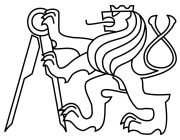
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Handling heap data structures in backward symbolic execution
Student:	Bc. Robert Husák
Supervisor:	doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2018/19

Instructions

Starting point: Backward symbolic execution uses constraint solvers [1], usually in the form of SMT solvers [2], to find program inputs that lead to a certain bug. Current SMT solvers support basic data structures, but cannot directly handle more complex data structures, for example, pointer structures on the heap.

Objective: Extend the backward symbolic execution tool AskTheCode [3] with the handling of heap data structures.

Methodology:
1) Investigate and summarize the relevant literature.
2) Identify the most promising method(s) for transforming operations on heap data structures into constraints for SMT solvers.
3) Design an integration of those methods into the AskTheCode tool and implement the result.
4) Document and test your extension on appropriate examples.

References

[1] P. Dinges and G. Agha. Targeted test input generation using symbolic-concrete backward execution. In 29th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2014. ACM.
[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability Modulo Theories." In Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications, (A Biere, M J H Heule, H van Maaren, and T Walsh, eds.), IOS Press, Feb. 2009, pp. 825–885.
[3] Code Assertions Verification Using Backward Symbolic Execution, diploma thesis, https://is.cuni.cz/webapps/zzp/detail/179174/

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague September 8, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Handling heap data structures in backward symbolic execution

Mgr. Robert Husák

Department of Software Engineering
Supervisor: doc. Dipl.-Ing. Dr. techn. Stefan Ratschan

May 9, 2018

Acknowledgements

I would like to thank my supervisor doc. Dipl.-Ing. Dr. techn. Stefan Ratschan for his advice and patience. I am also grateful for the support of my family, without which I would be unable to finish this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 9, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Husák, Robert. *Handling heap data structures in backward symbolic execution.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Cílem této práce je významně rozšířit AskTheCode, dříve vytvořený doplněk nástroje Microsoft Visual Studio, který využívá zpětnou symbolickou exekuci pro verifikaci asercí v kódu C#. Jedno z největších omezení našeho doplňku byla neschopnost analyzovat objekty na haldě a operace na nich. Pro doplnění této funkcionality jsme nejdříve začali rešerší existujících postupů, ze kterých jsme vybrali tři nejzajímavější: *línou inicializaci*, *symbolickou inicializaci* a využití teorie polí. Tyto techniky jsme důkladně analyzovali s přihlédnutím na specifické požadavky našeho nástroje. Díky jejím očekáváným výkonnostním charakteristikám jsme vybrali použití teorie polí. Tuto techniku jsme transformovali, aby ji bylo možné využít ve zpětné symbolické exekuci, včetně efektivního využití zásobníků podmínek v SMT řešičích. Na řadě příkladů v jazyce C# jsme následně ukázali korektnost této implementace.

Klíčová slova zpětná symbolická exekuce, symbolická halda, líná inicializace, symbolická inicializace, teorie polí

Abstract

This thesis enhances AskTheCode, a previously created extension of Microsoft Visual Studio which uses backward symbolic execution to verify assertions in

C# code. One of the biggest AskTheCode limitations was the inability to reason about heap objects and operations. In order to implement this feature, we started by surveying existing techniques. As the most promising ones were selected *lazy initialization*, *symbolic initialization* and the utilization of the theory of arrays. After an analysis driven by the specific requirements of our tool, we decided to utilize the theory of arrays, mainly due to its expected performance benefits. We transformed the technique to be usable for backward symbolic execution, utilizing assertion stacks of SMT solvers as efficiently as possible. Our solution was proven to be correct by an evaluation on several C# examples.

Keywords backward symbolic execution, symbolic heap, lazy initialization, symbolic initialization, theory of arrays

Contents

List of Figures

Introduction

In order to improve efficiency of a software development process, it is often useful to equip developers with various code analysis tools, such as *Resharper* or *CodeRush* in the case of C#. These tools help to inspect and refactor source code, but cannot precisely reason about its general semantics nor correctness. For these purposes, error-finding tools such as *Coverity* or *SonarLint* are available; however, they target only general code problems and often produce a high number of false positives. Even more insight is provided by *Microsoft IntelliTest* or *Code Contracts static checker*, but their proper usage demands a significant time investment.

As a result, we decided to devise a Visual Studio extension that would help developers to easily verify arbitrary assumptions about their code and to find root causes of various domain specific errors. Instead of trying to verify whole programs, it utilizes a demand-driven approach, focusing only on a certain code assertion at a time. Although almost any reasoning about the semantics of a program is generally undecidable, this way we have a better chance of providing meaningful information to users.

Because of its demand-driven approach, we called the tool AskTheCode [1]. It utilizes backward symbolic execution, whose basic principle is to start an exploration at a program point of interest and explore all the execution paths leading to it from a possible entry point. Each of these paths can be characterized by a system of constraints put on the input of the program, commonly referred to as a path condition. The techniques developed to reason about such first-order formulas are described in the first chapter, symbolic execution itself in the second one.

Currently, AskTheCode is limited to work only with integer and boolean variables and cannot handle any complex objects. The aim of this thesis is to extend its functionality so that it is eventually able to reason about heap objects and their operations. In the third chapter we survey the most promising existing techniques for this purpose. Their usability in practical settings is shown in the following chapter, where we mention some existing

tools utilizing them. The fifth chapter then summarizes the fundamentals of AskTheCode itself.

With the sixth chapter we move to the implementation itself, starting by specifying exact requirements on the resulting solution. One of the most important hurdles mentioned is the fact that the techniques previously surveyed are described to be directly used only in the forward variant of symbolic execution and we must transform them in order to be useful to us. Another challenges are the design decisions to be made regarding the AskTheCode integration and performance considerations.

In the seventh chapter we analyse all the techniques with regard to the requirements and select the most promising one, whose implementation and integration is then explained from a high level perspective. The low-level perspective is then used in the eight chapter, which explains the changes performed in the individual classes among the modules of AskTheCode.

The last chapter describes the mechanisms used to ensure that our implementation is correct with respect to our goal. After concluding the whole thesis, two attachments are present: the AskTheCode user documentation and the content of the enclosed CD.

Part I

Background

Chapter 1

Satisfiability Modulo Theories

Satisfiability modulo theories (SMT) is a decision problem whose solvers are frequently used in various techniques of software and hardware verification. We will explain its basic motivation, provide its formal definition and then focus on the area of SMT solvers.

1.1 Motivation

Suppose we want to determine the satisfiability of the following first-order formula:

$$x < y \ \wedge \ \neg(z \leq y) \ \wedge \ z < x$$

Apart from the logical operators, we can see the variables x, y, z and the symbols $\leq$, $<$ and 0. If we did not restrict the semantics of these symbols, we might have ended up with a false positive result, because the solver could have created it in an arbitrary way, e.g. $\forall a, b(a < b \wedge \neg(a \leq b))$. Therefore, we need to validate the formula in the context of an existing theory, such as linear integer arithmetic. To reason about certain formulas, more than one theory may be needed, e.g. both integer arithmetic and real arithmetic. We can also be interested in creating a custom symbol and checking whether there exists an interpretation that renders it valid in a formula.

1.2 Definition

To target the mentioned needs, we need to extend first-order logic with the notion of sorts, forming many-sorted first-order logic. A sort resembles a data type in programming, it can be for example an integer or a real number. A many-sorted *signature* $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ comprises a set of *sorts* $\mathcal{S}$, a set of *function symbols* $\mathcal{F}$ and a set of *predicate symbols* $\mathcal{P}$. Each function symbol $f \in \mathcal{F}$ bears an arity of the sorts $\sigma_1 \times ... \times \sigma_n \to \sigma$, whereas each predicate

symbol $p \in \mathcal{P}$ is associated with an arity of $\sigma_1 \times ... \times \sigma_n$. For $n = 0$, f is called a *constant symbol* and p is called a *propositional symbol*. In order to combine multiple signatures into one, we simply unite their respective feature sets, possibly renaming the symbols in case of name collisions.

Variables Σ_v, *terms* Σ_t, *atoms* Σ_a and *formulas* Σ_f are defined in the standard recursive way known from first-order logic. A Σ-*formula* $\varphi \in \Sigma_f$ is a formula containing only the symbols from Σ, logical operators, quantifiers and variables, whereas each variable is of a sort included in Σ. The usage of the symbols and variables must also correspond to their respective arity and particular sorts, e.g. a predicate p with an arity σ_a cannot be used in a formula $\varphi = p(v)$ if the sort of the variable v is not σ_a.

Apart from the commonly used logical operators, we will often use the if-then-else construct $ite(\varphi, t_1, t_2)$. If φ is $true$, the resulting term has the value t_1, otherwise t_2.

A Σ-*theory* $\mathcal{T}$ is a set of Σ-formulas called axioms. In order to combine a Σ_1-theory $\mathcal{T}_1$ and a Σ_2-theory $\mathcal{T}_2$ into a Σ-theory $\mathcal{T}$, we combine Σ_1 and Σ_2 into Σ and put $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2$. A Σ-*model* $\mathcal{A}$ contains a non-empty domain M_σ for each sort σ in Σ and *interpretations* of all symbols in Σ, which will be marked by a superscript of $\mathcal{A}$. An interpretation of a function f with an arity $\sigma_1 \times ... \times \sigma_n \to \sigma$ is a projection $f^{\mathcal{A}} = M_{\sigma_1} \times ... \times M_{\sigma_n} \to M_\sigma$. Similarly, an interpretation of a predicate p with an arity $\sigma_1 \times ... \times \sigma_n$ is a projection $p^{\mathcal{A}} = M_{\sigma_1} \times ... \times M_{\sigma_n} \to \{true, false\}$. Terms, atoms and formulas are interpreted recursively, free variables in formulas are interpreted in the same way as constants.

A Σ-model $\mathcal{A}$ *satisfies* a Σ-formula φ iff $\varphi^{\mathcal{A}} = true$. We say $\mathcal{A}$ is a model of a Σ-theory $\mathcal{T}$, written $\mathcal{A} \models \mathcal{T}$, iff it satisfies all of its axioms; $\mathcal{T}$ is *satisfiable* iff such model exists. Finally, a Σ-formula φ is considered *satisfiable modulo* Σ-*theory* $\mathcal{T}$ if $\mathcal{T} \cup \varphi$ is satisfiable. In order to validate the satisfiability modulo multiple theories, those theories can be combined into a single one [2, 3].

1.3 Solvers

As we can see, the SMT problem is a generalization of the Boolean satisfiability problem (SAT), which operates only on propositional symbols and does not allow quantifiers. Although the SAT problem is NP-complete, various algorithms effective in practice have been developed. Probably the most established one is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [4, 5], which works on a backtracking basis. Basically, in each step, a literal is selected, a truth value is assigned to it, the formula is simplified and the resulting one is checked recursively.

There are two major approaches to utilizing this experience in the field of SMT solvers. *The eager approach* works by transforming the given SMT formula into an equivalent SAT one and then passing it to a SAT solver. *The*

lazy approach accompanies particular theory solvers into the DPLL algorithm directly, known as DPLL(T) [6]. The latter is currently more popular, because of its efficiency and modularity [7]. In fact, the DPLL works as a basic framework there and the solvers of particular theories can be inserted into it.

Therefore, the decidability of an SMT instance depends on the particular theories used in it. For example, nonlinear integer arithmetic is generally undecidable, whereas the first order theory of the natural numbers with addition is decidable [8]. The decidability of certain theories can depend on the usage of quantifiers, e.g. the extensional theory of arrays $T_A^=$: [9]

$$\forall a, i, j (i = j \Rightarrow read(write(a, i, v), j) = v)$$

$$\forall a, i, j (i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j))$$

$$\forall a, b (a \neq b \Rightarrow \exists i \in I . read(a, i) \neq read(b, i))$$

This theory is decidable only if no quantifiers are used in the SMT formula.[1] [10] $T_A^=$ is an example of theories crafted to reason about the logic of software, together with the theory of algebraic data-types, the theory of bit-vectors etc. [7].

