AskTheCode: Interactive Call Graph Exploration for Error Fixing and Prevention

Robert Husák, Jan Kofroň, Filip Zavoral
Charles University, Faculty of Mathematics and Physics,
Prague, Czech Republic
husak@ksi.mff.cuni.cz, jan.kofron@d3s.mff.cuni.cz, zavoral@ksi.mff.cuni.cz

In order to prevent and fix errors in program code, developers need to understand its semantics to a significant extent. For this purpose, they use various approaches, such as manual call graph exploration or dynamic analysis with a debugger. However, these techniques tend to be cumbersome in a larger codebase, because they provide either underapproximate or overapproximate results and it is often hard to combine them. Therefore, we present AskTheCode, a Microsoft Visual Studio extension enabling to interactively explore a call graph, ensuring that only feasible execution traces are taken into consideration. AskTheCode is based on control flow analysis and backward symbolic execution. We show its potential to significantly improve developers' experience on a complex code example.

1 Introduction

When developers want to fix and prevent errors by understanding the semantics of the source code, they usually use manual call graph exploration and debugging. However, call graphs are often complicated and the calls can be constrained by various conditions. Debugging, on the other hand, might be misleading and incapable of capturing all the possible situations. Therefore, finding the exact context under which a bug can occur is tedious and error-prone [7].

We believe there is an opportunity to utilize a sound analysis technique for this task, for several reasons. First, the context is usually much more limited than in the case when we try to verify the program as a whole. Second, the developers might be more willing to wait long enough for the analysis to complete, because it may save much more time to them than if they wanted to do it manually. Last, even if the problem is too complicated to be handled properly in a reasonable time, developers can interact with the tool, applying appropriate abstractions and assumptions. Therefore, we have created a tool named AskTheCode, which utilizes control flow analysis and backward symbolic execution to help developers investigate particular problems in the source code.

2 Design

The primary purpose of AskTheCode is to reason about the context under which a certain situation in a program can occur. Therefore, an efficient way to formulate what problem we need to address is writing an assertion a_0 specifying the expected semantics and check in which situations it can be violated. Notice that this approach is suitable not only for bug fixing, but also for their prevention, because developers can use the tool to confirm their possibly incorrect ideas about the program semantics.

To verify an assertion, a natural approach is to inspect the code against the control flow and look for any input that eventually violates it. Because AskTheCode is a Microsoft Visual Studio extension aimed

Submitted to:
InterAVT 2019

at C# code analysis, it can access the Roslyn .NET compiler [11]. We use it to construct control flow graphs and the call graph of the methods related to the problem being solved. Next, we utilize backward symbolic execution [1, 2] to perform the assertion verification itself, employing the Z3 SMT solver [3]. Depending on the user configuration, an *entry point* is either a non-private method in the same class as where a_0 occurs, or a public method in a public class within the project. To tackle path explosion and other symbolic execution problems, users can also adjust loop unwinding, recursion limit, and timeout.

All the execution traces from *entry points* to violating a_0 are continuously gathered during the run of backward symbolic execution. We provide users with a panel allowing them to interactively explore those traces as if they were a program under execution. They can step forward and backward throughout the particular methods and statements and see their intermediate values. Furthermore, they can also inspect the relevant heap structure and its changes.

There is also a high-level overview in the form of an annotated call graph, rendered using the Microsoft Automatic Graph Layout library [10]. The graph is continuously updated as backward symbolic execution runs on background. At the beginning, it consists only of the method m_0 containing a_0, but then it expands as the explored states extend to the callers and callees of m_0. Whenever an execution trace is discovered, all the methods it traverses are emphasized in red to draw the user's attention. On the other hand, if there are not any states capable of extending to a method m_u, we mark m_u as unreachable by making its background green. The user can also explicitly ignore certain methods, letting the analysis focus on less complicated ones.

