APPLYING SMT SOLVING AND WEAKEST-PRECONDITION REASONING TO CODE REACHABILITY ANALYSIS

DAVID BERGVELT

Abstract. The question of whether or not a section of code is reachable is relevant to numerous problems within software engineering, verification, and testing. Though determining code reachability by simple observation may be possible in small programs, mechanized techniques become necessary as the size and complexity of programs grow. In this paper, we examine one such technique known as weakest-precondition reasoning, a method of searching backwards through control flow graph representations of programs to determine the minimal set of conditions necessary to reach a given section of code. More specifically, we describe how weakest-precondition reasoning may be applied to the sequential reachability problems of the Rigorous Examination of Reactive Systems (RERS) challenge, exploring several different implementations of this technique. We show that by augmenting a weakest-precondition search with an SMT solver to perform simplification and model-finding, it is possible to determine code reachability at speeds comparable or even superior to other existing methods.

1. Introduction

In today's increasingly software-dependent world, it is more important than ever to verify that programs behave as their authors intended. One aspect of program correctness is code reachability, in particular the expected conditions under which a segment of code will be executed. For security and stability reasons, it is important for programmers to fully understand the ways in which their code may be executed. If there exists a method by which some section of code may be run in a way unintended by its authors, this represents a bug that may have serious consequences, especially for security or safety-critical applications. Code reachability is also relevant to software testing, where it may be applied to optimize the testing process after modifying a codebase by only running tests which execute modified sections of code.

Due to the size and complexity of modern software systems, code reachability can be a difficult problem to solve. Fortunately, there exist numerous strategies by which we can automate the process of determining reachability. One of these strategies is *weakest-precondition reasoning*, a method introduced by Dijkstra [4] based on Hoare logic [6], which allows us to calculate the minimum set of logical conditions necessary to execute a particular sequence of code. This set of conditions, known as the *weakest precondition*, can be used not only to derive whether or not a piece of code is reachable, but can also tell us precisely which circumstances (if any) cause the code to be reached. In the nontrivial cases (i.e. where the weakest precondition is not a tautology or contradiction), we are presented with a satisfiability problem, namely determining if there exists a mapping from the space of possible program variable values to the free variables in the weakest precondition such that the weakest precondition evaluates to True. Such a satisfiability problem may be solved by a *satisfiability modulo theories (SMT) solver* [2]. If no mapping exists that makes the weakest precondition evaluate to True, the code is not reachable. However, if such a mapping does exist, we can view this mapping as a test vector that causes our section of code to be reached.

1.1. **The Rigorous Examination of Reactive Systems (RERS) Challenge.**

In this paper, we will examine how weakest-precondition reasoning may be applied to the sequential reachability problems of the RERS challenge. Each of these problems is presented as a C source file containing five general sections, as seen in Figure 1.

First, a collection of global variables are declared and initialized. Next, we enter the main loop of the program. The first step of this loop is to read an input character from the command line, and check that this character is a valid input (i.e. that it is a member of the alphabet of accepted inputs for this particular problem). Next, a series of highly obfuscated calculations is performed, modifying the global variables which were initialized in the first step. Finally, depending on the state of the global variables after these calculations are complete, the program will either loop back to accept another input, or trigger one of 100 possible error states. The goal is to determine whether or not each of these 100 error states is reachable, and to identify which sequence of input characters will trigger each reachable error state.

Date: November 13, 2017.

