Get Me Here: Using Verification Tools to Answer Developer Questions

Microsoft Research Technical Report 2014-10

Mike Barnett, Robert DeLine, Akash Lal, Shaz Qadeer
{ mbarnett, rdeline, akashl, qadeer }@microsoft.com

ABSTRACT

While working developers often struggle to answer reachability questions (e.g. How can execution reach this line of code? How can execution get into this state?), the research community has created analysis and verification technologies whose purpose is systematic exploration of program execution. In this paper, we show the feasibility of using verification tools to create a query engine that automatically answers certain kinds of reachability questions. For a simple query, a developer invokes the "Get Me Here" command on a line of code. Our tool uses an SMT-based static analysis to search for an execution that reaches that line of code. If the line is reachable, the tool visualizes the trace using a Code Bubbles representation to show the methods invoked, the lines executed within the methods and the values of variables. The Get Me Here tool also supports more complex queries where the user specifies a start point, intermediate points, and an end point, each of which can specify a predicate over the program's state at that point. We evaluate the tool on a set of three benchmark programs. We compare the performance of the tool with professional developers answering the same reachability questions. We conclude that the tool has sufficient accuracy, robustness and performance for future testing with professional users.

1. INTRODUCTION

Developers often struggle to answer reachability questions such as *How can execution reach this line of code?* or *How can execution get into this state?* LaToza and Myers recently documented this problem with a series of studies. In a lab study, they found half the erroneous changes that participants made during development tasks were due to misunderstandings about reachability. In a survey, they found 82% of 460 professional developers rated reachability questions as at least somewhat difficult. Their observational field study of 17 developers found 9 of the 10 longest development tasks primarily involved seeking information about reachability [10]. Hence, looking for answers to reachability questions can be both time-consuming and error-prone. This study expands the results of previous studies on information seeking. Ko, DeLine, and Venolia found that among the information needs they observed in a field study, the ones that were hardest and slowest to satisfy included *What code caused this program state?* (61% unsatisfied, up to 21 minutes seeking answers) and *In what situations does this failure occur?* (41% unsatisfied, up to 49 minutes) [6]. Similarly, Sillito, Murphy and De Volder's study of information needs found developers struggling with such questions as *When during execution is this method called?*, *Which execution path is being taken in this case?*, and *Under what circumstances is this method called or exception thrown?*[14]

As developers have struggled to answer reachability questions, the research community has invented many technologies, like symbolic execution and model checking, whose purpose is the systematic exploration of program execution. To date, these technologies have been typically used for program verification, namely proving that a program P upholds a property ϕ ($P \models \phi$) or producing an execution trace showing where the property is violated. Instead, in this paper, we repurpose existing verification tools as a *query engine*: given a program P and an execution query Q, we produce an execution trace that is consistent with Q.

Our query engine, Get Me Here, is named after the simplest type of query it supports. As an example of our tool in use, suppose that a user is inspecting a stack class used to store operands in a desktop calculator. She wonders whether there are any circumstances in which the program can attempt to pop an empty stack. To find out, she invokes the "Get Me Here" menu item on the line of code that throws an exception on an empty stack (Figure 1). Get Me Here then responds by displaying a visualization of the execution trace that ends in the `throw` statement (Figure 2).

Our visualization is based on the Code Bubbles paradigm [3]. Each method call in the trace is shown in its own code bubble, with arrows showing the method calls, green text highlighting the executed code and code annotations showing the values of variables. When the user looks at the lower right corner of this trace, she sees a call to `Pop` where its stack data structure (`_dataStack`) is empty and therefore the `throw` statement is executed. She looks left to the immediate caller `ParseTerm` and sees that the stack has two operands when parsing the modulus operator (`case "%":`). However, the code that follows mistakenly pops three operands. While this example happens to reveal a bug in