In order to unify the interface of various SMT solvers, the SMT-LIB standard was introduced [11]. The solvers complying to it provide a text interface with commands and responses written in a language with Lisp-like syntax, making them easy to parse. There are various commands for declaring custom sorts, functions, predicates and manipulating the formula to be verified. See an example of the verification of certain 2D integer vector dot product properties in Listing 1.1:

Listing 1.1: Example SMT-LIB code to reason about certain properties of the 2D integer vector dot product

```
(declare-datatypes () ((Vec (vec (x Int) (y Int)))))

(define-fun dot ((u Vec) (v Vec)) Int
  (+ (* (x u) (x v)) (* (y u) (y v)))
)
(define-fun mul ((s Int) (u Vec)) Vec
  (vec (* s (x u)) (* s (y u)))
)

(declare-const c1 Int)
(declare-const c2 Int)
(declare-const a Vec)
(declare-const b Vec)
```

[1] Quantifiers found in the aforementioned axioms do not cause any problems as their semantics is transferred to the solver.

```
(push)
(assert (not (= (dot a b) (dot b a))))
(check-sat)
(pop)

(push)
(assert (= (dot (mul c1 a) (mul c2 b))
           0))
(check-sat)
(get-model)
(pop)
```

Using the `declare-datatypes` command, we declare a new sort in our theory: 2D integer vector called `Vec`. Three functions are added to the signature and interpreted: `vec` to construct a vector from two integers and the couple of `x` and `y` to extract the respective integer components from a vector. Next, we utilize the `define-fun` command to define two another functions: the dot product and scalar multiplication. The `declare-const` command is used to add four constant symbols to our theory: two integers and two vectors.

The formula to be verified is a conjunction of assertions added by the `assert` command. Because one often needs to verify formulas similar to each other, an assertion stack is used. By the `push` command, a "restore point" is created on the top of the stack. When the `pop` command is called, the point is removed from the top of the stack and all the assertions, declarations and definitions added since it are forgotten. This mechanism allows the solver to cache the knowledge of the common part of the verified formulas, such as inferred lemmas etc.

After calling the `check-sat` command, three different results are possible: satisfiable (SAT), unsatisfiable (UNSAT) and unknown. In the first case, a model of the provided theory can be obtained from the SMT solver by the `get-model` command. The last case can occur either when the used theories are undecidable or when a resource limit is reached, which can happen due to the NP-completeness of the SMT problem.

In Listing 1.1, we use the assertion stack to verify two different formulas using the same signatures and interpretations of the `dot` and `mul` functions. At first, we show how to validate a certain formula for all possible interpretations of the declared constants on the example of the commutativity of the dot product. Instead of asserting it directly, we assert its negation instead. Then, if the negation proves to be unsatisfiable, the formula itself is verified. In the other case, we can get the interpretation that disproves the original formula by using the `get-model` command. If we want only to find one interpretation the formula is satisfied by, we assert the formula directly, as we can see in the other case, where we ask whether a result of a certain operation can be zero.

In our case, the first `check-sat` returns UNSAT, verifying the commutativity. The second one returns SAT and the related `get-model` returns the following:

Listing 1.2: Example model retrieved by an SMT-LIB compliant solver

```
(model
  (define-fun c2 () Int 0)
  (define-fun b () Vec (vec 38 1236))
  (define-fun a () Vec (vec 0 7719))
  (define-fun c1 () Int 0)
)
```

As we can see, when the constants `c1` and `c2` are zero, the result of the expression `(dot (mul c1 a) (mul c2 b))` is zero as well.

Chapter 2

Symbolic Execution

Aforementioned SMT solvers gave rise to various static program analysis algorithms. In this thesis, we will focus mainly on the technique called *symbolic execution*. First, we explain the reasons that led to its invention. Second, two variants of it are presented, the forward one and the backward one. Next, we mention the most important problems of this technique, together with the options of how to deal with them. Last, we discuss various options of how to formalize symbolic execution.

2.1 Motivation

Consider the following fragment of code:

Listing 2.1: Example fragment of C# code with an error

```
1 x = -x;
2 if (x > 0) {
3   return 1;
4 } else if (y < 0) {
5   return -1;
6 } else {
7   Debug.Assert(x == 0);
8   return 0;
9 }
```

Suppose the original intention was to return the reverse sign function of the integer variable `x`; however, an unfortunate mistype to another existing integer variable `y` occured in the second `if` clause. Even if the code is completely covered by unit tests, the error does not need to be discovered. The problem of the common way the tests are performed is that their execution usually represents just a fraction of the possible values that can really be stored in the variables. The aim of symbolic execution is to eliminate this problem by

replacing these values by symbolic constants, trying to describe all the possible executions of a program by their combination into expressions and formulas.

2.2 Forward Symbolic Execution

When talking about symbolic execution, we usually mean its *forward* variant [12]. In this case, the algorithm starts by gathering the inputs to the program being analysed, a symbol is assigned to each of them. In our example from Listing 2.1, two symbolic constants named x_0 and y_0 for the corresponding variables are created. Whenever a new variable is encountered, its value is obtained as an expression of the already known symbols. Because a symbol can only have one interpretation, when a new value is assigned to a known variable, it is considered a new variable as well. In our example, the assignment `x = -x` will cause a new symbolic constant x_1 to be created and from now on, it will be used whenever the `x` variable is used.

The current position in the program, the knowledge of the currently known symbols and their relation to the program variables is called a *state* of the algorithm. A *path condition*, another part of the state, is a formula that uniquely describes the execution path by specifying constraints on the symbolic values. In the beginning, the path condition is an empty conjunction, resulting in *true*. Depending on the strategy of the particular algorithm, various conditions can be added to it to mimic the manipulation with the program variables. In our example, the assignment `x = -x` causes $x_1 = -x_0$ to be added to the path condition.

Whenever a branching occurs, the state is split and the symbolic predicate corresponding to the branching condition is added to path condition of the first one and its negation to the second one. In our example, the path conditions of the states after the first `if` clause will be $x_1 > 0$ for the former and $\neg(x_1 > 0)$ for the latter. After the second `if` clause the second state will split again, adding y_0 and yielding three states with the following path conditions:

$$\varphi_1 = (x_1 = -x_0) \wedge x_1 > 0$$

$$\varphi_2 = (x_1 = -x_0) \wedge \neg(x_1 > 0) \wedge y_0 < 0$$

$$\varphi_3 = (x_1 = -x_0) \wedge \neg(x_1 > 0) \wedge \neg(y_0 < 0)$$

In our case, these path conditions describe all the possible execution paths of the program. In order to discover whether there are any inputs that steer the program along these paths, we can pass these formulas one by one to an SMT solver. The interpretations of x_0 and y_0 obtained from the respective models would then represent the `x` and `y` inputs of the program. As we can see, symbolic execution can be used to obtain test inputs that achieve high code coverage.

Particularly important inputs are those which cause errors in the program, e.g. null dereference, division by zero or invalid assertion. All such errors can be expressed by symbolic constraints and added to the current path condition to be verified by the SMT solver. If UNSAT is retrieved, the error cannot occur on this execution path. Otherwise, there exists at least one input that causes it, which can be obtained from the respective model. In the case of the `Debug.Assert(x == 0)` assertion, we simply pass the following path condition to the SMT solver:

$$\varphi_4 = \varphi_3 \wedge \neg(x_1 = 0)$$

$$= (x_1 = -x_0) \wedge \neg(x_1 > 0) \wedge \neg(y_1 < 0) \wedge \neg(x_1 = 0)$$

It returns SAT and produces a model, its interpretations of x_0 and y_0 can be for example 1 and 0, respectively.

2.3 Backward Symbolic Execution

As we can see, classic forward symbolic execution can be used to explore the semantics of a whole program, finding possible errors by searching from its entry point. If we want to focus on a specific potential error or a location that is suspected of being unreachable, it is often more effective to use *backward symbolic execution* [13]. As its name suggests, it starts on a specific program point and searches backwards against the execution flow until it reaches the entry point. Then, it uses an SMT solver to find feasible paths and sample inputs in the same way as the original forward variant.

Say we want to verify the `Debug.Assert(x == 0)` assertion from Listing 2.1 using backward symbolic execution. We start by declaring a symbolic constant x_0 corresponding to the version of the variable `x` used in the assertion. Apart from this knowledge, the initial state of the algorithm contains also an initial path condition $\neg(x_0 = 0)$. Going against the control flow, we identify that the expression `y < 0` must be `false` in order that the current position in the code is reached. Therefore, we add another symbolic constant y_0 for `y` and add $\neg(y_0 < 0)$ to the path condition. Similarly, we also add $\neg(x_0 > 0)$. Reaching the `x = -x` statement, we discover that x_0 does not model an arbitrary input to the program. Instead, its value fully depends on the value stored in the previous version of `x`, the one on the right side of the assignment. Correspondingly to the forward variant, we create a symbolic constant x_1, consider it to refer to the current value of `x` and add $x_0 = -x_1$ to the path condition. Eventually the following path condition is produced:

$$\varphi_5 = \neg(x_0 = 0) \wedge \neg(y_0 < 0) \wedge \neg(x_0 > 0) \wedge (x_0 = -x_1)$$

As we can see, $\varphi_5 \Leftrightarrow \varphi_4$, only the meaning of x_0 and x_1 is flipped. If the SMT solver now produces a model where $x_1 = 1$ and $y_0 = 0$, it means the input causing the error is `x = 1` and `y = 0`.

In general, backward symbolic execution is more suitable when we want to verify a fixed set of possible problems in the program instead of deeply analysing it as a whole. For simple programs similar to the one in Listing 2.1, the advantage is not apparent. However, it is usually impossible to automatically verify complicated systems completely; therefore, such goal-driven approach can at least help to reason about some of their properties.

2.4 Caveats

As we can see from the description above, symbolic execution basically aims to explore all the possible execution paths of a program. The total number of these paths grows exponentially with each branch; furthermore, it can be infinite if cycles or recursion are used. This phenomenon is called the *path explosion* problem.

To explain how it can be alleviated, let us note that symbolic execution in practice usually works on a best-effort basis. The algorithm is given certain memory and time limits and it tries to cover as many interesting execution paths of the program as possible. In order to determine what paths are we particularly interested in, a heuristic can be provided to the algorithm. For example, when looking for test inputs to achieve high code coverage, the heuristic may prefer the paths leading to previously unvisited locations. The limits can be highly customized as well, e.g. in our case the algorithm may stop if an input extending the coverage has not been found in the last 30 seconds.

Another technique attempting to tackle path explosion while preserving the information is *state merging*. Any two states s_a, s_b of the algorithm with the same position in the code can be merged into one state by appropriately combining their relations to the program variables and path conditions. As a result, the number of the calls of the SMT solver can be reduced, because all the paths reachable from both s_a and s_b will not be searched twice. On the other hand, the formulas to verify will certainly be more complicated, which can prolong the time consumed by the SMT solver. Therefore, it is beneficial to use a heuristic to decide when to merge the states [14].

Apart from the path explosion problem, there is another complication. Certain operations are either difficult or impossible to model, such as the interaction with the environment, transcendental functions over floating point numbers etc. In such cases, it is often useful to abandon the effort to prove the absolute correctness of the program and explore it in an experimental way. In contrast to symbolic execution, compiling and testing the program by providing certain inputs is often called *concrete execution*. The result of combining those techniques is therefore named *concolic testing* [15]. The

basic idea behind it is that the program is given certain inputs, its execution is observed and any conditional jumps occurred during it are recorded. Then, the SMT solver is used to find different inputs to steer the execution towards the previously unexplored parts of the program. This process is repeated until a limit is reached or the program is considered to be explored. A neat effect of concolic testing is also its speed, because concrete execution is generally faster than the symbolic one.

Apparently, concolic testing is not directly applicable to backward symbolic execution, because it is impossible to run the program backwards. However, a technique called *symcretic execution* attempts to introduce concrete execution there as well. In the first phase, backward symbolic execution tries to find any paths leading to a particular program point of interest from the entry point, possibly skipping statements difficult to model. Then, the program is executed with various inputs satisfying the previously found conditions, trying to reach back. This phase can be used to overcome complicated loops or unmodelled operations by trying enough number of inputs using various heuristics [16].