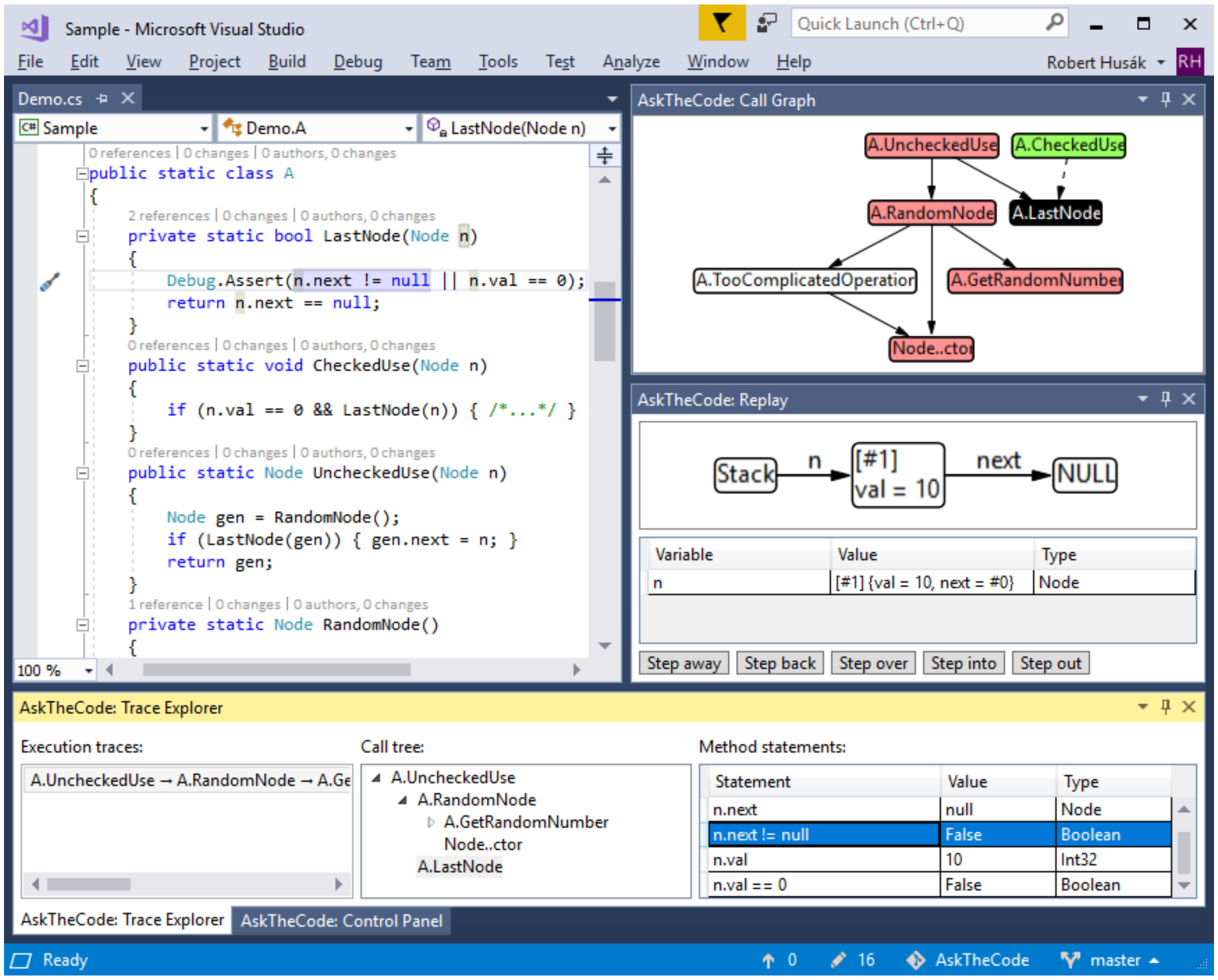
3 Example

To demonstrate how can AskTheCode help developers, let us inspect the code example in Figure 1. It presents an excerpt from a library working with singly linked lists. They are represented by the `Node` class containing an integer field `val` and a reference to the following node called `next`. Due to a certain implementation issue, it is demanded that whenever a `Node` is last in a list, its `value` must be zero. In `LastNode`, we check this property using an assertion. Imagine that we have received a bug report stating that this property is sometimes violated, but we were not given any exact context under which it can happen.

```
private bool LastNode(Node n) {
  Debug.Assert(n.next != null || n.val == 0);
  return n.next == null;
}
public void CheckedUse(Node n) {
  if (n.val == 0 && LastNode(n)) { /*...*/ }
}
public Node UncheckedUse(Node n) {
  Node gen = RandomNode();
  if (LastNode(gen)) { gen.next = n; }
  return gen;
}
private Node RandomNode() {
  int v = GetRandomNumber();
  if (v == 0) return new Node(0, null);
  if (v == 1) return new Node(10, null);
  return TooComplicatedOperation();
}
```

Figure 1: Example C# code with an assertion in `LastNode` to be verified.

The most common approaches to solving these tasks are manual call graph exploration and debugging, both of which can be problematic in certain situations. Manual call graph exploration requires us to inspect the logic of each affected method to ensure the situation is handled in there. In our example, we need to inspect `CheckedUse`, although it never allows the error to happen. Debugging, on the other hand, reveals only the situations which can indeed cause the error being inspected. However, it might be often difficult to reproduce the error and we cannot be sure that we have discovered all the contexts under which it can occur. Furthermore, the information provided by the debugger is limited. Considering our example, after reproducing



```
0 references | 0 changes | 0 authors, 0 changes
public static class A
{
    2 references | 0 changes | 0 authors, 0 changes
    private static bool LastNode(Node n)
    {
        Debug.Assert(n.next != null || n.val == 0);
        return n.next == null;
    }
    0 references | 0 changes | 0 authors, 0 changes
    public static void CheckedUse(Node n)