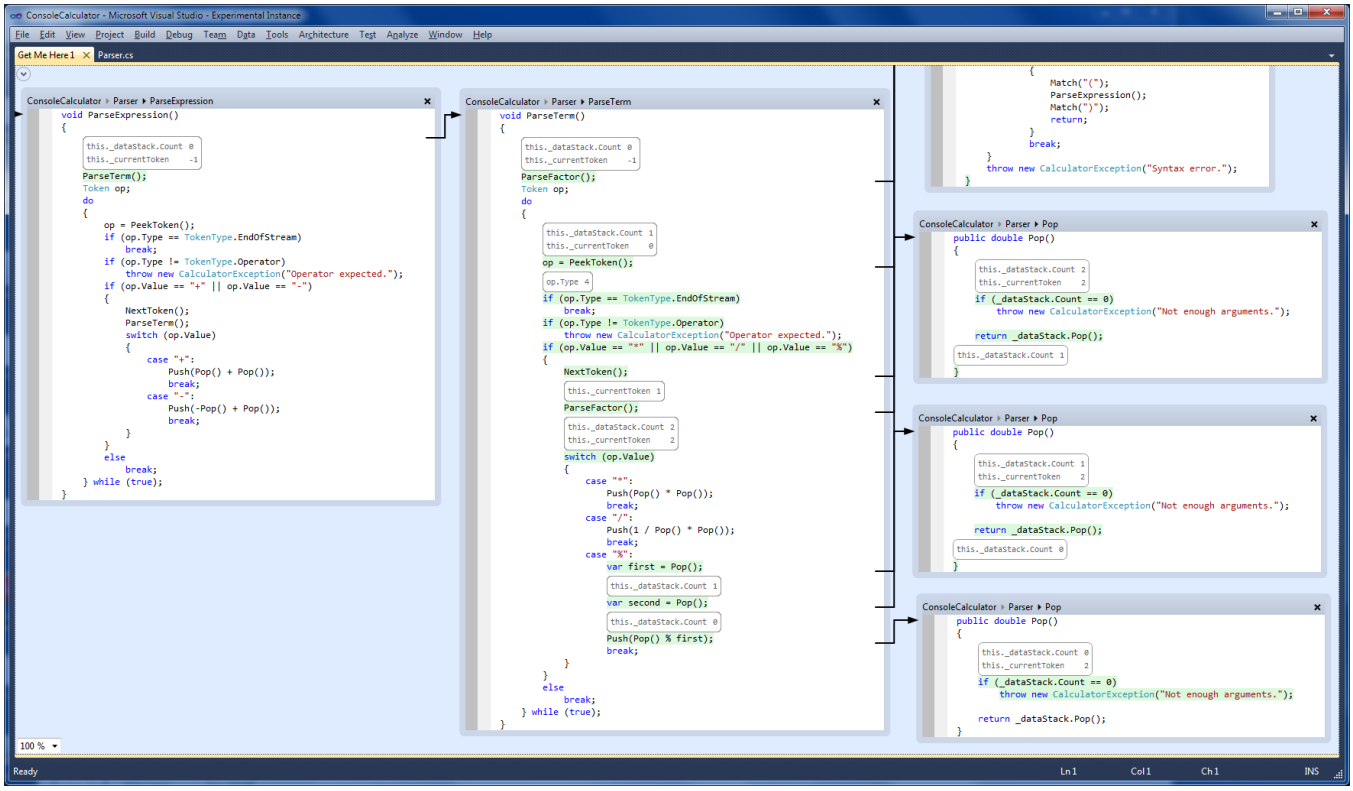

Figure 2: Get Me Here displays an execution trace, using a code bubble for each called method, arrows for method calls, green highlights for executed code, and code annotations to show the values of variables.

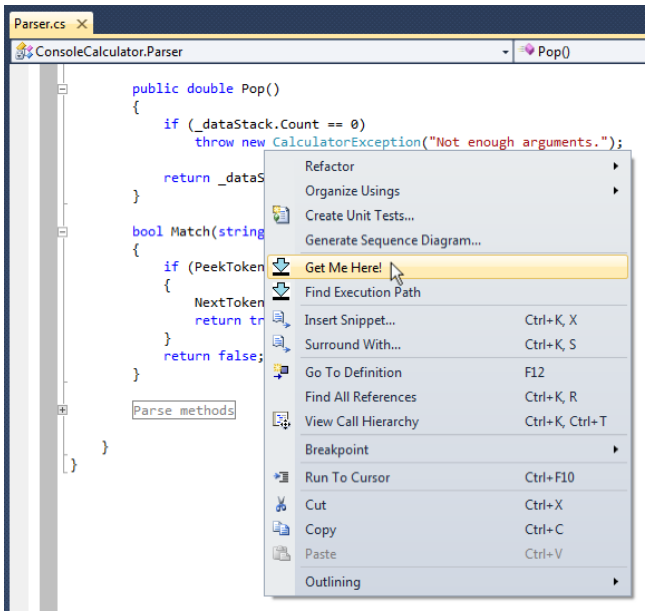

Figure 1: Invoking Get Me Here on a line of code.

the code, Get Me Here is a general query engine that searches for specified execution traces, whether the user's motivation is fault finding, feature localization, or program comprehension.

The main contribution of this paper is the demonstration that verification techniques, in particular an SMT-based static analysis, can be used as an online query engine in a development environment to answer reachability questions. In Section 2, we compare this approach with existing techniques. In Section 3, we describe the Get Me Here tool in detail, including the full query language it supports and how the user can change the modeling of the environment to trade off accuracy for speed. In Section 4, we evaluate Get Me Here on a set of benchmark programs. Using a lab study of 12 professional developers, we demonstrate that Get Me Here is capable of answering reachability questions that people struggle to answer. We also show that Get Me Here is capable of accurately and robustly answering arbitrary queries about these benchmarks. Section 5 concludes and describes a few further ways that verification tools could enhance development environments.

2. RELATED WORK

Get Me Here is most similar to LaToza and Myers's Reacher search tool [11]. Current development environments, like Microsoft Visual Studio and Eclipse, allow the user to traverse a program's call graph, either through hypertext navigation commands (e.g., *go to definition*) or through tool windows presenting hierarchical displays of caller/callee re-

lationships. A developer can use these features to answer reachability questions by exploring the call graph one call at a time. Reacher alleviates this tedium by allowing the user to issue a search for features, like method calls or field assignments, that are "upstream" or "downstream" of a given method. The user can then add search results to a central call-graph visualization.

Both Reacher and Get Me Here provide automated answers to reachability questions, but the overall user experience is different. Reacher's goal is to allow the user to get the "big picture" of reachability relationships in the program's call graph. Hence, Reacher supports iteratively searching and browsing to build up a visual graph representation that shows many relationships simultaneously. In contrast, Get Me Here's goal is to allow the user to find a single program execution of interest, by iteratively providing more constrained queries. These differing goals have implications in key design choices, including the query languages (search terms versus trace constraints), the visualizations (call graph versus execution trace), and the static analysis technology used (modular interprocedural analysis with summaries versus whole-program, SMT-based analysis).