2.5 Formalization

Having reviewed the principles behind symbolic execution, we will now inspect different ways of how to define it formally. One of the difficult tasks is to define a language for the programs to be analysed. On the one hand, it needs to be concise enough not to impede the understandability of the overall explanation. On the other hand, it must be complex enough to demonstrate all the capabilities of the technique in the given context. Having the programming language defined, the definition of the program state needs to be provided, together with the algorithm that processes the states with respect to the particular programming constructs.

As a result, the degrees of formalism and detail vary among the literature. It is common for an author to create a custom programming language and a state definition that capture the features needed for the particular publication. For example, Kuznetsov in his paper about efficient state merging [14] presents a programming language consisting only of four instructions: an assignment of an expression to a variable, a conditional jump, an assertion and a program halt. In the paper, a state (l, pc, s) consists of a program location l, a path condition pc and a symbolic store s that maps each variable to an expression over input variables and concrete values. We can see that there are no notions of advanced features such as function calls or objects stored on the heap.

A diametrically opposite example is the exhaustive formalization of Java bytecode symbolic execution from Xianghua [17], where the whole Java Virtual Machine is mapped to a symbolic state which contains global variables, an operand stack and a heap among others. Regarding the symbolic heap, a

useful formalization is provided also by Hillery in [18].

As this thesis is focused only on a certain aspect of symbolic execution and formalizing the whole mechanism tends to be too complicated, we decided not to do it. Instead, in the next chapter we will formulate the problem of symbolic heap handling separately from the overall symbolic execution algorithm itself. Later, we will prove the validity of our approach by adding the implementation of the symbolic heap to an already existing symbolic execution tool.

Chapter 3

Symbolic Heap

The programs analysed by symbolic execution and other SMT-related techniques often contain code constructs that cannot be transformed into logical formulas in a straightforward manner. The one we will focus on in this thesis is the manipulation with the objects located in heap memory.

In this chapter we will cover several existing techniques invented to handle this problem. Although they can be modified to work with the backward variant of symbolic execution, we will describe them in the context of the forward one, because they were originally presented that way. As we have already mentioned, the complete algorithm of symbolic execution will not be formalized due to its complexity. Instead, in order to compare the techniques, we will create a common interface consisting of a set of operations each technique must provide to the core symbolic execution algorithm.

As a result, from the standpoint of a heap modelling technique, we do not have to claim any specific requirements on the original language of the analysed program. Instead, we demand the symbolic execution algorithm to preprocess it in a way so that we can reason only about the heap objects and omit tasks such as call stack modelling or interpreting different variants of loops.

In particular, we need that there are only two kinds of data types: primitive types, which can be directly modelled by an underlying SMT solver, and compound types, which contain fields of other types and are stored on the heap. The concept of strongly typed variables is also needed, each variable of a primitive type is meant to be stored on the stack and passed by value whereas each variable of a compound type is only a reference to an object on the heap. The references are meant only to access the referenced objects and to be compared between each other, there are not meant as pointers, hence no support for pointer arithmetic. As we can see, the most straightforward to model this way are certain subsets of languages like C# or Java, where the compound types correspond to classes. Therefore, we will call the compound types that way.

The formalization of these requirements follows. Let C denote a set of classes, each class c being a pair $(M_c\ F_c)$, where M_c contains its methods and F_c its fields. The unions of all these entities in the program are located in M and F:

$$M = \bigcup_{c \in C} M_c \qquad F = \bigcup_{c \in C} F_c$$

For a method $m \in M$, the set V_m contains all of its local variables. We demand that $\forall m_1, m_2 \in M\ (V_{m_1} \cap V_{m_2} = \emptyset)$, which can be easily done by prefixing the name of each variable with the corresponding class and method name. A set V includes all local variables present in the methods:

$$V = \bigcup_{m \in M} V_m$$

Their types and the types of the fields are indicated by a function $t : V \cup F \to \mathcal{S} \cup C$. On the right side of the mapping, $\mathcal{S}$ is the set of all sorts contained in the signature Σ used in an underlying SMT solver. As mentioned before, we can think of such variables as of the primitive types, whereas a variable $t(v) \in C$ is considered to be of a reference type. To simplify reasoning about them, we introduce the following definitions:

$$V_{\mathcal{S}} = \{v \in V \mid t(v) \in \mathcal{S}\} \qquad F_{\mathcal{S}} = \{f \in F \mid t(f) \in \mathcal{S}\}$$
$$V_C = \{v \in V \mid t(v) \in C\} \qquad F_C = \{f \in F \mid t(f) \in C\}$$

For each class there exists a special strongly typed null variable to denote empty references:

$$\forall c \in C\ \exists null_c \in V\ (t(null_c) = c)$$

Note that all the reference variables are expected to point to objects on the heap, there is no notion of low-level pointers and of accessing stack variables by references. Furthermore, we decided not to target runtime polymorphism in this thesis, focusing mainly on the access and manipulation of the heap.

After defining the demands on the overall algorithm and the structures used in the interface, we will proceed to the interface itself. Let us define a *symbolic heap* as a tuple of variable length $(\varphi\ a_1\ a_2\ ...\ a_m)$, containing a path condition φ and m other elements $a_1, a_2, ..., a_m$. The path condition is, again, a logical formula expressing the requirements needed to reach a certain position in a program. The other elements are specific to each technique mentioned later in this chapter. All the possible heaps of a particular kind are contained in the set H.

The fundamental idea is that each state s of symbolic execution contains a corresponding symbolic heap $h_s \in H$. Whenever the algorithm encounters a heap-related operation in the analysed program, it executes the corresponding

operation on h_s. A new symbolic heap h'_s is produced, corresponding to h_s after performing the operation. Then, h'_s is used to form a new state s'. There are some cases, however, when a single operation on a heap can yield two or more different results. For example, when writing a field value to a reference r, there are two possible outcomes: h_{ok} with the successfully rewritten value and h_{err} representing the error of r being null. The algorithm of symbolic execution must be aware of these possibilities and handle them appropriately, for example by forking s into two states s_{ok} and s_{err}. As a result, all the operations produce a set of heaps $H' \in \mathcal{P}(H)$ instead of retrieving only a single heap. The operation list follows:

- `NEW` : $H \times V_C \to \mathcal{P}(H)$
  `NEW`(h, v) creates a new instance of the class $t(v)$ and assigns its reference to v.
- `EQ` : $H \times V_C \times V_C \to \mathcal{P}(H)$
  `EQ`(h, v_1, v_2) asserts the equality of v_1 and v_2.
- `NEQ` : $H \times V_C \times V_C \to \mathcal{P}(H)$
  `NEQ`(h, v_1, v_2) asserts the inequality of v_1 and v_2.
- `ASSERT` : $H \times \Sigma_f \to \mathcal{P}(H)$
  `ASSERT`(h, φ) adds the formula φ to the path condition of h.
- `ASSIGN` : $H \times V_C \times V_C \to \mathcal{P}(H)$
  `ASSIGN`(h, v, v_{in}) assigns the reference v_{in} to v.
- `READREF` : $H \times V_C \times V_C \times F_C \to \mathcal{P}(H)$
  `READREF`(h, v_{out}, v, f) assigns the reference $v.f$ to v_{out}.
- `WRITEREF` : $H \times V_C \times F_C \times V_C \to \mathcal{P}(H)$
  `WRITEREF`(h, v, f, v_{in}) assigns the reference v_{in} to $v.f$.
- `READVAL` : $H \times V_C \times F_{\mathcal{S}} \to \mathcal{P}(H \times \Sigma_t)$
  `READVAL`$(h, v, f) = \{(h_1\ t_1), ..., (h_n\ t_n)\}$ inserts the value of $v.f$ to the term t_i for each resulting heap h_i.
- `WRITEVAL` : $H \times V_C \times F_{\mathcal{S}} \times \Sigma_t \to \mathcal{P}(H)$
  `WRITEVAL`(h, v, f, t) writes the term t to $v.f$.

After presenting the theoretical framework, we will show how to use this abstraction in practice. In Listing 3.1, there is a definition of the `Node` class, whose instances are meant to be allocated on the heap. Each node contains an integer value and a reference to the next node, forming a singly linked list. The `SwapNode` method accepts an implicit nonnull parameter `this` of type `Node`. It is expected to modify the instances so that the current one and the following one are ordered by their values in non-decreasing order, returning

Listing 3.1: Sample C# code containing heap object manipulation [19]

```
1  class Node {
2    public int val;
3    public Node next;
4
5    public Node SwapNode() {
6      Node result = this;
7      if (this.next != null) {
8        if (this.val > this.next.val) {
9          Node t = this.next;
10         this.next = t.next;
11         t.next = this;
12         result = t;
13       }
14       Debug.Assert(result != null);
15       Debug.Assert(result.next != null);
16       Debug.Assert(result.val <= result.next.val);
17     }
18     return result;
19   }
20 }
```

the resulting list. There are also three assertions provided to check the output correctness.

In Figure 3.1 is shown the sequence of operations which would be performed by symbolic execution to analyse the sample C# code using an empty input heap h_0. The initialization of the environment is performed on the lines 1 to 3. Notice the helper variables hlp_0, hlp_1 and hlp_2, added in order to process the nested operations. The central block starting on the line 5 and ending on the line 18 is responsible for analysing the control flow of the method. For the sake of conciseness, we pass entire heap sets to each operation, which is a shortcut for applying the operation on all the heaps in the set and unioning the result. The sets H'_5 and H'_6 contain also the terms returned from `READVAL`. There are three possible execution paths, hence the result sets of heaps H_{p_1}, H_{p_2} and H_{p_3}. Because the second and the third path contain assertions to be checked, they are inspected together in the last block. The formula φ_a contains the disjunction of all conditions leading to violating the assertions.

The example will be used to illustrate several techniques aimed at expressing the heap in a symbolic way. We decided to select those that are either successfully used in practice or have the potential to achieve that. The names of the first two techniques are *lazy initialization* [19] and *symbolic initialization* [18], the third technique uses mapping to the theory of arrays [20, 21].

Figure 3.1: The analysis of the code in Listing 3.1 using a symbolic heap

1: $C = \{Node\},\ M_{Node} = \{SwapNode\}$
2: $F_{Node} = \{val, next\},\ t(val) = \mathbb{Z},\ t(next) = Node$
3: $V = \{this, result, t, null_{Node}, hlp_0, hlp_1, hlp_2\}, \forall v \in V(t(v) = Node)$
4:
5: $H_1 = \texttt{NEQ}(h_0, this, null_{Node})$
6: $H_2 = \texttt{ASSIGN}(H_1, result, this)$
7: $H_3 = \texttt{READREF}(H_2, hlp_0, this, next)$
8: $H_{p_1} = \texttt{EQ}(H_3, hlp_0, null_{Node})$
9: $H_4 = \texttt{NEQ}(H_3, hlp_0, null_{Node})$
10: $H'_5 = \texttt{READVAL}(H_4, this, val)$
11: $H'_6 = \bigcup_{(h\ val_0) \in H'_5} \{(h'\ val_0\ val_1) \mid (h'\ val_1) \in \texttt{READVAL}(h, hlp_0, val)\}$
12: $H_{p_2} = \bigcup_{(h\ val_0\ val_1) \in H'_6} \texttt{ASSERT}(h, val_0 \leq val_1)$
13: $H_7 = \bigcup_{(h\ val_0\ val_1) \in H'_6} \texttt{ASSERT}(h, val_0 > val_1)$
14: $H_8 = \texttt{ASSIGN}(H_7, t, hlp_0)$
15: $H_9 = \texttt{READREF}(H_8, hlp_1, t, next)$
16: $H_{10} = \texttt{WRITEREF}(H_9, this, next, hlp_1)$
17: $H_{11} = \texttt{WRITEREF}(H_{10}, t, next, this)$
18: $H_{p_3} = \texttt{ASSIGN}(H_{11}, result, t)$
19:
20: $H_{a_0} = H_{p_2} \cup H_{p_3}$
21: $H_{a_1} = \texttt{EQ}(H_{a_0}, result, null_{Node})$
22: $H_{a_2} = \texttt{READREF}(H_{a_0}, hlp_2, result, next)$
23: $H_{a_3} = \texttt{EQ}(H_{a_2}, hlp_2, null_{Node})$
24: $(H_{a_4}\ val'_0) = \texttt{READVAL}(H_{a_2}, result, val)$
25: $(H_{a_5}\ val'_1) = \texttt{READVAL}(H_{a_4}, hlp_2, val)$
26: $H_{a_6} = \texttt{ASSERT}(H_{a_5}, val'_0 > val'_1)$
27: $H_a = H_{a_1} \cup H_{a_3} \cup H_{a_6}$
28: $\varphi_a = \bigvee_{(\varphi \ldots) \in H_a} \varphi$