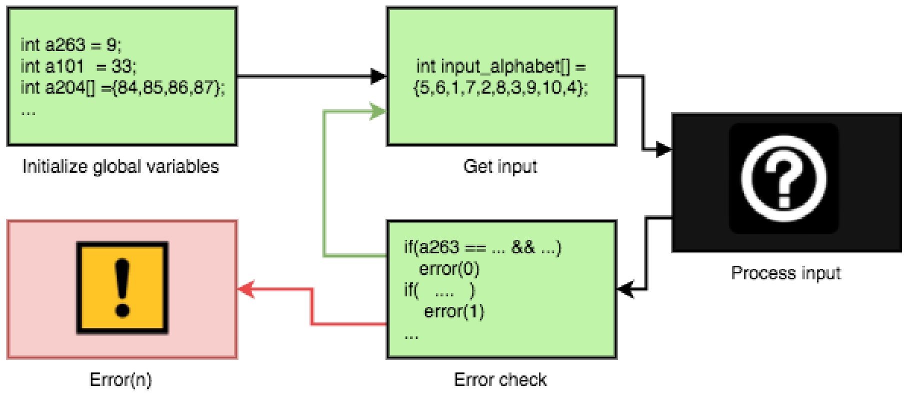

FIGURE 1. Overview of a RERS reachability problem

2. GRAPH CONSTRUCTION

In order to perform weakest-preconditioning analysis on a program, an abstract representation of that program is needed. Specifically, we would like an abstract representation that captures the structure of the represented program in such a way that we can easily mimic execution traces by moving forwards and backwards through sections of code —including across function calls, returns, loops, and conditional branches (this will help us efficiently use the sequencing rule when calculating the weakest precondition, and will be elaborated on in the next section). We will use a control flow graph for this purpose [1, Chapter 9] [9, Section 3.2.2] [5].

The construction of a control flow graph from a source file is performed in two phases. First, the source file is lexed and parsed into an abstract syntax tree. This phase is done using the OCamllex lexer and OCamlyacc parser along with a simplified context-free grammar for C, modified from Jeff Lee's 1985 Yacc grammar for C to only include the syntactic elements found in the RERS problems (e.g. for loops cannot be parsed) [7]. The result of this first phase is an abstract syntax tree describing the syntactic structure of the target program.

During the second phase, the abstract syntax tree is translated into a control flow graph. In a nutshell, this translation is done by traversing the abstract syntax tree and generating a node for every program instruction encountered, connecting those nodes with edges according to the type of instructions they represent. The correspondence between program instructions and control flow graph nodes can be seen in the definition for the node type seen in Figure 2: an assignment instruction becomes an assignment node, an if or if/else instruction becomes a conditional node, and so on. Especially important are *terminal nodes*, which represent calls to the function triggering the various error states whose reachability must be determined. It is these terminal nodes that will be the starting point for our weakest-precondition search.

3. WEAKEST-PRECONDITION SEARCHING

One of the central facets of weakest-precondition reasoning is its basis in Hoare logic [4]. Using a Hoare triple of the form $\{P\}S\{R\}$, where P represents a precondition, S represents a program statement, and R represents a postcondition [6], we can introduce a weakest-precondition function wp with the following property:

$$\forall P.\{P\}S\{R\} \Rightarrow \Big(P \Rightarrow wp(S,R) \wedge \{wp(S,R)\}S\{R\}\Big)$$

Taking a program statement and postcondition as arguments, the function wp attempts to find a precondition satisfying the Hoare triple containing that same statement and postcondition given as arguments. In addition, because it is the *weakest* precondition, the result of wp should be logically implied from any other precondition satisfying the Hoare triple.

The behavior of wp is defined through the following rules[1]:

(Skip Rule) $$wp(\mathbf{skip}, R) = R$$

(Sequencing Rule) $$wp\,(S_1; S_2, R) = wp\,\Big(S_1, wp\,(S_2, R)\Big)$$

[1]It should be noted that while there exist additional rules for the behavior of wp which allow for reasoning about other program statement types (such as loops), only these four rules are strictly necessary for reasoning about the RERS problems.

```
type node =
  Begin_node of {out_node : int * edge_label}
| Assign_node of
   {var : Abs_syn.var; asg_value : Abs_syn.exp list;
     node_num : int; out_node : int * edge_label;
      in_nodes : (int * edge_label) list}
| Cond_node of
   {cond : Abs_syn.exp; node_num : int;
     then_node : (int * edge_label); else_node : int * edge_label;
      in_nodes : (int * edge_label) list}
| Output_node of
   {output_val : Abs_syn.exp; node_num : int;
     out_node : int * edge_label;
      in_nodes : (int * edge_label) list}
| Input_node of
   {node_num:int; out_node:int * edge_label;
       in_nodes : (int * edge_label) list}
| Error_msg_node of
   {node_num :int; out_node : int * edge_label;
      in_nodes : (int * edge_label) list}
| Terminal_node of
   {crash_val : int; node_num :int;
      in_nodes : (int * edge_label) list}
| Function_exit_node of
  {node_num : int;  out_nodes : (int * edge_label) list;
      in_nodes : (int * edge_label) list}
| Call_node of
  {node_num : int; calling : string; out_node : int * edge_label;
     in_nodes : (int * edge_label) list}
| Dummy_node of {node_num : int;  in_nodes : (int * edge_label) list}
```

FIGURE 2. Node type definition

```
type edge_label =
  Seq
| Yes
| No
| CallFrom of string
| ReturnTo of string
```

FIGURE 3. Edge type definition

(Assignment Rule) $$wp(x := E, R) = R[x \leftarrow E]$$

(Conditional Rule) $$wp(\textbf{if}\, E\, \textbf{then}\, S_1\, \textbf{else}\, S_2, R) = \Big(E \Rightarrow wp(S_1, R)\Big) \wedge \Big(\neg E \Rightarrow wp(S_2, R)\Big)$$

3.1. **Implementation.** The four rules above may be applied to implement an algorithm that searches through control flow graphs to determine code reachability. Due to the backwards-facing nature of weakest-precondition reasoning, this algorithm will search through control flow graphs in reverse, beginning from the node[2] representing the code whose reachability is to be determined. In addition, the initial postcondition is set to True, indicating a contradiction-free state with no constraints.

Each backwards step in the search represents an application of the Sequencing Rule, progressively nesting applications of *wp*. Upon reaching an assignment or conditional node, the corresponding rules are applied. However,

[2]In the case of the RERS problems, a terminal node.

because any conditional node will be encountered via one its branches, the Conditional Rule may be simplified to $wp(\textbf{if}\, E\, \textbf{then}\, S_1\, \textbf{else}\, S_2, R) = E \wedge wp(S_1, R)$ when approaching via the True branch, and $wp(\textbf{if}\, E\, \textbf{then}\, S_1\, \textbf{else}\, S_2, R) = \neg E \wedge wp(S_2, R)$ when approaching via the False branch. All other types of instructions encountered within the RERS problems are simply handled via the Skip Rule because they do not affect weakest-precondition calculations.

While the above four rules form the backbone of the search algorithm, additional behavior must be specified in order to make the algorithm useful for determining code reachability. First, it is necessary to define what should be done when encountering (reverse) branch points in a control flow graph, such as the first node in a function called at multiple places elsewhere in the program. This problem is not difficult to solve at a theoretical level: simply search every possible path. However, actually implementing this strategy poses a number of challenges, especially when confronted with the vast, labyrinthine control flow graphs representing the RERS problems.

When performing a branching search, some kind of data structure is needed to keep track of the state of various search subtasks exploring different branches. This data structure would need to store information about each active subtask, such their current location in the graph, as well as the weakest-precondition they have each calculated thus far. The primary implementation concern here is space efficiency: a large number of branching points implies a large number of subtasks to track, and lengthy search traces imply large weakest-preconditions. Both of these issues arise in the RERS problems, leading to a large number of search subtasks that each consume a significant amount of memory to track their progress. After some experimentation, a depth-first search strategy (using a stack to keep track of subtasks) was chosen due to the lower average number of subtasks active at any given time.