Get Me Here can also be described as an interrogative debugging tool, in the style of Ko and Myers's WhyLine [7]. The WhyLine and Get Me Here, however, provide dual functionality. With the WhyLine, the user exhibits an execution of interest by running the program and providing the necessary input. Then, given this concrete execution trace, a user asks a *why* or *why not* query, and the tool automatically shows the relevant parts of the code to demonstrate (or contradict) the user's query. In contrast, with Get Me Here, the user forms a query about parts of the code, and the tool automatically generates an execution trace, including any relevant input necessary to reach those parts of the code.

Get Me Here has the same motivation as program slicing [16], namely to use static analysis to reduce a program's complexity to aid program comprehension. The main difference is that a static slicing technique produces a simpler program that shares a well-defined *set* of behaviors with the original program; whereas, Get Me Here focuses on a *single* program behavior (execution trace). Focusing on a single behavior allows Get Me Here to present more details, for example, the values of program variables, at the cost of possibly excluding behaviors of interest to the user. (We discuss the iterative use of Get Me Here to focus on behaviors of interest in Section 3.3.) Get Me Here has less in common with dynamic slicing techniques [15] and fault localization algorithms like Delta Debugging [17], which start with the assumption that the user has a set of interesting execution traces (for example, those that pass and fail tests) and wants to use them to find relevant code. Get Me Here addresses the dual problem of synthesizing execution traces based on parts of the code that are of interest.

3. IMPLEMENTATION

Get Me Here is implemented as an extension to Microsoft Visual Studio and uses several existing verification technologies (Figure 3). It is implemented only for .NET languages, i.e., object-oriented managed languages that compile to MSIL (byte-code) such as C# and Visual Basic. First we describe the back end that answers the user's query (the downward arrows in the figure). The final step is submitting a first-order formula to an SMT solver, Z3 [4]. Then we explain how the results are presented to the user (the upward arrows). Finally, we explore further details about more advanced queries and how environment models are built.

3.1 Query Engine

Get Me Here uses a whole-program analysis, so it runs as a post-build step after the user compiles their program. Each compilation results in a set of .NET assemblies which contain the program's metadata and bytecode. We use a byte-code translator (BCT) [2] to translate the assemblies into an intermediate language called Boogie [1, 12, 13]. The use of an intermediate language makes the translation easier than directly producing the input language of the underlying theorem prover. In addition, there are many tools, including our assertion injector (described below), that use Boogie's object model to effect Boogie-to-Boogie transformations. Boogie is intended for verification rather than execution. It is not object-oriented. Instead of classes containing (instance) methods, a Boogie program comprises a set of global procedures where the implicit instance reference ("this") becomes an explicit parameter. The type system is very simple, encompassing boolean and integer values and maps (arrays with an arbitrarily-typed domain). Each procedure consists of the usual kinds of program constructs: assignment statements, conditionals, and loops. Everything else is encoded as user-defined datatypes, functions, and axioms. BCT encodes source-language constructs such as dynamic dispatch, events, and delegates as well as a model of the heap and the object type hierarchy.

Whenever the user builds the program, Get Me Here runs a post-build step that uses a byte-code translator (BCT) to translate the program's compiled .NET assemblies into an intermediate language called Boogie [1, 12, 13]. Boogie is intended for verification rather than execution and relies on constructs like `assert`, `assume` and nondeterminism for explicitly modeling source-language constructs like dynamic dispatch, events, and control-flow constructs.

The Boogie program is then modified each time the user issues a query. The Assertion Injector produces a new Boogie program containing a compiled version of the query. For a query with a single, condition-less end point, Assertion Injector places the statement

```
assert false;
```

at the line of code the user wants to reach. (We discuss more complex queries in section 3.3.) All previously existing assertions are turned into assumptions. An assumption places a logical constraint on a program execution: the theorem prover adds it to the list of facts that hold on the trace. Contradictory assumptions prune the search space by making a particular execution trace infeasible.

We illustrate the process with the synthetic fragment of code shown in Figure 4. The example is meant to illustrate several things. The method `Read`, which reads a character from the console and returns it or -1 if there is nothing to be read, is part of the .NET Framework: it is a library method that is *not* considered part of the user's program. Trans-

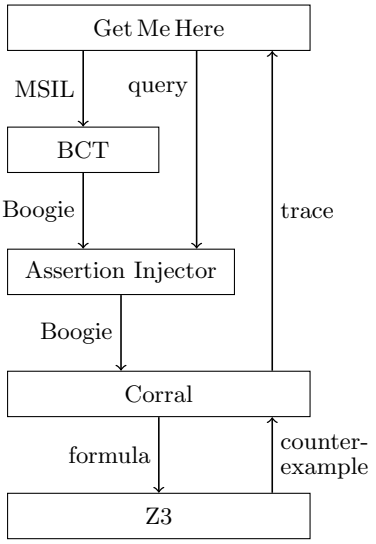

Figure 3: Get Me Here architecture

```
1  obj.x = Console.Read();
2  var y = Console.Read();
3  this.x++;
4  if (obj.x > y) {
5    if (obj.x == 10)
6      Console.Writeline("Get Me Here!");
7    // ... more code ...
```

Figure 4: A small fragment of C# code.

lating the entire framework into Boogie is both impractical and unreasonable: we want to explore the user's program and not get lost in the intricacies of the Framework's implementation. In terms of the analysis, this means that the value returned from the method is an arbitrary integer (since that is the type of the return value). If that is insufficient, then the user may write a *stub*, an environment model, that gives the semantics of the method call. This is explained in further detail in Section 3.4.