3.1 Lazy Initialization

The first one is named after the approach of handling input variables. Starting with an empty heap, whenever a reference variable previously unknown to the heap is introduced, it can point to three possible types of locations. Either it is null, an alias of an existing heap location or a completely new heap object. Lazy initialization creates a new heap for each such possibility. The formal definition of a heap utilizing this technique follows:

$$h^{LI} = (\varphi\ \eta\ R)$$

$$\eta : V_C \mapsto M \qquad M = \{l_{null}\} \cup M_{in} \cup M_{new} \qquad R : M \times F \mapsto M \cup \Sigma_t$$

$$h_0^{LI} = (true\ \eta_0\ \emptyset) \qquad \forall c \in C(\eta_0(null_c) = l_{null})$$

The partial function η, called the *environment*, is used to map variables of reference types to locations in the memory, denoted as M. It is divided into three subdomains: l_{null} is a special location to express null, M_{in} contains the input objects and M_{new} holds the objects created during the program. The *reference map*[2] R captures the data contained in the particular memory locations, which can be of two types for a certain location $l \in M$ and a field $f \in F$. If $f \in F_C$, $R(l, f)$ points to another place in M; otherwise, if $f \in F_S$, $R(l, f)$ contains an arbitrary term of the sort $t(f)$. The initial heap h_0^{LI} contains only the mapping of the null variables to l_{null}. The fundamental operations for initializing input variables and related references are defined below:

- $init((\varphi\ \eta\ R), v) = \begin{cases} \{(\varphi\ \eta\ R)\}, & v \in \eta^{\leftarrow} \\ \{h_{null}, h_{new}\} \cup H_{alias}, & v \notin \eta^{\leftarrow} \end{cases}$

$$h_{null} = (\varphi\ \eta[v \to l_{null}]\ R)$$

$$h_{new} = (\varphi\ \eta[v \to fresh_{M_{in}}()]\ R)$$

$$H_{alias} = \{(\varphi\ \eta'\ R) \mid l \in (\eta^{\rightarrow} \cup R^{\rightarrow}) \cap M_{in}, \eta' = \eta[v \to l]\}$$

- $init(h, v.f) = H'_{\exists} \cup H'_{\not\exists}$

$$H' = init(h, v)$$

$$H'_{\exists} = \{(\varphi\ \eta\ R) \mid (\varphi\ \eta\ R) \in H',\ (\eta(v)\ f) \in R^{\leftarrow}\}$$

$$H'_{\not\exists} = \begin{cases} H'_C, & t(f) \in C \\ H'_{\mathcal{S}}, & t(f) \in \mathcal{S} \end{cases}$$

$$v_f = fresh_V() \qquad H'_f = init(H' \setminus H'_{\exists}, v_f)$$

$$H'_C = \{(\varphi\ \eta\ R') \mid (\varphi\ \eta\ R) \in H'_f, R' = R[(\eta(v)\ f) \to \eta(v_f)]\}$$

$$H'_{\mathcal{S}} = \{(\varphi\ \eta\ R') \mid (\varphi\ \eta\ R) \in H', R' = R[(\eta(v)\ f) \to fresh_{\Sigma_v}()]\}$$

Note that when having a function g, we use $g^{\leftarrow}$ for its pre-image and $g^{\rightarrow}$ for its image. The function $g' = g[a \to b]$ differs from g only in the element a: $g'(a) = b$. We can also use sets in this syntax, e.g. $g'' = g[c \to d \mid c \in A]$. The function $fresh_D$ produces a value from the given domain D which is not used anywhere else.

As we can see, the core of lazy initialization lies in the first variant of *init*. If v is not present in η, it is either mapped to l_{null}, to a new input object or to an already existing input object. The second variant of *init* expresses

[2]Note that all the *maps* and *mappings* mentioned in this thesis are meant to be understood as partial functions of their respective domains, unless said otherwise.

the initialization of a variable and its field. If $f \in C$, *init* is called again on the given field; otherwise, a new symbolic variable of the respective sort is created to represent the primitive value. Let us now inspect the particular heap operations:

- $\texttt{NEW}((\varphi\ \eta\ R), v) = \{(\varphi\ \eta'\ R')\}$

$$l_{new} = fresh_{M_{new}}() \qquad \eta' = \eta[v \to l_{new}]$$
$$R' = R[(l_{new}\ f) \to l_{null} \mid f \in F_{t(v)} \cap F_C]$$

- $\texttt{EQ}(h, v_1, v_2) = H'$

$$H' = \{(\varphi\ \eta\ R) \mid (\varphi\ \eta\ R) \in init(h, v_1, v_2),\ \eta(v_1) = \eta(v_2)\}$$

- $\texttt{NEQ}(h, v_1, v_2) = H'$

$$H' = \{(\varphi\ \eta\ R) \mid (\varphi\ \eta\ R) \in init(h, v_1, v_2),\ \eta(v_1) \neq \eta(v_2)\}$$

- $\texttt{ASSERT}((\varphi\ \eta\ R), \varphi) = \{(\varphi'\ \eta\ R)\}$

$$\varphi' = \varphi_h \wedge \varphi$$

- $\texttt{ASSIGN}(h, v, v_{in}) = H'$

$$H' = \{(\varphi\ \eta'\ R) \mid (\varphi\ \eta\ R) \in init(h, v_{in}),\ \eta' = \eta[v \to \eta(v_{in})]\}$$

- $\texttt{READREF}(h, v_{out}, v, f) = H'_{err} \cup H'_{ok}$

$$H' = init(h, v.f)$$
$$H'_{err} = \{(\varphi\ \eta\ R) \mid (\varphi\ \eta\ R) \in H',\ \eta(v) = l_{null}\}$$
$$H'_{ok} = \{(\varphi\ \eta'\ R) \mid (\varphi\ \eta\ R) \in H' \setminus H'_{err},\ \eta' = \eta[v_{out} \to R(\eta(v), f)]\}$$

- $\texttt{WRITEREF}(h, v, f, v_{in}) = H'_{err} \cup H'_{ok}$

$$H' = init(h, v, v_{in})$$
$$H'_{err} = \{(\varphi\ \eta\ R) \mid (\varphi\ \eta\ R) \in H',\ \eta(v) = l_{null}\}$$
$$H'_{ok} = \{(\varphi\ \eta\ R') \mid (\varphi\ \eta\ R) \in H' \setminus H'_{err},\ R' = R[(\eta(v)\ f) \to \eta(v_{in})]\}$$

- $\texttt{READVAL}(h, v, f) = T'_{err} \cup T'_{ok}$

$$H' = init(h, v.f)$$
$$T'_{err} = \{((\varphi\ \eta\ R)\ v_{invalid}) \mid (\varphi\ \eta\ R) \in H',\ \eta(v) = l_{null}\}$$
$$T'_{ok} = \{((\varphi\ \eta\ R)\ R(\eta(v)\ f)) \mid (\varphi\ \eta\ R) \in H', \eta(v) \neq l_{null}\}$$

- $\texttt{WRITEVAL}(h, v, f, t) = H'_{err} \cup H'_{ok}$

$$H' = init(h, v)$$

$$H'_{err} = \{(\varphi\ \eta\ R) \mid (\varphi\ \eta\ R) \in H',\ \eta(v) = l_{null}\}$$

$$H'_{ok} = \{(\varphi\ \eta\ R') \mid (\varphi\ \eta\ R) \in H' \setminus H'_{err},\ R' = R[(\eta(v)\ f) \to t]\}$$

They are relatively straightforward, because the shape of the heap is completely captured in η and R. The readability may complicated by the fact that the most of them need to call *init* and process all the heaps created by it. We at least simplified the notation of calling it repeatedly, e.g. $\bigcup_{h' \in init(h,v_1)} init(h', v_2)$ is written as $init(h, v_1, v_2)$. To express undefined operations caused by null dereference, the corresponding heaps have the *err* underscript.

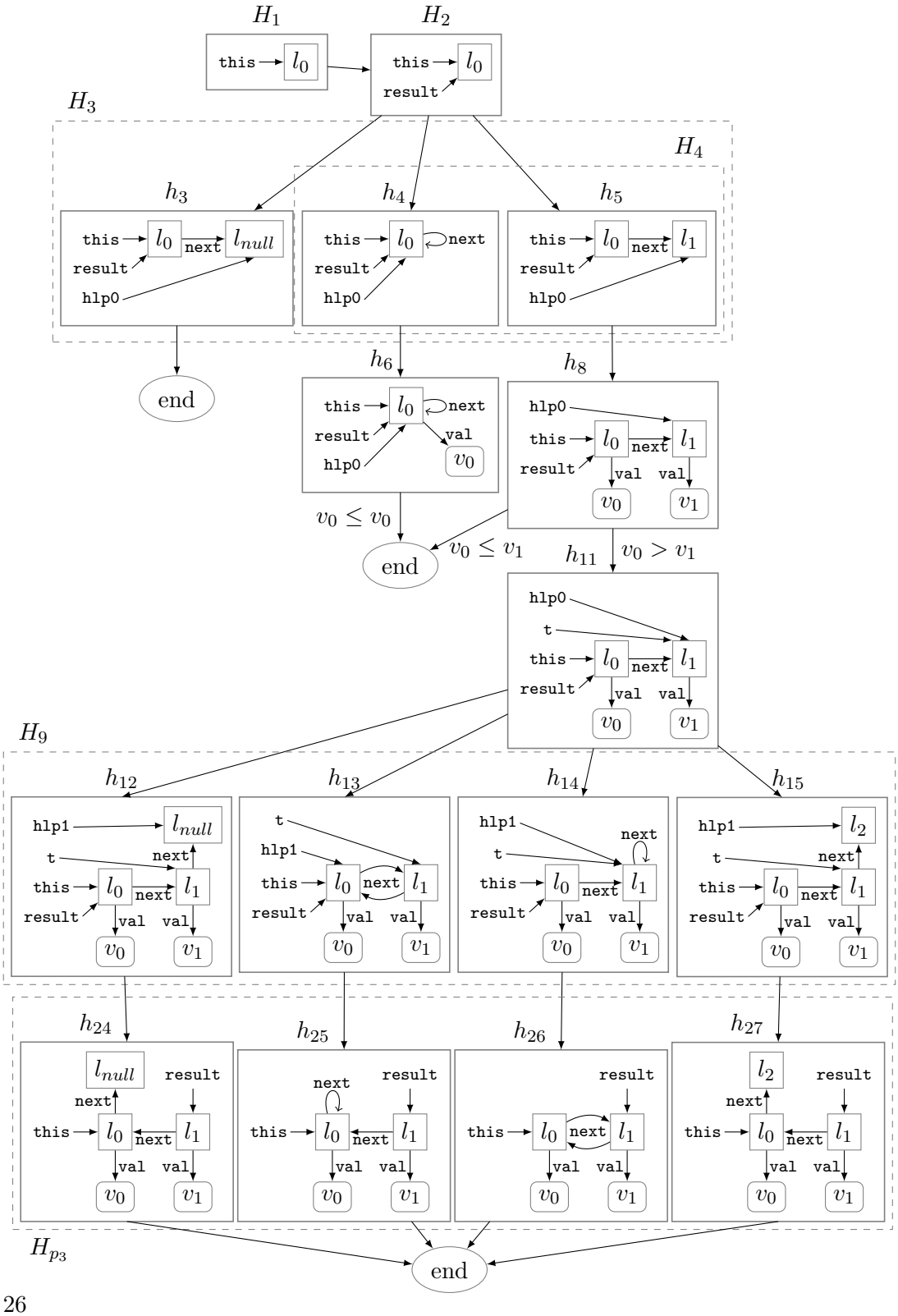
In Figure 3.2 we can see the intermediate results of the analysis listed in Figure 3.1. Each heap $h = (\varphi\ \eta\ R)$ is expressed as a rectangle displaying the heap structure, which is stored in η and R. If there are multiple heaps in a heap set, the set is displayed as a dashed rectangle around them. To obtain the path condition φ, find a path expressed by the arrows leading to h from H_1 and create the conjunction of all the attached formulas. Certain unimportant entities were hidden to save space, e.g. the hlp_0 variable from the point where it is not used any more.