    {
        if (n.val == 0 && LastNode(n)) { /*...*/ }
    }
    0 references | 0 changes | 0 authors, 0 changes
    public static Node UncheckedUse(Node n)
    {
        Node gen = RandomNode();
        if (LastNode(gen)) { gen.next = n; }
        return gen;
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private static Node RandomNode()
    {
```

Variable	Value	Type
n	[#1] {val = 10, next = #0}	Node

Execution traces:

A.UncheckedUse → A.RandomNode → A.Ge

Call tree:

- A.UncheckedUse
  - A.RandomNode
    - A.GetRandomNumber
    - Node..ctor
  - A.LastNode

Method statements:

Statement	Value	Type
n.next	null	Node
n.next != null	False	Boolean
n.val	10	Int32
n.val == 0	False	Boolean

Figure 2: AskTheCode in Microsoft Visual Studio 2017

the error and pausing the program execution on the violated assertion, we can see UncheckedUse in the call stack. However, we cannot see what exactly happened in the call of RandomNode, because we no longer know the indeterministic value v and hence the executed branch.

The mentioned problems can be solved using AskTheCode, as we can see in Figure 2. The call graph displayed in the top right panel shows that LastNode is called both by CheckedUse and UncheckedUse. CheckedUse is displayed with a green background and dashed arrow, as it is proven not to cause the error. On the other hand, UncheckedUse is emphasized in red together with its callees RandomNode, GetRandomNumber and the Node constructor, as there was found an assertion violating program trace going through them. RandomNode can also potentially call another method, TooComplicatedOperation, whose definition we have intentionally skipped. As we can tell from its name, it was too complicated for backward symbolic execution to handle automatically. Therefore, as it is shown in the call graph, we at least know that it is worth to be inspected manually, unlike CheckedUse.

Regarding the found trace, we can explore it in the bottom panel by inspecting its call tree and the particular statements. Whenever we select a statement in the table, the appropriate piece of code is selected in the opened code editor in the top left part. Furthermore, there is a replay panel on the right in which we can see the contents of the heap and of all the local variables at the given step of the trace.