The Boogie program produced for the example is shown in Figure 5. Boogie does support high-level source constructs such as conditionals and loops, but we generate instead the de-sugared form that uses just assignment, procedure call, and goto statements. The control flow induced by the conditional statements is encoded using assume statements; Boogie's goto statements are non-deterministic. They take a list of labels and may branch to any of the target blocks. Fields are represented as maps whose domain are object references. Their range is the type of the field. So the value of the map $C.x$ is some integer for any object, even if its type is not C. This is of course impractical when actually executing a program, but does not cause any problems for the theorem prover when it is reasoning about the program. We do not reason about the contents of strings; string literals are encoded as unique constants whose value is unknown.

The Boogie program containing the compiled query is then

```
1   anon0:
2     call $tmp0 := System.Console.Read();
3     C.x[obj] := $tmp0;
4     call $tmp1 := System.Console.Read();
5     y := $tmp1;
6     C.x[obj] := C.x[obj] + 1;
7     goto anon15_Then, anon15_Else;
8
9   anon15_Then:
10    assume C.x[obj] > y;
11    goto anon16_Then, anon16_Else;
12
13  anon16_Then:
14    assume C.x[obj] == 10;
15    assert false;
16    call System.Console.WriteLine(
17       $string_literal_0);
18    goto anon17_Then, anon17_Else;
19
20  anon16_Else:
21    assume C.x[obj] != 10;
22    goto anon17_Else;
23
24  anon17_Else:
25    // ... more code ...
```

Figure 5: A simplified view of the Boogie program resulting from the compiled C# code. The encoding of exceptional control paths has been elided, as well as the bookkeeping used to map Boogie source lines back to source lines in the original program. Line 15 shows the assertion that would be injected if the user had asked Get Me Here to reach the call to `WriteLine`.

$$\begin{aligned} \exists a, b \quad : \quad & C.x_0[obj] = a \wedge y_0 = b \wedge \\ & C.x_1[obj] = C.x_0[obj] + 1 \wedge \\ & C.x_1[obj] > y_0 \wedge x_1 = 10 \end{aligned}$$

Figure 6: A simplified version of the resulting Z3 fomula. It is *not* written in Z3's input language.

given to Corral [8, 9]. Corral is a whole-program analyzer for Boogie programs. It uses an SMT solver (Z3) for fast and precise reasoning about programs with unbounded data types such as integers and maps, and operations such as arithmetic, map select and update, etc. Corral essentially works by converting a subset S of the program's beahviors to a Z3 formula ϕ_S such that ϕ_S is satisfiable if and only if S contains an assertion violation. If S is not sufficient, then Corral issues a different query to Z3 to either expand S or concludes that the program does not have any assertion violations. This strategy of iteratively expanding S allows Corral to be goal-directed. For small programs (single procedure, no loops), Corral simply picks S to be the set of all program behaviors. For instance, for the Boogie program shown in Figure 5, the formula shown in Figure 6 is (a simplified version of) the Z3 formula.

All state changes in the program are encoded by having multiple *incarnations* of the same variable with equations linking the values between states. Incarnations are denoted with subscripts, thus x_1 is the value of x after it has been incremented from its value of x_0. When this formula is fed to Z3, it will try and find values of a and b that satisfy the formula. Thus, overall, Z3's search for a satisfying assignment essentially corresponds to searching over all possible executions of the program.

We now briefly describe how Corral works on larger programs (for which generating a single Z3 formula is either not feasible or too expensive). Corral uses the heuristic that for most programs, only a few number of variables and a few number of procedures will be relevant to the assertions in the program. Consequently, at any point in time, the subset S is defined as a pair (V, I), where V is a subset of the set of all Boogie variables, and I is a partially-inlined program. Initially, V is empty and I is just the entry procedure of the program. Semantically, S encodes the set of all program executions that are contained in I and only reason about variables in V, while abstracting away the rest. As the set V is increased, Corral gains more precision (trading off time because the Z3 formulae get more complicated). When I grows (i.e., more procedures are inlined), then the syntactic scope of Corral increases and it can find longer counterexamples (again, while trading off time). The actual process of Corral interacting with Z3 on how to expand V and I is more technically involved; we refer the interested reader to publications on Corral [8].

Because Boogie is a turing-complete language, the problem of finding assertion violations in Boogie is undecidable. Thus, it is possible that the expansion of S inside Corral may go on forever. In order to counter non-termination, Corral imposes a bound on I. (The set V is always bounded by the set of all global varibles in Boogie, which is a finite number.) This bound limits the number of loop iterations and recursive calls. For this paper's case studies, a bound of one (i.e., a path can have at most one loop iteration per loop and one recursive call) was sufficient to produce the results of our case studies.