$\texttt{NEQ}(h_0, this, null_{Node})$ causes the initialization of *this* and the elimination of the heap where it references l_{null}. As a result, H_1 contains only a single heap with $\eta(this) = l_0$. The following `ASSIGN` only maps *result* to the same location as *this*, creating H_2. By $\texttt{READREF}(H_2, hlp_0, this, next)$, the value of $R(l_0, next)$ is initialized and all the possibilities are contained in $H_3 = \{h_3, h_4, h_5\}$. Considering the definition of *init*, h_3 corresponds to h_{null}, h_5 to h_{new} and h_4 is the only member of H_{alias}. `NEQ` and `EQ` are used to divide H_3 into $H_{p_1} = \{h_3\}$ and $H_4 = \{h_4, h_5\}$, respectively. Two subsequent calls of `READVAL` are then used on h_4 and h_5 to obtain the terms stored in `this.val` and `this.next.val`, producing h_6 and h_8. The former is associated with one symbolic variable v_0 for both operations whereas the latter with two distinct ones, v_0 and v_1. H_{p_2} contains these heaps enriched with path conditions $v_0 \leq v_0$ and $v_0 \leq v_1$, respectively. Because $v_0 > v_0$ is unsatisfiable, H_7 contains only h_8 extended by adding a path condition of $v_0 > v_1$. $\texttt{ASSIGN}(H_7, t, hlp0)$ then yields h_{11} with the added variable t. This time, `READREF` produces four heaps $H_9 = \{h_{12}, h_{13}, h_{14}, h_{15}\}$, because there are already two locations that can be possibly aliased. After two calls to `WRITEREF`, we can see that $\eta(result) = l_1$ and $R(l_1, next) = l_0$ in all the heaps $H_{p_3} = \{h_{24}, h_{25}, h_{26}, h_{27}\}$. The assertions must be valid, implied by the path conditions and the shapes of the produced heaps.

As we can see, lazy initialization is quite powerful thanks to the simplicity of the produced path conditions. On the other hand, the upper bound of the

heap count is the factorial of the input heap size. Therefore, the complexity of a program to be analysed may be significantly limited when using this technique.

Figure 3.2: The usage of lazy initialization for the analysis in Figure 3.1



3.2 Symbolic Initialization

The main problem of the previous technique is the enormous growth of the number of heaps. Symbolic initialization aims to reduce it using a structure called *guarded value set*. It will be illustrated on the formal definition of such heap:

$$h^{SI} = (\varphi\ \eta\ L\ R)$$

$$\eta : V_C \mapsto \Sigma_v \qquad L : \Sigma_v \mapsto \mathcal{P}(\Sigma_f \times M) \qquad R : M \times F \mapsto \Sigma_v \cup \Sigma_t$$

$$h_0^{SI} = (true\ \eta_0\ L_0\ \emptyset)$$

$$\forall c \in C\ (\eta_0(null_c) = r_{null}) \qquad L_0(r_{null}) = (true\ l_{null})$$

In contrast with lazy initialization, another layer of mapping was added. The environment η does not longer map the program variables to particular locations in memory. Instead, it translates them to symbolic integer variables, called references. The *location map* L associates each reference r with the previously mentioned guarded value set. Every item in such set contains a location l and a logical formula representing the condition under which r maps to l. All the conditions in the set must be mutually exclusive so that no two locations can be pointed to at the same time. The reference map R from lazy initialization was altered to accommodate the mentioned layer as well. Therefore, the values of class reference fields are mapped back to the references, forming a bipartite graph.

The initial heap h_0^{SI} contains a special location l_{null} and the corresponding reference r_{null}. Other references can be of three types: supposing n is an arbitrary natural number, r_n^s denotes a stack reference, r_n^a an auxiliary reference and r_n^i an input reference. Whenever we encounter an unknown reference, we must, again, initialize it nondeterministically to possibly alias other input references. To distinguish the input references from the other types, we use the $isInput$ function. The initialization operations are described below. For the sake of conciseness, we skipped certain details of the algorithm, such as conditional initialization. Refer to the original article for the full description [18].

- $init((\varphi\ \eta\ L\ R), v) = \begin{cases} \{(\varphi\ \eta\ L\ R)\}, & v \in \eta^{\leftarrow} \\ \{(\varphi\ \eta'\ L'\ R)\}, & v \notin \eta^{\leftarrow} \end{cases}$

$$r = input_R() \qquad l = fresh_L()$$

$$\rho = \{(r_a\ l_a) \mid isInput(r_a)\ \wedge\ r_a = min_r(R^{\leftarrow}[l_a])\}$$

$$\theta_{null} = \{(\varphi'\ l_{null}) \mid \varphi' = (r = r_{null})\}$$

$$\theta_{new} = \{(\varphi'\ l)\} \mid \varphi' = (r \neq r_{null} \wedge (\wedge_{(r'_a\ l'_a)\in\rho}\ r \neq r'_a))\}$$

$$\theta_{alias} = \{(\varphi'\ l_a) \mid \exists r_a((r_a\ l_a) \in \rho \wedge \varphi' = (r \neq r_{null}\ \wedge\ r = r_a \wedge \wedge_{(r'_a\ l'_a)\in\rho\ \wedge\ r'_a<r_a}\ r \neq r'_a))\}$$

$$\theta = \theta_{null} \cup \theta_{new} \cup \theta_{alias}$$

$$L' = L[r \to \theta] \qquad \eta' = \eta[v \to r]$$

- $init(h, v.f) = \begin{cases} \{(\varphi\ \eta\ L\ R'_{\mathcal{S}})\}, & t(f) \in \mathcal{S} \\ \{(\varphi'\ \eta'\ L'\ R'_C)\}, & t(f) \in C \end{cases}$

$$(\varphi\ \eta\ L\ R) = init(h, v)$$

$$R'_{\mathcal{S}} = R[(l\ f) \to fresh_{\Sigma_v} \mid \exists(\theta\ l) \in L(\eta(v))\ ((l\ f) \notin R^{\Leftarrow})$$

$$X = \{(v_i\ l_i) \mid \exists(\theta\ l_i) \in L(\eta(v))\ ((l_i\ f) \notin R^{\Leftarrow}), v_i = fresh_V()\}$$

$$(\varphi'\ \eta'\ L'\ R') = init(h, v_1, ..., v_n)$$

$$R'_C = R'[(l_i\ f) \to \eta'(v_i) \mid (v_i\ l_i) \in X]$$

As well as in the previous technique, there are two variants of the *init* function. The first one initializes a given variable v in the case it is unknown. For this purpose, it collects to ρ all the existing input references that can be possibly aliased by the new reference r. They are then used to form the resulting guarded value set θ. The particular subsets θ_{null}, θ_{new} and θ_{alias} represent the possible values of r: null, a new location and an alias to an existing location.

Again, the second variant of *init* is a shortcut to initialize a variable v together with its field f. If f is scalar, the task is solved by creating new symbolic variables of the given type using $fresh_{\Sigma_v}$. Otherwise, if f is of a reference type, we create an auxiliary program variable v_i for each location l_i whose field f is not yet initialized. The set $\{v_1, ..., v_n\}$ is then used to initialize all the new locations using the repetitive calling of the first *init* variant. Finally, the couples $(l_i\ f)$ are associated with the newly initialized guarded value sets. The heap operations follow:

- $\texttt{NEW}((\varphi\ \eta\ L\ R), v) = \{(\varphi\ \eta'\ L'\ R')\}$

$$r = stack_R() \qquad l = fresh_L()$$

$$\eta' = \eta[v \to r] \qquad L' = L[r \to \{(true\ l)\}]$$

$$R' = R[(l\ f) \to r_{null} \mid f \in F_{t(v)} \cap F_C]$$

- $\texttt{EQ}(h, v_1, v_2) = \{(\varphi'\ \eta\ L\ R)\}$

$$(\varphi\ \eta\ L\ R) = init(h, v_1, v_2)$$

$$\Phi_\alpha = cEq(h, v_1, v_2) \qquad \Phi_1 = cNeq(h, v_1, v_2) \qquad \Phi_2 = cNeq(h, v_2, v_1)$$

$$\varphi' = \varphi \wedge (\bigvee_{\varphi_\alpha \in \Phi_\alpha} \varphi_\alpha) \wedge (\bigwedge_{\varphi_1 \in \Phi_1} \neg\varphi_1) \wedge (\bigwedge_{\varphi_1 \in \Phi_2} \neg\varphi_2)$$

- $\texttt{NEQ}(h, v_1, v_2) = \{(\varphi' \ \eta \ L \ R)\}$

$$(\varphi \ \eta \ L \ R) = init(h, v_1, v_2)$$

$$\Phi_\alpha = cEq(h, v_1, v_2) \qquad \Phi_1 = cNeq(h, v_1, v_2) \qquad \Phi_2 = cNeq(h, v_2, v_1)$$

$$\varphi' = \varphi \wedge (\bigwedge_{\varphi_\alpha \in \Phi_\alpha} \neg\varphi_\alpha) \vee (\bigvee_{\varphi_1 \in \Phi_1} \varphi_1) \vee (\bigvee_{\varphi_1 \in \Phi_2} \varphi_2)$$

- $\texttt{ASSERT}((\varphi \ \eta \ L \ R), \varphi) = \{(\varphi' \ \eta \ L \ R)\}$

$$\varphi' = \varphi_h \wedge \varphi$$

- $\texttt{ASSIGN}(h, v, v_{in}) = \{(\varphi \ \eta' \ L \ R)\}$

$$(\varphi \ \eta \ L \ R) = init(h, v_{in})$$

$$\eta' = \eta[v \to \eta(v_{in})]$$

- $\texttt{READREF}(h, v_{out}, v, f) = \{(\varphi_{err} \ \eta \ L \ R), (\varphi_{ok} \ \eta' \ L' \ R)\}$

$$(\varphi \ \eta \ L \ R) = init(h, v.f)$$

$$\varphi_{null} = \bigvee_{(\varphi_\theta \ l_{null}) \in L(\eta(v))} \varphi_\theta \qquad \varphi_{err} = \varphi \wedge \varphi_{null} \qquad \varphi_{ok} = \varphi \wedge \neg\varphi_{null}$$

$$r = stack_R() \qquad \eta' = \eta[v_{out} \to r]$$

$$L' = L[r \to \ (\varphi_r \wedge \varphi_f \ l') \mid (\varphi_r \ l) \in L(\eta(v)), (\varphi_f \ l') \in L(R(l, f)), SAT(\varphi \wedge \varphi_r \wedge \varphi_f)]$$

- $\texttt{WRITEREF}(h, v, f, v_{in}) = \{(\varphi_{err} \ \eta \ L \ R), (\varphi_{ok} \ \eta' \ L' \ R')\}$

$$(\varphi \ \eta \ L \ R) = init(h, v.f)$$

$$\varphi_{null} = \bigvee_{(\varphi_\theta \ l_{null}) \in L(\eta(v))} \varphi_\theta \qquad \varphi_{err} = \varphi \wedge \varphi_{null} \qquad \varphi_{ok} = \varphi \wedge \neg\varphi_{null}$$

$$\Psi = \{(\varphi' \ l \ r_{cur}) \mid (\varphi' \ l) \in L(\eta(v)), r_{cur} = R(l, f)\}$$

$$X = \{(l \ \theta) \mid (\varphi' \ l \ r_{cur}) \in \Psi, \theta = st(L, \eta(v_{in}), \varphi', \varphi_{ok}) \cup st(L, r_{cur}, \neg\varphi', \varphi_{ok})\}$$

$$R' = R[(l \ f) \to aux_R() \mid (l \ \theta) \in X]$$

$$L' = L[R'(l, f) \to \theta \mid (l \ \theta) \in X]$$

- $\texttt{READVAL}(h, v, f) = \{((\varphi_{err} \ \eta \ L \ R) \ T), ((\varphi_{ok} \ \eta \ L \ R) \ T)\}$

$$(\varphi \ \eta \ L \ R) = init(h, v.f)$$

$$\varphi_{null} = \bigvee_{(\varphi_\theta \ l_{null}) \in L(\eta(v))} \varphi_\theta \qquad \varphi_{err} = \varphi \wedge \varphi_{null} \qquad \varphi_{ok} = \varphi \wedge \neg\varphi_{null}$$

$$\Psi = \{(\varphi' \ t) \mid (\varphi' \ l) \in L(\eta(v)), t = R(l, f)\} = \{(\varphi'_1 \ t_1), ..., (\varphi'_n \ t_n)\}$$

$$T = ite(\varphi'_1, t_1, ite(...ite(\varphi'_n, t_n, v_{invalid})...))$$

- $\texttt{WRITEVAL}(h, v, f, t) = \{(\varphi_{err}\ \eta\ L\ R), (\varphi_{ok}\ \eta\ L\ R')\}$

$$(\varphi\ \eta\ L\ R) = init(h, v.f)$$

$$\varphi_{null} = \bigvee_{(\varphi_\theta\ l_{null}) \in L(\eta(v))} \varphi_\theta \qquad \varphi_{err} = \varphi \wedge \varphi_{null} \qquad \varphi_{ok} = \varphi \wedge \neg\varphi_{null}$$

$$R' = R[(l\ f) \rightarrow ite(\varphi', t, t_{cur}) \mid (\varphi'\ l) \in L(\eta(v)), t_{cur} = R(l, f)]$$