Why is it that a depth-first search implies fewer active search subtasks? This brings us to another aspect of our algorithm's behavior that must be specified: determining success or failure. Let us first approach this at the level of an individual search subtask. Consider the following example: a subtask T with current weakest precondition P_T has just reached a conditional node with condition $1 > 2$ via the True branch. Clearly, it should not be possible for the True branch of this conditional branch to be taken. Accordingly, when we apply our simplified Conditional Rule we get a weakest precondition of $(1 > 2) \wedge P_T$, a contradiction. This example illustrates the criterion for failure at the level of individual subtasks: if the application of a weakest-precondition rule results in a logical contradiction for a given subtask, then the start node of the search cannot be reached from the current node via the path taken by the subtask in question. Conversely, a successful subtask is one that reaches the head of the control flow graph with a contradiction-free weakest precondition.

An important property of contradictory weakest preconditions is that they imply that the start node of the search cannot be reached via any paths having the contradictory path (i.e. the path corresponding to the contradictory weakest precondition) as a subpath. As a result, further searching along the a path is unnecessary after a contradiction has been found, and the search subtask which the contradictory weakest precondition belongs to may be discarded. This is why a depth-first strategy is quite memory-efficient for this application. Each subtask must be pursued until it terminates, which may happen either by discovering a contradiction, or successfully reaching the head of the control flow graph. Even though new subtasks must still be spawned and placed on the stack whenever branching points are crossed, the number of active subtasks tends to be far smaller when using a depth-first strategy than when using a breadth-first strategy, since the current search subtask must terminate before any others may be worked on (thereby spawning more subtasks, which themselves spawn more subtasks, and so on).

Now that we know how to handle branching and determine the success or failure of a search subtask, we are faced with a larger and more abstract question: how do we obtain the reachability information we are looking for for a given terminal node in a RERS problem? There are actually two problems to be solved here. First, we must determine whether the target terminal node is reachable at all. This requires finding at least one non-contradictory path from the target node back to the head of the control flow graph. Thus, the criterion for a successful search task is the same as that for a successful search subtask because a successful search subtask is by definition associated with a non-contradictory path between the target node and graph head. Conversely, a search task can only fail (thereby determining that a node is unreachable) if every path backwards from the target node eventually[3] results in a contradiction.

The second problem that must be solved in order to obtain the reachability information required to solve the RERS problems is that of finding input vectors. More specifically, if we know that a terminal node is reachable, how do we find the sequence of inputs necessary to reach that node? In the next section, we will examine and compare several possible solutions to this problem.

[3]Note that due to the Halting Problem it is impossible to generally prove that search subtasks will ever terminate. A possible workaround is to declare a maximal "depth" which subtasks may search to before being artificially terminated. However, this metric must be carefully chosen: deep searches tend to be slower, while shallow searches may be cut off before producing meaningful results.

3.2. **Strategies for Finding Input Vectors.** Within the `main` function of each RERS problem exists a program instruction which reads a single character from the standard input. Correspondingly, each control flow graph representation of a RERS problem contains a so-called *input node*. An important step towards finding an input vector for a reachable node is figuring out how search subtasks should handle reaching the input node.

Essentially, the input node may be viewed as an assignment node. This should not be surprising, since the input instruction within `main` in each problem does in fact perform an assignment, namely assigning the character read from `stdin` to a variable `input`. However, it is distinct from other assignment statements in that the value being assigned is external, and independent of program variables. While the values on the right hand side of regular assignment statements can always be immediately evaluated during program execution (due to either being constants or expressions with data dependencies on existing variables), the value assigned to `input` comes purely from outside the program. We can treat this value as a special variable called $input_n$, where n is the number of times[4] the current subtask has passed the input node.

Though the independence of the value assigned to `input` may seem like a trivial point, it has an important implication in our weakest-precondition search: any search subtask that reaches the head of a control flow graph cannot contain any free variables other than $input_n$ for various values of n. Regular variables must be declared and initialized before they may be used, so any program variables which show up in a subtask precondition as it searches the graph will eventually be substituted away via the Assignment Rule when the subtask reaches the node where each variable is first initialized (which will naturally occur before reaching the head of the control flow graph). The variable `input`, in contrast, will only be substituted for other variables of the form $input_n$, meaning that any subtask that reaches the graph head may only have a weakest precondition that is an expression containing constants and one or more instances of $input_n$ (and no other variables). An input vector is found by identifying which value can be substituted for each $input_n$ such that the weakest-precondition evaluates to True.