Since Get Me Here injected a false assertion at the end point, when Corral produces a counterexample, then it must be a trace that finishes at the end point. The counterexample reported by Corral consists of a trace in the Boogie program along with values of variables at various points in the trace. Get Me Here translates this trace into one that makes sense for the C# program. The resulting trace has the following form:

$$
\begin{array}{rcl}
\mathit{Trace} & := & \mathit{Statement}^* \\
\mathit{Statment} & := & \texttt{call}\ \mathit{MethodName} \\
 & | & \texttt{location}\ \mathit{File}, \mathit{LineNumber} \\
 & | & \texttt{state}\ \{\mathit{Name} \mapsto \mathit{Value}\} \\
 & | & \texttt{return} \\
\mathit{Name} & := & \mathit{Identifier} \mid \mathit{Address} \\
\mathit{Value} & := & \texttt{int}\ \mathit{Integer} \\
 & | & \texttt{bool}\ \mathit{Boolean} \\
 & | & \texttt{fieldmap}\ \{\mathit{Identifier} \mapsto \mathit{Value}\} \\
 & | & \texttt{arraymap}\ \{\mathit{Integer} \mapsto \mathit{Value}\} \\
 & | & \texttt{ref}\ \mathit{Address}
\end{array}
$$

The `call` and `return` statements form a call tree, `location` indicates that the program executed the source code at the given location, and `state` describes a symbolic heap at the most recent `location`. The state map assigns values to local variables, global variables, and symbolic addresses that represent sharing and cycles in the heap graph. For example, if two local variables `x` and `y` both refer to an object with integer field `f` with value 3, then the state map contains $\{\texttt{x} \mapsto \texttt{ref}\ a_0, \texttt{y} \mapsto \texttt{ref}\ a_0, a_0 \mapsto \texttt{fieldmap}\{\texttt{f} \mapsto 3\}\}$.

3.2 Trace Visualization

The execution trace visualization is based on Debugger Canvas [5], which is an implementation of the Code Bubbles paradigm [3] for Visual Studio. For each `call` in the execution trace, Get Me Here creates a code bubble containing that method's text. Get Me Here uses the trace's `location` entries to highlight executed code in green and uses the trace's `state` entries to annotate the code with the values of variables. The visualization draws solid arrows to represent direct method calls and dashed arrows to represent indirect method calls (for example, due to events).

Technically, the trace file contains more information than is displayed in the visualization. In particular, within a method body, the trace file specifies the order in which statements are executed and interleaved with outgoing calls. To avoid clutter, the visualization does not present the order of statement execution, but simply uses background color to represent whether a statement is executed at all. In the absence of loops, this loss of information creates no ambiguities, since the programming language semantics dictate the order of execution. (Recursion is handled by creating one code bubble for each recursive call.) In the future, to accommodate loops, we will add a timeline slider to allow the user to witness the order of execution.

3.3 Multiple-Waypoint Queries

Get Me Here also supports a more general kind of query in which the user can specify multiple waypoints, namely, an optional start point, zero or more intermediate points, and an end point. Each waypoint can specify an optional condition, which is a predicate over the program state. The user interface for these queries is based on Visual Studio's familiar debugging margin (Figure 7). Just as the user can click on the debugging margin to create a breakpoint, the user can click on Get Me Here's margin (shown in dark gray) to create

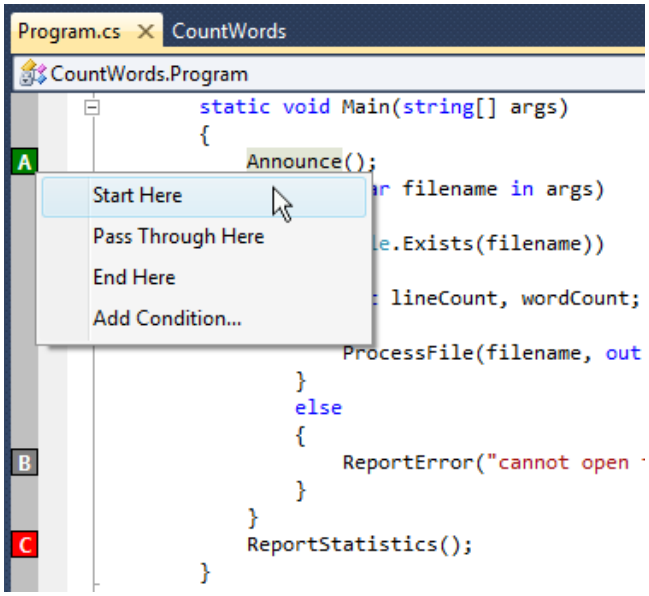

Figure 7: Invoking a Get Me Here query with a specified start line, intermediate line, and end line.

a waypoint. A context menu allows the user to specific the kind of waypoint, plus an optional condition. The user interface borrows familiar visual elements from route finding services, like Bing Maps or Google Maps. In particular, we letter the waypoints in the order they must be reached, use green for the start point and use red for the end point. The user invokes the "Find Execution Path" menu item (shown in Figure 1) to launch a search to satisfy a multiple-waypoint query. Using the "Get Me Here" menu item is a shortcut that both creates a query (a condition-less end point wherever the user invoked the menu and a start point at the program's entry point) and launches the search. We feel that this type of query is likely to be very common.