`NEW`, `ASSERT` and `ASSIGN` are fairly straightforward and similar to the corresponding operations in lazy initialization. The other ones tend to be a bit more complicated, because we need to operate on multiple heaps compressed to one. In the case of `EQ`, we use the helper functions $cEq(h, v_1, v_2)$ and $cNeq(h, v_1, v_2)$. The former is used to find the set of conditions under which the variables v_1 and v_2 point to the same locations. The latter provides us constraints under which v_1 points to such locations that v_2 never points to them under any constraint. Used together, they extend the path condition so that it enforces the equality of v_1 and v_2, if possible. `NEQ` is just a logical dual of `EQ`.

The remaining operations check for null references in a similar way as the previous technique, only this time are the constraints stored in the path condition. Then, `READREF` creates a new stack reference r, maps v_{out} to it and gathers all the possible locations l that can be pointed to by $v.f$. Each constraint $\varphi_r \wedge \varphi_f$ of such location is eventually checked to be satisfiable against the current path condition.

`WRITEREF` must work in a conditional way, because v can potentially point to multiple locations. Therefore, the produced guarded value set contains both the mappings to the current references r_{cur} and the mappings to the newly added reference $\eta(v_{in})$. A helper strengthening function is used:

$$st(L, r, \varphi, \varphi_g) = \{(\varphi \wedge \varphi'\ l') \mid (\varphi'\ l') \in L(r) \wedge SAT(\varphi \wedge \varphi' \wedge \varphi_g)\}$$

Again, the satisfiability check with the current path condition is a part of the process. The produced guarded value set is assigned to a new auxiliary reference, propagated to R and L.

`READVAL` and `WRITEVAL` are not mentioned in the original article, but we attempted to design them in a similar fashion as the previous operations. `READVAL` gathers all the possible constrained terms and joins them to a nested structure of the *ite* functions. `WRITEVAL` updates the current values so that they conditionally point either to the new term t or to the current term t_{cur}.

In Figure 3.3 are depicted the heaps produced during the analysis in Figure 3.1. The notation is based on Figure 3.2, but we can see certain differences. Each heap set contains only one heap and these heaps have a more complicated structure. References in circles were added and for every association from a reference to a location there is a condition attached, unless it is *true*.

The associations containing unsatisfiable conditions are hidden to save space. Furthermore, certain elements are not displayed in each heap, such as r_{null} and l_{null}. The rule for obtaining a path condition of a certain heap is the same as in the previous case. For the sake of simplicity, the heaps with unsatisfiable path conditions are not shown in the figure, such as the ones representing the error of dereferencing a null reference.

The `NEQ` operation initializes *this* to point via r_0^i either to l_0 or l_{null} and subsequently disables the latter. The following `ASSIGN` then associates *result* with r_0^i as well, producing H_2. Then, `READREF` creates two references r_0^i and r_0^s. The former appears during the initialization of *this.next*, referring either to l_{null}, aliased l_0 of a newly created l_1. To simplify the reference equality formulas, we use the following shortcut e_b^a:

$$\forall a, b \in \mathbb{N}\ (e_b^a \Leftrightarrow r_a^i = r_b^i) \qquad \forall a \in \mathbb{N}\ (e_n^a \Leftrightarrow r_a^i = r_{null}^i)$$

The stack reference r_0^s only copies the guarded value set from r_0^s and it is associated with hlp_0. `EQ` produces H_{p_1} by adding e_n^1 to its path condition, whereas H_4 contains its negation $\neg e_n^1$ due to `NEQ`. The two subsequent calls to `READVAL` cause $R(l_0, val)$ and $R(l_1, val)$ to be initialized to l_0 and l_1, respectively. The corresponding values of val_0 and val_1 are then v_0 and $ite(e_0^1, v_0, v_1)$. Therefore, the `ASSERT` producing H_{p_2} adds the following condition to φ:

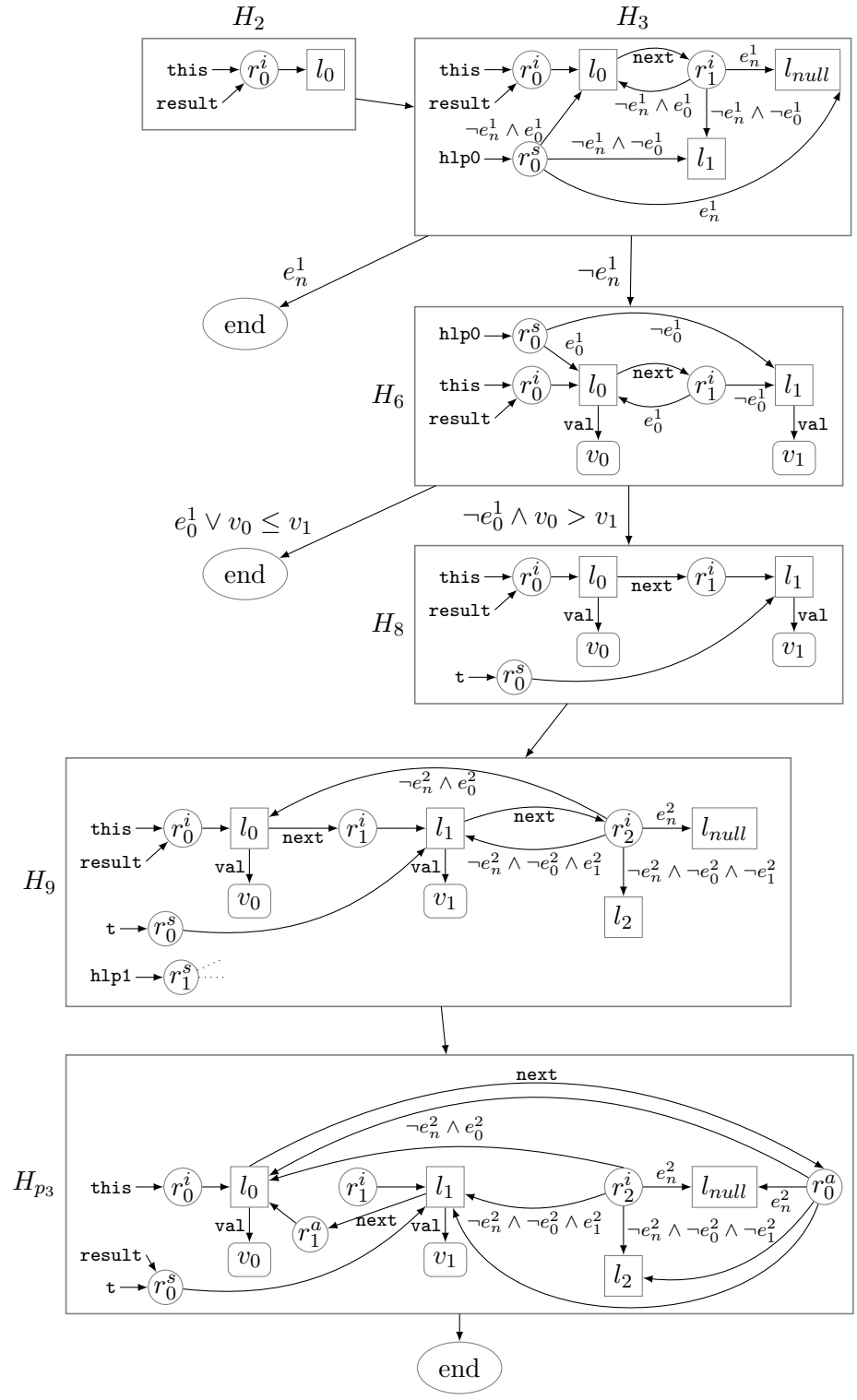
$$val_0 \leq val_1 \ \Leftrightarrow\ v_0 \leq ite(e_0^1, v_0, v_1) \ \Leftrightarrow\ (e_0^1 \wedge v_0 \leq v_0) \vee (\neg e_0^1 \wedge v_0 \leq v_1)$$

$$\Leftrightarrow\ (\neg e_0^1 \vee e_0^1) \wedge (e_0^1 \vee v_0 \leq v_1) \Leftrightarrow e_0^1 \vee v_0 \leq v_1$$

H_7 is produced with the negation of this formula, hence $\neg e_0^1 \wedge v_0 > v_1$. By the assignment of hlp_0 to t, H_8 is produced. As we can see in H_9, the further `READREF` initializes another reference, r_2^i. Its whole value set is, again, copied to r_1^s and assigned to $\eta(hlp_1)$. The two following calls to `WRITEREF` produce the auxiliary references r_0^a and r_1^a and put them to the appropriate places in the heap. No conditional write is involved in this case, because both locations are associated with their references without any conditions. Therefore, r_0^a and r_1^a just copy the corresponding value sets. Finally, `ASSIGN` produces H_{p_3} by assigning t to *result*. Again, the validity of the assertions can be inspected by reviewing the shapes of the resulting heaps combined with the respective conditions $v_0 \leq v_1$ and $v_0 > v_1$.

To summarize, symbolic initialization successfully tackles several disadvantages of the previous technique, such as the tremendous number of generated heaps. However, the constraints in guarded value sets can still grow polynomially. Furthermore, certain operations require a number of calls to the SMT solver, possibly consuming a lot of resources.

Figure 3.3: The usage of symbolic initialization for the analysis in Figure 3.1



3.3 Theory of Arrays Mapping

The approach of this technique is completely different from the previously mentioned ones. As its name suggests, it utilizes the theory of arrays to describe the shape of the heap. Therefore, it is not necessary to explicitly keep the information about the whole heap. Instead, the symbolic heap holds only the mapping of program entities to symbolic variables and all the relations between heap objects are propagated as constraints in the path condition. Let us begin with the formal definition of this type of symbolic heap:

$$h^A = (\varphi\ \eta\ \alpha\ l)$$

$$\eta : V_C \mapsto \Sigma_v \cup \mathbb{Z} \qquad \alpha : F \to \Sigma_v \qquad l \in \mathbb{N}$$

$$h_0^A = (true\ \eta_0\ \alpha_0\ 0) \qquad \forall c \in C\ (\eta_0(null_c) = 0) \qquad \forall f \in F\ (\alpha_0(f) = f_0)$$

In this case, the environment η only maps reference variables to integers, each resulting value can be either a constant or a symbolic integer variable. The field map α associates each field $f \in F$ with a variable of the array sort, whose key sort is integer and the value sort depends on the type of the particular field. If $f \in F_C$, it is integer as well, otherwise it is $t(f)$. These array variables will be used to access the values stored on the heap. Reading a value will be directly performed by using the *read* array operation, writing a value will require creating a new variable and assigning the result of *write* to it. To track the count of the manually created objects, we use an integer l. The initial heap h_0^A only maps the null variables to zero and all the fields in the program to the first versions of the corresponding arrays, having zero subscripts. A helper function to initialize input variables is defined below:

- $init(\varphi, \eta, v) = \begin{cases} (\varphi\ \eta), & v \in \eta^{\leftarrow} \\ (\varphi'\ \eta'), & v \notin \eta^{\leftarrow} \end{cases}$

$$r = fresh_{\Sigma_v}() \qquad \eta' = \eta[v \to r]$$

$$\varphi' = \varphi \wedge (r = 0 \vee (r < 0 \wedge \bigwedge_{f \in F_{t(v)} \cap F_C} read(f_0, r) \leq 0))$$

At first, the environment maps the program variable v to a newly created symbolic integer variable r. Then, we constrain r and all the first versions of the related reference fields to be less than or equal to zero. To understand why we need to do it, let us explain how we distinguish the input heap objects from those created during the analysis using `NEW`.