3.2.1. *Substitution Search.* A naive strategy for identifying an input vector from a weakest-precondition P containing instances of $input_n$ is to simply generate every possible input sequence of length n and then evaluate P for each possible input sequence by substituting each $input_n$ in P for the corresponding character in the input sequence being tested. While this strategy will theoretically produce accurate results, it is highly inefficient in practice, becoming prohibitively slow when applied with large n values.

3.2.2. *SMT Solver Application.* A better strategy can be implemented by first observing that the process of identifying values for $input_n$ that cause P to evaluate to True is a satisfiability problem, and is therefore an excellent target for an SMT solver.

An SMT solver is a tool that searches for values that can be assigned to the free variables in a logical expression so that the expression evaluates to True (and is thereby *satisfied*). Unlike a SAT solver, which can only generate boolean values to solve satisfiability problems in propositional logic, an SMT solver may search within the domains of any first-order theories supported by the solver. In the context of this project, the theory of integer arithmetic is necessary to reason about weakest preconditions containing the integer program variables found within the RERS problems.

An important feature of many SMT solvers is their preservation of an internal state that is updated with each subsequent logical formula they are presented with. This continuously evolving state can be exploited for large performance gains in our weakest-precondition algorithm, namely by using the solver state as our "in-progress" weakest precondition for a search subtask, with the solver acting as a simplification engine. OCamlyices, the primary SMT solver used in this project, maintains a stack of internal states that grows with each new formula it is given [3]. If a newly applied formula causes the solver to enter a contradictory state, then the state stack may simply be popped to return to the state prior to when the contradiction-triggering formula was applied.

This has a direct correspondence to the stack structure used to manage search subtasks. This correspondence allows us to store only one weakest precondition in memory at a time, instead of storing one for each search subtask, a huge reduction in memory usage (from $O(n)$ for n active subtasks to $O(1)$). A single global solver state is updated for each step taken by a search task, increasing its stack size by one at every step. In the event that a contradiction is found, the state stack is popped k times, where k is the number of nodes in the path between where the contradiction occurred, and the last branch point passed by the (now contradictory) subtask. After being popped k times, the top of the solver state stack will be precisely the "in-progress" weakest precondition for the search subtask at the top of the subtask stack. This follows because a depth-first search will push additional tasks onto the task stack whenever a branching point is encountered, but only work on those tasks when the tasks above them in the stack have terminated and popped. In this case, if we have a contradiction that occurs after k new solver states have been pushed onto the solver stack since the last branch point in the graph, we will require those k solver states to be popped in order

[4]Because `input` is assigned a new value with each pass of the while loop in `main`, we must distinguish instances of `input` by which loop iteration they are encountered in.

to correctly begin searching from that last branch point, the starting location of the search subtask on top of the subtask stack.

Now that we have seen the advantages an SMT solver can bring to the space complexity of our search algorithm, what about the time complexity? To understand the speed benefits gained by using an SMT solver, we will first need to understand the factors that make our other implementation strategy, the substitution search, relatively inefficient.

Recall that any search subtask that generates a contradiction by applying a weakest-precondition rule can be discarded since any further search along the current path would be pointless. In the interest of minimizing unnecessary work, it is important to spot these contradictions as soon as they occur. However, this is not always an easy task. Contradictions may thought of as belonging to various "classes" according to how difficult they are to identify. For example, a formula $(i > 1) \wedge \neg(i > 1)$ can be seen as a contradiction with no knowledge of integer arithmetic, as it is an instance of a contradiction in propositional logic of the form $A \wedge \neg A$ for some boolean A. However, a formula $(i > 1) \wedge (i < 1)$ can only be shown to be a contradiction by proving that there exists no integer value for i satisfying the formula. The first example belongs to a large class of formulae that can all be immediately identified as contradictions because they match the pattern of the contradiction $A \wedge \neg A$. The second example is much more difficult, and cannot be generally identified as a contradiction by pattern matching.