Supporting multiple waypoints is important given that the static analysis relies on an SMT solver. The solver is free to choose any counterexample, which means that Get Me Here can present any execution trace that reaches the end point. Of course, the solver's preferred choice may not necessarily be the user's preferred choice! Allowing a query to specify intermediate points and conditions allows the user to search iteratively for the trace she prefers. For example, if Get Me Here presents a trace that take a normal path through a method and the user prefers to see an exceptional path, she can place an intermediate point inside a `catch` statement. If Get Me Here presents a trace in which an environment variable has the value `"en-us"` and the user prefers the value `"en-uk"`, she can set a conditional waypoint at the line of code where that environment variable is read. In short, we expect that a common usage pattern will be for a user first to issue a simple endpoint-only query then follow with more complex multiple-waypoint queries to home in on a desired execution scenario. Because each code bubble in an execution trace is a full-featured editor with its own Get Me Here margin, a user can specify and launch a new query directly from the trace visualization of the previous query.

We encode user queries with one or more waypoints by instrumenting the program as follows. In the most general case, let the query have a starting point, N intermediate points ($N \geq 1$) and an end point. Each of these waypoints has a condition $\texttt{Cond}_i$. We introduce a new variable `$tracker` of type `int`. At the starting point of the query, we insert the following peice of code:

```
assume(Cond_0);
$tracker := 0;
```

At the n^{th} intermediate point, we insert the code:

```
assume $tracker == n - 1;
assume(Cond_n);
$tracker := $tracker + 1;
```

At the end point, we insert the code:

```
assume $tracker == N;
assume(Cond_{N+1});
assert false;
```

When program exploration begins at the start node, the nature of this instrumentation ensures that the only way to reach the `assert false` is to go through the waypoints in order before reaching the end node.

3.4 Trading off Accuracy for Speed

Corral is a whole-program analyzer for Boogie programs. However, Get Me Here does not give Corral a Boogie translation of the the user's whole program (including all assemblies used directly or indirectly), since this would be too much code to analyze in a reasonable amount of time. Instead, we allow the user to make the trade-off between speed and accuracy at the level of individual methods, by presenting Corral with one of three versions of a method's body:

- Full code (the slowest and most accurate). BCT provides a complete translation of the method's body.
- Model code. We provide a simpler, surrogate version of a method's body, typically written in a source language like C#.
- No code (the fastest and least accurate). We provide no body, in which case Corral assumes that the method has no side effects and uses nondeterministic values for the return value and output parameters.

By default, Get Me Here provides full code for all methods that are part of the user's project files and no code for the .NET framework assemblies. Get Me Here uses the model code for certain framework methods that are necessary to reach parts of the user's code. For example, the Tetris program uses the WinForms library for creating its graphical user interface (GUI). As is typical with GUI libraries, the Tetris code registers many methods as callbacks for particular UI events and then calls `Application.Run` to wait for these methods to be called. If Corral were presented with no body for `Application.Run`, none of the registered callbacks would be reachable.

To solve this problem, we have experimented with two model versions of the WinForms library. The first version correctly

```
public static void Run(Form form)
{
  form.OnLoad(EventArgs.Empty);
  form.OnShown(EventArgs.Empty);
  while (!form.IsClosed)
  {
    if (Picker.PickBool())
    {
      var control =
        Control.AllControls.PickOne();
      control.DoRandomEvent();
    }
    else
    {
      var menuItem =
        MenuItem.AllMenuItems.PickOne();
      menuItem.OnClick(EventArgs.Empty);
    }

    var timer = Timer.AllTimers.PickOne();
    timer.OnTick(EventArgs.Empty);
  }
  form.OnClosed(EventArgs.Empty);
}
```

Figure 8: The model of the UI event loop we used in the Tetris case study.

models many details of the library, including the organization of the UI into a tree of elements, the collection of these trees into a stack to implement modal dialogs, and the algorithm for propagating events up the tree at the top of the stack. The second version (Figure 8) simply keeps a global set of all UI elements and repeatedly sends an arbitrary event to an arbitrary UI element. The second model is more permissive that the actual WinForms implementation, but is faster for Corral to explore. We allow the user to choose between the two models. For those programs, like Tetris, whose behavior does not depend on such details as the hierarchical propagation of events, the less accurate model provides faster query responses.

4. EVALUATION

Our ultimate goal is to test whether developers using Get Me Here are are more productive and more accurate in completing software maintenance tasks. However, in order to test our tool with users, it needs to be robust enough to handle arbitrary queries and responsive enough to answer queries in a reasonable amount of time. Hence, this initial case study evaluates the tool's performance on three benchmark programs, chosen for suitability for user testing. To measure the extent to which developers struggle to answer reachability questions on these programs, we ran a formative user study in which participants used Visual Studio alone to reproduce and fix one bug in each program. Here, we describe our benchmark programs, the results of the user study, and our measurements of Get Me Here on these three benchmarks.

4.1 Benchmark Programs

Our three benchmark programs were chosen with several criteria in mind. They were downloaded from the internet and used unmodified (except for the seeded bugs in the user study) to be representative of real programs. To test the scalability of our approach, they are different orders of magnitude in size. They use three different styles of interaction with the user (console, GUI and web). They represent problem domains that are familiar to a typical developer, without the need for domain expertise. Finally, they are all written in the same programming language so that developers recruited for the study needed to know only one language.