If we analyse any two references, both of which were initialized by a different call to `NEW`, we can see that they cannot be equal and neither of them can be null. As a result, a suitable way to model this behaviour is by assigning a unique integer constant to each reference created this way. This is simply done

by incrementing l with each call to `NEW` and assigning it to the given reference. The references contained in the newly created object cannot yet contain any valid references; therefore, we set them to zero, the value representing null.

On the other hand, if we need to read a value from a previously unknown reference, we cannot model it using a certain integer constant, because we do not know the exact shape of the input heap. However, it cannot be an arbitrary number, there are certain constraints involved. Namely all the references in the input heap can either be null or point to another object in the input heap. They cannot point to any objects created using `NEW`, because these objects did not exist during the previous execution of the program, where the input heap was created. Therefore, because we model null by zero and objects created during the analysis by positive integers, we can simply model the references from the input heap by negative integers. As a result, by setting these variables less than or equal to zero we satisfy the aforementioned constraints.

Having explained the reasons behind the overall design of the technique, the particular operations are defined as follows:

- $\texttt{NEW}((\varphi\ \eta\ \alpha\ l), v) = \{(\varphi'\ \eta'\ \alpha\ l')\}$

$$l' = l + 1 \qquad \eta' = \eta[v \to l']$$

$$\varphi' = \varphi \wedge \bigwedge_{f \in F_{t(v)} \cap F_C} read(\alpha(f), l') = 0$$

- $\texttt{EQ}((\varphi\ \eta\ \alpha\ l), v_1, v_2) = \{(\varphi''\ \eta'\ \alpha\ l)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v_1, v_2) \qquad \varphi'' = \varphi' \wedge (\eta'(v_1) = \eta'(v_2))$$

- $\texttt{NEQ}((\varphi\ \eta\ \alpha\ l), v_1, v_2) = \{(\varphi''\ \eta'\ \alpha\ l)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v_1, v_2) \qquad \varphi'' = \varphi' \wedge (\eta'(v_1) \neq \eta'(v_2))$$

- $\texttt{ASSERT}((\varphi_h\ \eta\ \alpha\ l), \varphi) = \{(\varphi'\ \eta\ \alpha\ l)\}$

$$\varphi' = \varphi_h \wedge \varphi$$

- $\texttt{ASSIGN}((\varphi\ \eta\ \alpha\ l), v, v_{in}) = \{(\varphi'\ \eta''\ \alpha\ l)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v_{in}) \qquad \eta'' = \eta'[v \to \eta'(v_{in})]$$

- $\texttt{READREF}((\varphi\ \eta\ \alpha\ l), v_{out}, v, f) = \{(\varphi_{err}\ \eta'\ \alpha\ l), (\varphi_{ok}\ \eta''\ \alpha'\ l)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v) \qquad \eta'' = \eta'[v_{out} \to fresh_{\Sigma_v}()]$$

$$\varphi_{err} = \varphi' \wedge (\eta'(v) = 0)$$

$$\varphi_{ok} = \varphi' \wedge (\eta'(v) \neq 0) \wedge (\eta''(v_{out}) = read(\alpha(f), \eta'(v)))$$

- $\texttt{WRITEREF}((\varphi\ \eta\ \alpha\ l), v, f, v_{in}) = \{(\varphi_{err}\ \eta'\ \alpha\ l), (\varphi_{ok}\ \eta'\ \alpha'\ l)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v, v_{in}) \qquad \alpha(f) = f_i \qquad \alpha' = \alpha[f \to f_{i+1}]$$

$$\varphi_{err} = \varphi' \wedge (\eta'(v) = 0)$$

$$\varphi_{ok} = \varphi' \wedge (\eta'(v) \neq 0) \wedge (\alpha'(f) = write(\alpha(f), \eta'(v), \eta'(v_{in})))$$

- $\texttt{READVAL}((\varphi\ \eta\ \alpha\ l), v, f) = \{((\varphi_{err}\ \eta'\ \alpha\ l)\ t), ((\varphi_{ok}\ \eta'\ \alpha\ l)\ t)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v) \qquad t = read(\alpha(f), \eta'(v))$$

$$\varphi_{err} = \varphi' \wedge (\eta'(v) = 0)$$

$$\varphi_{ok} = \varphi' \wedge (\eta'(v) \neq 0)$$

- $\texttt{WRITEVAL}((\varphi\ \eta\ \alpha\ l), v, f, t) = \{(\varphi_{err}\ \eta'\ \alpha\ l), (\varphi_{ok}\ \eta'\ \alpha'\ l)\}$

$$(\varphi'\ \eta') = init(\varphi, \eta, v) \qquad \alpha(f) = f_i \qquad \alpha' = \alpha[f \to f_{i+1}]$$

$$\varphi_{err} = \varphi' \wedge (\eta'(v) = 0)$$

$$\varphi_{ok} = \varphi' \wedge (\eta'(v) \neq 0) \wedge (\alpha'(f) = write(\alpha(f), \eta'(v), t))$$

In Figure 3.4 we can see the progress of the analysis from Figure 3.1. Each heap $h = (\varphi\ \eta\ \alpha\ l)$ is expressed as a rectangle displaying the mappings η on the left side and α on the right side. Because for the latter there are always exactly two fields *next* and *val* in the pre-image, only the corresponding array variables are displayed. Again, the path condition φ is constituted from the formulas attached to the arrows on the path from h_0 to h. The counter l is omitted there, because we are not using `NEW` in the example.

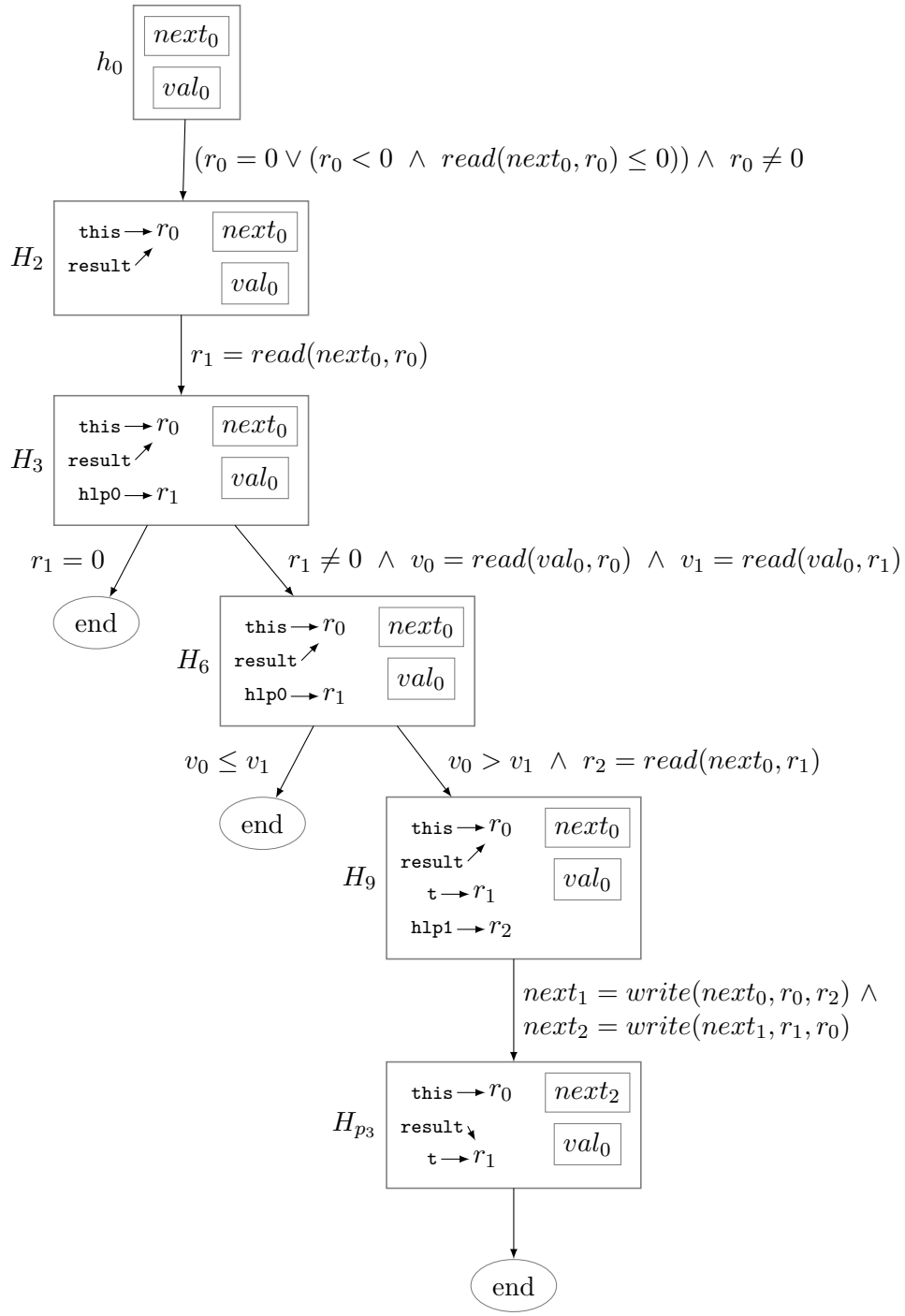
As shown in h_0, the fields *next* and *val* are mapped to the array symbolic variables $next_0$ and val_0, respectively. The initial environment η_0 only maps $null_{Node}$ to zero, which is not shown in the picture to save space. During the following `NEQ` operation, the variable *this* is initialized. As a result, it is mapped to a newly created symbolic integer variable r_0, which is constrained by the formula $r_0 = 0 \vee (r_0 < 0 \ \wedge \ read(next_0, r_0) \leq 0)$. The second constraint $r_0 \neq 0$ is produced by the main logic of `NEQ`. `ASSIGN` then adds only the mapping of *result* to the same variable that *this* is mapped to. `READREF` produces a new symbolic integer variable r_1 and sets it as equal to the expression $read(next_0, r_0)$. Note we skipped adding $r_0 \neq 0$, because it is already present in the path condition. The following branching by `EQ` and `NEQ` produces two expected conditions: $r_1 = 0$ is added to H_{p_1} and $r_1 \neq 0$ to H_4. To create H_6 from H_4, we call `READVAL` twice, resulting in the symbolic variables v_0 and v_1. H_{p_2} and H_7 are created by adding corresponding assertions formed from these variables. `ASSIGN` and `READREF` then produce H_9 in a similar way as they did it before with H_3. During the following calls to `WRITEREF`, the mapping of

next is updated twice, with the final version being $next_2$. The final `ASSIGN` only changes the mapping of *result* to r_1.

The validity of the assertions is not as simple to see as in the previous techniques, but the path conditions leading to breaking them will eventually be proven unsatisfiable. For example, the assertion `result != null` is valid, because both r_0 and r_1 are constrained to be non-zero.

In conclusion, the main benefit of this technique is the lack of the immediately visible complexity. It is moved from the heap itself to the SMT solver, which can be very useful, if it is powerful enough.

Figure 3.4: The usage of array theory mapping for the analysis in Figure 3.1



Chapter 4

Related Work

In this chapter we will mention some of the existing tools based on symbolic execution. Symbolic PathFinder [22], used for Java bytecode, combines symbolic execution with symbolic model checking [23], which, basically, attempts to describe all the possible states of the program and the transitions between them. The heap is modelled using lazy initialization, fitting nicely in this combination. The tool is capable of handling various complex mathematical constraints and multiple threads. Its reliability is substantial, as it is practically used by NASA.