On its bottom, there are buttons to navigate through the trace in a debugger-like fashion, with the added capability of stepping backward in the history. From the trace, we can now easily discover that at least one of the problematic situations happens when `v` in `RandomNode` is 1.

4 Related Work

In the field of human interface design, there are numerous tools helping developers to understand the semantics of the code [12, 4]. Probably the closest one to our approach is Reacher [8], a tool to interactively and intuitively explore complex call graphs. However, these tools do not reason about reachable paths as soundly as symbolic execution. Furthermore, they are not directly aimed at verifying assertions, hence no production of error traces.

On the other hand, to discover erroneous program inputs, we can use symbolic execution, whose current state of the art is summarized in [1]. Most of these techniques analyse a program by systematically exploring its state space from various entry points in a forward fashion, aiming for high code coverage. In order to reason about particular assertions, we use the backward variant of symbolic execution, whose most important representative is Snugglebug [2]. Despite its advanced capabilities used to alleviate the path explosion problem, it can still occur in practice. In such situations, Snugglebug and other related tools [5, 9] cannot provide sufficient information due to their lack of interactivity.

5 Conclusion

On a complex example which models situations known from practice, we demonstrate that AskTheCode can help developers understand causes of errors. Utilizing interactive approach, certain limitation of used backward symbolic execution can be alleviated, e.g. by voluntarily omitting certain problematic places or controlling loop unwinding. In the future, we plan to extend the interactivity even more, and implement more advanced features such as state merging [6] or directed call graph construction [2], ultimately making AskTheCode production-ready.

Acknowledgements

This work was supported by the project PROGRESS Q48, the Czech Science Foundation project 18-17403S and the grant SVV-2017-260451.

References

[1] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu & I. Finocchi (2018): *A survey of symbolic execution techniques*. *ACM Computing Surveys (CSUR)* 51(3), p. 50, doi:10.1145/3182657.

[2] S. Chandra, S. J. Fink & M. Sridharan (2009): *Snugglebug: A Powerful Approach to Weakest Preconditions*. *SIGPLAN Not.* 44(6), pp. 363–374, doi:10.1145/1543135.1542517.

[3] L. De Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[4] R. Deline & K. Rowan (2010): *Code canvas: Zooming towards better development environments*. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, 2, pp. 207–210, doi:10.1145/1810295.1810331.

[5] P. Dinges & G. Agha (2014): *Targeted Test Input Generation Using Symbolic-Concrete Backward Execution*. In: *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, Västerås, Sweden, doi:10.1145/2642937.2642951.

[6] V. Kuznetsov, J. Kinder, S. Bucur & G. Candea (2012): *Efficient State Merging in Symbolic Execution*. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, ACM, New York, NY, USA, pp. 193–204, doi:10.1145/2254064.2254088.

[7] T. D. LaToza & B. A. Myers (2010): *Developers Ask Reachability Questions*. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, ACM, New York, NY, USA, pp. 185–194, doi:10.1145/1806799.1806829.

[8] T. D. LaToza & B. A. Myers (2011): *Visualizing call graphs*. In: *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 117–124, doi:10.1109/VLHCC.2011.6070388.

[9] K. Ma, K. Yit Phang, J. S. Foster & M. Hicks (2011): *Directed Symbolic Execution*. In: *Static Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 95–111, doi:10.1007/978-3-642-23702-7_11.

[10] L. Nachmanson, A. Nocaj, S. Bereg, L. Zhang & A. Holroyd (2016): *Node Overlap Removal by Growing a Tree*. In: *Graph Drawing and Network Visualization*, Springer International Publishing, Cham, pp. 33–43, doi:10.1007/978-3-319-50106-2_3.

[11] T. Neward & J. Hummel (2014): *Rise of Roslyn*. *MSDN* 29(11), p. 70. Available at `https://msdn.microsoft.com/en-us/magazine/dn818501.aspx`.

[12] J. Smith, Ch. Brown & E. Murphy-Hill (2017): *Flower: Navigating program flow in the IDE*. In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 19–23, doi:10.1109/VLHCC.2017.8103445.