- **Calculator**, 650 lines of C#, is a console program that accepts simple arithmetic expressions and evaluates them. It has 249 methods. We seeded a bug in which the modulus operator pops three operands rather than two.
- **Tetris**, 3000 lines of C#, is an implementation of the popular Tetris game, with a graphical user interface built using the WinForms library. It has 357 methods. We seeded a bug by removing a case from a `switch` statement that chooses which game piece should fall next.
- **Tailspin Toys**, 11,000 lines of C# (along with 400 lines of ASPX, 1400 lines of JavaScript), is a sample retail web site for purchasing toys. It has 1613 methods. This code already had a bug in which adding an item twice to the shopping cart results in only one item.

Both Calculator and Tetris contain `Main` methods as starting points. Tailspin Toys, however, is a web service and exposes multiple entry points. To accommodate this, we created a harness that nondeterministically calls methods marked with the attribute `AcceptVerbs(HttpVerbs.Get)` or `AcceptVerbs(HttpVerbs.Post)`. The creation of such harnesses could easily be automated.

4.2 Formative User Study

To measure the extent to which developers struggle to answer reachability questions on these benchmark programs, we ran a user study in which participants were asked to reproduce and fix one bug in each program. We recruited 12 professional developers from the Puget Sound area, all male, average of 41 years old, with an average of 14 years of professional experience. Two participants completed only the first task, and there data are not included in the study.

We ran each participant independently in a two-hour session. After completing a demographic questionnaire, each participant was given three tasks to complete. Each task consisted of a vague bug report, whose wording is given below. Each participant was first asked to show the experimenter the steps necessary to consistently reproduce the bug. Then each participant was asked to fix the bug. Each participant was given roughly a half-hour to complete each task. All participants completed tasks 1 and 3, and all but one completed task 2.

Task 1: Your team mate sent you the following email: When I tested the Tetris game, it would crash at random.

When I ran in in the debugger, it would eventually have an assert failure with this message: "bad nextFig!"

Task 2: Your tester send you the following email: I forgot to tell you before I left work that the Calculator game me a bad error message. It said "not enough arguments" even though I used it correctly. Sorry I can't remember exactly what I typed in since I'm away from my desk.

Task 3: Your tester send you the following email: I spent the day trying out the Tailspin Toys web site. Mostly, it works great! I only found one bug. I had entered some stuff in the shopping cart and I got an uncaught exception that says "something went wrong". (Nice exception message!) I've been trying since then, but I haven't been able to repro the problem.

Each bug report contains a string that all participants managed to find immediately in its respective program. We call the corresponding control point — program statement — the *task query*. By design, reproducing the bug amounts to a reachability question: under what circumstance does the program reach the line of code containing that string? The participants varied considerably in the amount of time it took them to reproduce the bug.

The Calculator program proved to be the biggest challenge for our participants to reproduce, despite its modest size. There are many call paths that lead to the `Pop` method, which means there are many parts of the code to check. Because developers took considerable time and felt considerable frustration in reproducing bugs in these programs, we conclude that they are worthy benchmarks.

4.3 Performance on the Task Queries

We used Get Me Here to find each of the three task queries mentioned in Section 4.2. Corral was able to produce a path in each case. Each time, the path passed through the buggy portion of code. The displayed path along with the data values made the bug quite evident. Details on Corral's performance on these queries is reported in Figure 10. For each of the task queries, we show: the number of minutes that Corral took to produce a path (Time); the number of iterations Corral required and the number of variables tracked (Iters/Vars); and some statistics about the size of the resulting trace, namely, the number of lines of code (LOC), the number of method calls (Calls), and the size of the stack trace at the task query (Stack). In our experience, the number of Corral iterations and tracked variables usually give a measure of the difficuly of a particular query.

For Console Calculator and Tailspin Toys, Corral found the relevant execution trace considerably faster than the developers did in the lab study. Tetris had the most complex logic of the three benchmarks (reflected in the large number of iterations and tracked variables), and consequently gave Corral a hard time. Even though Corral took longer than developers did in the lab study, all of this time is automated. Human time is spent only looking at the trace reported by Corral (and that time is not reported in these results). Console Calculator had the highest stack depth, and the task query was further away from `Main` than for other programs. This may be one reason why developers found it difficult to

Task	Time to reproduce bug (minutes)	
	min. – max.	mean (s.d.)
1. Tetris	1 – 8	2:18 (2:13)
2. Calculator	3 – 33	15:47 (7:43)
3. TailSpin	2 – 21	8:12 (5:33)

Figure 9: Developers' performance on the three reachability questions.

Task	Time	#Iters	Resulting Trace		
	(min)	(#Vars)	LOC	Calls	Stack
1. Tetris	7:36	6 (35)	250	96	6
2. Calculator	1:17	4 (11)	124	172	7
3. TailSpin	1:15	2 (4)	586	238	3

Figure 10: Corral's performance on three reachability questions.

debug Console Calculator. Tailspin Toys, despite its large size, proved to be the easiest for Corral to explore. It did, however, produce a long trace that highlighted 586 lines of code. This experiment shows that automated reasoning and manual reasoning can be complementary to each other.

4.4 Performance on Arbitrary Queries

It is possible that the queries issued in the previous section may not be representative of the possible queries that a user of Get Me Here may have issued. This section measures the performance of Corral on a large set of queries on our benchmark programs. For each program, we generated one Get Me Here query for each line of code that is either (1) a possible target of a branch, or (2) the beginning of a method. This generated 180 queries for Console Calculator, 567 queries for Tetris and 1278 queries for Tailspin Toys. We ran Corral on all of these queries with two goals: what is the average time taken by Corral to answer these queries, and for what fraction of queries was Corral not able to produce a definite reasult (i.e., it reached the depth bound, which is set to one for this study).

