge (\text{ condition } \rightarrow \text{ statements }) \wedge \ldots$$

[2]Here lies an assumption that all dependencies can be resolved bottom-up. This means that infinite call loops or recursive algorithms are not supported.

[3]no else statements

Conditions We assume that the conditions in the program only consist of equality checks of the form 'variable==value'. We can encode this by stating that the literals representing the variable have the corresponding two's complement encoding for the value. For example, we can encode that the boolean variable a is equal to 1 with the following formula:

$$\ldots \wedge l_a \wedge \ldots$$

Multiple conditions might be concatenated by (boolean) $\wedge$ and $\vee$ operators. We already have code which supports this, which means that we can directly encode these operators according to their definition.

Function calls We are now able to construct formulae for all mentioned statements, except for function calls. However, we assumed that the function call graph can be resolved. This means, that for every function call, it is either possible to generate code for it, or to inline the already generated code. We always have the following choices:

- Either, the function call is independent $\rightarrow$ generate a formula immediately.
- or, the function call is dependent, but formulae for all dependent functions has already been generated $\rightarrow$ inline the generated formulae.
- or, the function call is depentent, and for some dependent functions no code has yet been generated $\rightarrow$ skip this function for now, and process it when more code has been processed.

When this procedure finishes, we have generated a formula for the top-level function, and therefore have a representation for the entire program.

Properties The properties are obtained from the `errorCheck()` function. These properties consist of assertions represented by conditions. We encode these conditions as described earlier in this section, however we feed them as a property to the `ProblemSet`. Note that `errorCheck()` describes formulae for bad states, while IC3 checks whether a property always holds. To this end, we negate the formulae for the properties.

3.7 Measurements

Without a correctly functioning encoder (see section 3.6) for the RERS problems, we were not able to measure the performance of IC3 for large problems.

For the tests, all output was disabled, to only test the performance of the algorithm. At the start of each test, the main thread was put to sleep for 5 seconds [4], in order to allow the profiler and the JVM to start without influence the measurements.

Timing The timing measurements were performed by letting the algorithm run each test case n=10000 times. The program then printed the difference in milliseconds between the start and the end time.

[4] using `Thread.sleep()`

Test	P holds	Time needed	Average time per execution	Average heap size
IC3WMIM 1	yes	856 ms	85.6 μs	4.1 MB
IC3WMIM 2	no	19430 ms	1.94 ms	3.7 MB
ReachableBadState	no	18188 ms	1.82 ms	4.7 MB

Table 1: Measurement results

Memory The memory usage was measured using the VisualVM profiler. As the real memory usage cannot be measured due to the memory allocation characteristics of Java, the measured memory usage serves only as an indication. To migitate this behaviour, we explicitly call the garbage collector [5] before each call of `ic3.check()`. Given that garbage collection takes more time, we ran each test case only n=1000 times.

We used the following JVM parameters (start the JVM in server mode, with a maximum heap size of 3GB):

```
java -server -Xmx3g [filename]
```

An overview of all the measurements can be found in table 3.7.

3.8 Future Work

3.8.1 IC3

The SAT solvers might also be optimized for use with the IC3 algorithm. For instance, the IC3 algorithm often issues SAT queries which are equivalent to earlier queries. Furthermore, there is also some overlap on parts of the formulae (for example, the transition relation T will often be part of a SAT query) provided in the query.

The IC3 algorithm shows similarities with abstraction (CEGAR), symbolic (BDDs), and BMC model checking algorithms. Perhaps combining IC3 with these techniques (for instance, implementing the MIC algorithm using BDDs) will lead to an improvement in performance.

3.8.2 Implementation

We failed to implement the encoder for the RERS problems. However, the design shows that it should be possbile. Implementing this encoder will allow for measurements of IC3 for larger problems.

Hassan et al. [3] have improved the `down` algorithm. This improvement supersedes an intermediate improvement by Bradley et al. [2] which defines the *up* algorithm. The created Java implementation of the IC3 algorithm only implements part of the *down* algorithm. As the MIC algorithm is essential to the performance of IC3, implementing the improved algorithm could yield better overall performance.

Currently, our IC3 implementation, as well as our SAT solver, both run on the JVM. The configuration of this JVM directly influences the performance of the program. Not much time was spent optimizing these parameters. For example, it might be worthwhile to investigate the impact of the size of the young generation (used in garbage collection). A different approach might be to implement the algorithm in C (although an implementation is already available at `https://github.com/arbrad/IC3ref`).

[5] By invoking `System.gc()`

While Minisat is a popular SAT solver, different SAT solvers might have different performance characteristics. Also, using a compiled SAT solver using JNI will definitely improve performance.

The presented method for representing C programs as propositional logic formulae very probably is not the only existing method in existence to represent such programs. Different encodings might also result in a better performance of either the used SAT solver, or the generalisation algorithm.

3.8.3 Encodings

The C programming language can be used to solve a vast array of problems. Restricting the domain for which encodings have to be generated might allow for more general representations of problems due to knowledge on a higher level. For example checking properties on a model of Sokoban is probably more efficient in comparison to a C implementation of the game.

The current encoding algorithm already removes unreachable methods. However, unused variables (those which are only read/written in reachable code) are not removed. This will result in a smaller representation. Also, the 32 allocated literals per integer are not always used (for instance in booleans). Detecting these boundaries might decrease the representation size and the total amount of literals used.

3.9 Conclusion

We were able to construct a working implementation of the IC3 algorithm in Java, supporting multiple SAT solvers. We also designed an encoding scheme in order to verify RERS problems with the IC3 algorithm. Unfortunately there was not enough time left to correctly implement the RERS problem encoding scheme and test the algorithm with a larger test case.

References

[1] Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[2] Aaron R Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 173–180. IEEE, 2007.

[3] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. Better generalization in ic3. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 157–164. IEEE, 2013.

[4] Fabio Somenzi and Aaron R Bradley. Ic3: where monolithic and incremental meet. In *FMCAD*, pages 3–8, 2011.

[5] Wikipedia. Tseitin transformation. `http://en.wikipedia.org/wiki/Tseitin_transformation#Gate_Sub-expressions`, April 2015.

A Compiling and running the project

The program must be compiled with the Java JDK, with a version ≥ 8. The program has the following dependencies (all included in the lib folder):

- SAT4J
- Antlr4
- Lombok

The code can be built with the provided ant script. This is done by issuing the following command from the root directory of the project:

```
ant build
```

To create a runnable .jar file, run the following command from the root directory of the project:

```
ant jar
```

It is also possible to run the testcases using the following command:

```
ant run
```