The method employed by the substitution search implementation to identify contradictions was based on one used by Gunter and Peled in their Path Exploration Tool (PET) [5], using pattern matching for various arithmetic identities to perform simplification of logical formulae. While this method was faster for simplifying "nice" formulae with easy-to-spot contradictions, it was often unable to identify more insidious contradictions, leading to instances of contradictory formulae that were not spotted until well after the contradictory weakest-precondition calculation step was performed.

This problem can be solved by using the powerful simplification capabilities of an SMT solver. Despite being known for being computationally expensive, the time saved by immediately spotting contradictions and avoiding unnecessary further searching appears to far outweigh the time added by using the SMT solver as a simplification engine. Indeed, applying an SMT solver to our depth-first search algorithm yielded surprisingly low times during testing, correctly identifying the reachability (and necessary input sequences) for every terminal node for the RERS 2016 Problem 10 in only 26 seconds[5] (down[6] from the 2 minutes and 39 seconds of Smetsers et al. [8]).

4. Conclusion and Future Work

By applying weakest-precondition reasoning to the RERS reachability problems, it is possible to efficiently determine the reachability of arbitrary sections of code. This efficiency is due in no small part to the application of a state-based SMT solver, providing benefits in the domains of both space and time complexity.

Although this project serves as proof that weakest-precondition reasoning can be a valid strategy for reachability analysis, it still has significant limitations. First, it is only capable to solving RERS problems containing linear arithmetic. Because OCamlyices does not accept formulae containing nonlinear arithmetic, we cannot solve any RERS problems that include the multiplication or division of two variables. Although there exists the workaround of simply using a different SMT solver which does support nonlinear arithmetic, this will likely slow down our search algorithm due to the more sophisticated simplification being done by the solver. However, it is still entirely possible that this drop in efficiency will not be significant enough to render our method infeasible, and thus deserves further investigation.

Another limitation of this project is that it does not support all C language features, and therefore cannot be used to generally determine reachability for C programs. Reachability analysis can be extremely useful in many circumstances, so being able to generally determine code reachability for C programs could potentially make this project into a practical tool for many individuals involved with software engineering, verification, or security. Before that point is reached, work will need to be done to add in the C language features currently omitted, such as loops and arrays. As with supporting nonlinear arithmetic, supporting all language features of C may cause a drop in efficiency, but it is not obvious that this drop would make the algorithm unusable.

Altogether, this project demonstrates that combining weakest-precondition reasoning with SMT solving is a viable strategy for reachability analysis on a subset of C programs. Moreover, the efficiency displayed by this strategy in initial testing indicates a multitude of possibilities for productive future work, ranging from expanding the capabilities of the search algorithm in order to solve the remainder of the RERS problems, to using the project as the basis for a full general-purpose reachability analysis tool.

[5]Testing performed on a 2015 Macbook Pro with a 2.7 GHz Intel Core i5 CPU.

[6]This comparison not extremely meaningful due to the different machines used in each test. However, it does indicate that further, more formal testing may be warranted to compare these different methods of solving the RERS problems.

5. Acknowledgements

Partial support for this project was provided by Dr. Sicco Verwer for allowing this slightly unconventional independent project for his Testing and Reverse Engineering class at TU Delft, as well as Dr. Elsa Gunter who contributed general guidance and specific background knowledge in the area of control flow graphs [5].

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[2] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.
[3] Mickal Delahaye. Ocamlyices. `https://github.com/polazarus/ocamlyices`, 2013.
[4] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.
[5] Elsa L. Gunter and Doron Peled. Path exploration tool. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 405–419, London, UK, UK, 1999. Springer-Verlag.
[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.
[7] Jeff Lee. ANSI C Yacc grammar. `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`, 1985.
[8] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. Complementing model learning with mutation-based fuzzing. *CoRR*, abs/1611.02429, 2016.
[9] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

University of Illinois at Urbana-Champaign, USA
E-mail address: `bergvel1@illinois.edu`