The results are shown in Figure 11. This table shows the overall number of queries issued and the average time taken (in seconds) on these queries. The other columns divide up the queries according to the answer that Corral reported: Reachable (i.e., Corral found a path), Unreachable (i.e., there is no path that reaches the target), or Out-of-Bounds (i.e., Corral reached the depth bound and the result in inconclusive). For reachable queries, the table also gives the average length of the path (in terms of lines of code).

The results show that most queries were answered in a reasonable amount of time. Furthermore, only a fairly small number of queries resulted in an inconclusive answer for Console Calculator (5%) and Tailspin Toys (4%), although the number is higher for Tetris (21%). This shows that Corral is able to get a good amount of coverage on our benchmarks.

Most of queries that were deemed unreachable by Corral was because the end point was in dead code (either inside framework methods that were never called from the program, or inside methods that were not invoked from `Main`,

Task	Overall		Reachable			Unreachable		Out-of-Bounds	
	Num	Time	Num	Time	Length	Num	Time	Num	Time
Tetris	567	118	250	62	252	196	13.1	117	415.5
Calculator	180	9.1	65	14.9	65	105	3.6	10	29.4
TailSpin	1278	26.2	304	60.6	449	926	12.3	48	78.1

Figure 11: Performance of Corral on all auto-generated Get Me Here queries. All times are in seconds.

given our modeling of the stubs). For instance, the large number of unreachable queries for Tailspin Toys is because our harness did not invoke some of the functionality of the web service. Consequently, Corral did not take much time on such queries.

5. CONCLUSIONS

Even with the advanced features of today's development tools, developers often struggle to answer reachability questions about their code. In this paper, we repurposed existing verification tools to create a prototype query engine, Get Me Here, that is capable of automatically answering many reachability questions. In our case study, Get Me Here found execution traces for three reachability questions that developers struggled to answer in the lab. For two of the three queries, Get Me Here found the trace considerably faster than developers did. When given a set of arbitrary queries, Get Me Here provided accurate answers to most queries in a reasonable amount of time. This suggests that Get Me Here is sufficiently robust for a future user evaluation.

Our initial experience also suggests some interesting extensions. The first is the ability to transition between static and dynamic traces. When Get Me Here produces a trace, it typically contains all the interactions with the environment needed to reach a line of code. These interactions could be replayed to recreate the trace at runtime. For example, the trace from Console Calculator provides values for all the parsed tokens, which makes it possible to reconstruct the input needed to witness the same behavior in the debugger. In the reverse direction, logs of the program's behavior can be readily turned into multiple-waypoint queries in order to explore execution traces that are capable of generating the contents of the log. For example, it the user has a log entry that says `EXIT ERROR 0x12345`, then she can find the corresponding code `Log.Write("EXIT ERROR {0}", code)` and issue a Get Me Here query with the condition `code == 0x12345`. With the current prototype, transitioning between static and dynamic traces is possible, but tedious, suggesting the possibility of future automation.

Another extension is to increase the interaction between the programmer and the tool. For instance, if the tool's search were to report its progress by showing the frontier of explored code, then the programmer could iteratively (or even interactively!) guide the search. When Corral gets stuck because of its bounded search, then the programmer could locally adjust the bounds.

An easy way to refine the stubs is also crucial. Even with stubs for the most frequently used Framework classes, programmers will always run into situations where a new environment model is needed. The ability to write them in the same programming language as the implementation is written in makes it possible for the intended users of the system to effectively write stubs. Without stubs for the Framework's Stack class, Get Me Here reports an infeasible error trace that does not respect the semantics of the stack's actual behavior. Such error traces reduce the programmer's confidence in the tool. It requires only a very simple model for the Stack class in order to have Get Me Here produce a realistic trace.

Over the past years, the research community has made considerable progress toward automated program verification. Our goal with this preliminary paper is suggest a new goal for the same community: making analysis algorithms an online part of the development environment to keep developers informed about execution behavior throughout the development process.

6. REFERENCES

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[2] M. Barnett and S. Qadeer. BCT: A translator from MSIL to Boogie. In M. Huisman, editor, *Bytecode 2102: Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2012. To appear.

[3] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[5] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. In *Proceedings international conference on Software Engineering (to appear)*, ICSE '12, 2012.

[6] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[7] A. J. Ko and B. A. Myers. Debugging reinvented:

Asking and answering why and why not questions about program behavior. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[8] A. Lal, S. Qadeer, and S. Lahiri. Corral: A whole-program analyzer for Boogie. Technical Report MSR-TR-2011-60, Microsoft Research, 2011.

[9] A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *Computer Aided Verification (submitted)*, CAV '12, 2012.

[10] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.

[11] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011*, pages 117–124, 2011.

[12] K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at **http://research.microsoft.com/ leino/papers.html**.

[13] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, Mar. 2010.

[14] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[15] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[16] M. Weiser. Programmers use slicing when debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.

[17] A. Zeller. Yesterday, my program worked. today, it does not. why? *ACM Sigsoft Software Engineering Notes*, 24(6):253–267.