KLEE [24] is a symbolic virtual machine utilizing the LLVM [25] infrastructure, used mainly for C and C++ projects. It can model the whole environment of complex system programs, including the networking and the file system. In the case of operations that cannot be modelled, it is possible to combine the symbolic values with the concrete ones. Its successful use case is the analysis of GNU Coreutils [26], resulting in high line coverage and the discovery of three serious bugs. Heap objects are modelled using the theory of arrays, which is also the case of the following tools.

An example of a concolic testing tool from the .NET world is Microsoft Pex [21], currently known as the IntelliTest feature of Microsoft Visual Studio 2017 Enterprise. When applied on a set of specifically annotated tests, it is able to create test inputs with high code coverage. It utilizes a library called Extended Reflection to observe the behaviour of the running program and provides this information to a symbolic interpreter in the background, which uses an SMT solver to create new inputs.

All the previously mentioned tools implement the forward variant of symbolic execution. The backward one can be found, for example, in a Java bytecode analysis tool called Snugglebug [27]. It utilizes the demand driven approach to remove false positives from the warnings produced by other tools and to infer the program specification. For the latter, it uses the ability of SMT solvers to simplify expressions. If the specification is already present, Snugglebug can be used to find test inputs that violate it.

Chapter 5

AskTheCode

In this chapter we will present AskTheCode, a "light-weight" assertion verification tool developed during the previous research [1]. The first section explains reasons behind its creation and possible scenarios of its usage. The next section focuses on its architecture, the used algorithms and their implementation. The last section outlines its future development, e.g. the support for heap objects.

5.1 Purpose

When developers want to make their effort spent on preventing and fixing bugs more effective, they have a plethora of static analysis tools available. These tools vary by their inner workings, reliability, efficiency and other factors, but have one characteristics in common - the more powerful they are, the smaller programs are they able to analyse and the more difficult is to include them in the software development process [1].

Symbolic execution, the technique this thesis is focused on, cannot be usually used to reliably find bugs in large programs or even to verify them, due to its caveats mentioned before. On the other hand, concolic testing is practically usable, but its efficiency is limited by the degree of the program testability. Furthermore, the tests created for concolic testing are often more difficult to prepare than the standard ones.

Apart from symbolic execution, there are other techniques to precisely analyse the semantics of the program, but they are either poorly scalable or require too many code adjustments in order to function properly. An example of the former is symbolic model checking, the latter group contains various techniques attempting to verify code properties, e.g. automatic theorem proving [28, 29].

As a result, the tools used in practice for larger projects usually contain some kind of abstraction, which enables them to work more efficiently, but makes them prone to producing a moderate amount of false positives. Es-

pecially popular are the techniques based on pattern matching, identifying various code constructs considered to be a bad practice. These tools produce a list containing all the places in the code that could cause errors and problems in readability. If the list is responsibly reviewed by the developers and the potential problems are fixed, it can indeed contribute to a more reliable and safer product. Unfortunately, this process is usually tedious and the amount of false positives is high. Therefore, for the majority of companies it is more effective to put more effort on discovering bugs by testing rather than preventing them using such tools.

Another problem is that the tools search mostly for general kinds of common errors, such as null pointer dereferences, memory leaks, uninitialized variables, deadlocks or security issues [30]. When it comes to the problems occurring in the domain of the particular software, they usually cannot help with them. For example, if a program returns an incorrect result with respect to the given input, the developers must inspect the semantics of the program on their own.

Therefore, we decided to create AskTheCode, a tool to simplify reasoning about the semantics of a program, searching for causes of errors and preventing them. It is implemented as a Microsoft Visual Studio extension for C#, enabling a programmer to select an assertion to be verified. If an execution path that disproves the assertion is found, it is displayed to the user. Otherwise, the tool either informs about a successful verification or inability to reason about it. It requires no prior setup and complicated integration in the development process. Instead, it is expected to be used on demand, whenever the programmer wants to verify the validity of a certain assumption about the program semantics or find an input causing a certain bug [1].

5.2 Approach

Due to its goal-driven approach, the tool is based on backward symbolic execution. It was created with extensibility in mind, resulting in loosely coupled components with distinct responsibilities, all written in C#. Refer to the previous work [1] for the complete description, here we will describe only the most important modules: *SmtLibStandard*, *ControlFlowGraphs*, *PathExploration* and *Vsix*.

The *SmtLibStandard* module provides a way to work with symbolic constants and expressions conforming to the SMT-LIB standard [11]. It contains various classes allowing to create expression trees in a strongly typed manner. Currently, `Int` and `Bool` sorts are implemented, together with the most of the related functions. Furthermore, a set of interfaces for SMT solvers and the models produced by them is available, implemented by the Microsoft Z3 solver [31] in *SmtLibStandard.Z3*.

ControlFlowGraphs, as its name suggests, contains a representation of a

symbolic *control flow graph* (CFG), which expresses the control and data flow in a method. Again, it provides an interface for particular programming languages, implemented only by C# in *ControlFlowGraphs.Cli*. At this time, the methods represented by the CFGs contain references between each other, but do not have any notion of classes. Therefore, only static methods can be modelled reliably at this point. Moreover, when it comes to branching and loops, only `if`, `else` and `while` clauses are supported.

In *PathExploration*, the main logic of the backward symbolic execution algorithm is captured. Given CFGs, an SMT solver and a starting point of the analysis, it performs the backward interprocedural traversal of the given program until it reaches an entry point. The exploration is highly configurable and extensible using various heuristics. They can influence, for example, in what order the code blocks should be traversed or how often to call the SMT solver along the way. The precise meaning of the "entry point" can also be configured. Currently, it is an entry statement of every public method in a public class.

The last mentioned module, *Vsix*, provides a user interface and the integration into Microsoft Visual Studio. It orchestrates the previously mentioned modules, attempting to answer the queries supplied by the user. Its role is also to display any found results in a readable manner.

5.3 Future Development

The ultimate goal of AskTheCode is to become practically useful. In order to achieve this, we need both to extend the scope of the supported code constructs and to implement various optimizations of the used algorithm.

Regarding the former, the key improvement is the support of heap objects, which will be implemented in this thesis. Important will also be the ability to reliably model the behaviour of more complicated entities, such as strings, collections and the environment. Particularly challenging is the support of advanced control flow principles, such as the exception catching, automatically created enumerators or asynchronous methods.

The most straightforward optimization is previously mentioned state merging. In fact, the path exploration algorithm already includes it, only the implementation of passing the merged paths to the SMT solver is missing. Parallelism, another potentially significant optimization, should not require too much effort, because the algorithm was designed to handle it as well.

Another approach targets both the quality of the results and the tool efficiency. Its main idea is to perform a global code analysis in advance and then use its findings to simplify the actual exploration. For example, we could discover that certain references are never null and always point to the same object, or that a particular field is immutable, leading to its simplified modelling. Moreover, some questions could be answered instantly, without

the need to call backward symbolic execution. Inevitably, this approach will lead to another problems, such as the extra time spent during it or the need to perform the analysis repeatedly whenever the code changes. Possible remedy can be to implement the analysis in an incremental way and cache its results.

Part II

Solution

Chapter 6

Requirements

In the first part of the thesis we have summarized backward symbolic execution and multiple options how to implement reasoning about heap objects in it. Furthermore, we have mentioned several practically used projects utilizing these techniques. Also the fundamentals of AskTheCode were presented, which is the tool we want to extend with heap functionality handling.

The second part focuses on the reasons behind various design decisions and on the implementation itself. In this chapter we will collect all the requirements stemming from various aspects of the expected solution. The first one are the constraints of a symbolic heap in backward symbolic execution in general. Then, we will mention certain specifics of AskTheCode which need to be considered when extending it. Finally, we will explain why is the performance of the final solution crucial and what to avoid in order not to impede it.

6.1 Symbolic Heap

While AskTheCode utilizes backward symbolic execution, all the previously mentioned symbolic heap implementation techniques were used in the context of the forward variant. Therefore, for each technique we must carefully consider whether it is possible to convert it while maintaining its soundness and efficiency. In the following text we inspect the particular heap operations and discuss how does their semantics differ when converted.

Starting with the most straightforward one, `ASSERT` does not change its semantics, as it serves only as a way to propagate constraints of scalar variables. The same applies to `EQ` and `NEQ`, if they are applied to the references already known to the symbolic heap. The reason is that it simply constrains their equality as it would do it in the forward variant.

However, when at least one of the references is currently unknown to the symbolic heap, the situation becomes a little more complicated. Consider the Listing 6.1 and suppose we start backward symbolic execution from the line

Listing 6.1: Sample C# code to demonstrate the differences between the forward and the backward variant of symbolic execution with respect to a symbolic heap

```
1  class Node {
2    public int val;
3    public Node next;
4
5    public static void Foo() {
6      Node b = new Node(10, null);
7      if (a == b) {
8        Debug.Assert(false);
9      }
10
11     Node c = b;
12     c.next = a;
13     c.val = -1;
14     c.val = 7;
15     int v = c.val;
16     if (v > 0) {
17       Debug.Assert(true);
18     }
19   }
20 }
```

8. When we first encounter the references a and b on the line 7 we do not know anything about them apart from being equal. Therefore, we assume they reference either null or an object from an input heap, as we would have done in the forward variant. However, in this case we must expect that any of such variables can be later discovered to be explicitly allocated using `NEW`, such as b on the line 6. As we can see, a comes from an input heap whereas b does not, hence $a \neq b$. Because of that, the path condition is unsatisfiable and the line 8 is proven to be unreachable.

To summarize, in any operation where we want to obtain a certain knowledge about a reference for the first time, we must consider the reference to be a part of the input heap but be prepared to change that state later. The role of the `NEW` operation is to perform this transition.

In general, working with references that have not been assigned to yet in backward symbolic execution resembles handling scalar variables in the same algorithm. Although we do not know where their values will be obtained from, we gather their constraints. When such variable is assigned to, we know that all these constraints apply to the assigned value as well. This principle will be naturally used in `ASSIGN` and we must utilize it also in the operations

`READREF`, `WRITEREF`, `READVAL` and `WRITEVAL`.

Let us consider running backward symbolic execution from the line 17 in Listing 6.1. At first, we encounter an integer variable v on the line 16 and constrain it to be greater than 0. Then, when reading the value from the heap on the line 15 we must somehow connect v with the particular place in the heap. On the line 14 the same place is assigned the value 7 and it is important that we are able to match this write with the previously mentioned read and extend the path condition so that it constraints $v = 7$ to be true. In fact, we are only interested in the situation when a non-zero number of reads to the same location is followed by one write. Two or consecutive writes to the same location are not interesting, because only the final value can be then read. Writing to a location that was not read before also does not help us with the analysis. We can see these unimportant writes on the lines 13 and 12. Finally, on the line 11 we use the `ASSIGN` operation. Everything we have to do in it is to ensure that the information stored about the locations accessible from c is transferred to b.

In the previous example of field writes and reads, we did not consider the possibility that the reference c might be null. This is due to the nature of backward symbolic execution. When trying to find whether a location in a program is reachable, we traverse all the operations leading to it. For each operation, we are interested only in the conditions that make it pass the control to the next one. Therefore, all the operations accessing objects on the heap through a reference guarantee that the corresponding reference were not null. In our example, if c was null on the line 12, the field write would have thrown an exception and the following statements would have not been reached. However, that is exactly opposite to what we are searching for, so we must constrain c not to be null.

If our aim is to explicitly check whether c can be null before it is dereferenced, we must take a different approach. In that case, the analysis should be started from the operation just before such usage and it should behave as if we were verifying an assertion $c \neq null_{Node}$. Note that in our example we would need to start such analysis before the line 12. The reason is that if c is null, the line 13 is unreachable due to its dereference on the line before.

To summarize, although the symbolic heap operations in backward symbolic execution can have the same names and syntax as in the forward variant, their semantics significantly differs. These discrepancies are what we need to carefully consider when transforming an existing symbolic heap technique from the latter to the former.

6.2 AskTheCode Integration

In this thesis, not only are we concerned with the symbolic heap itself, but we also want to integrate it in AskTheCode. This integration is not expected to be

straightforward, because it needs to be performed on many levels that influence each other, namely CFG construction, path exploration and execution model creation.

Currently, CFGs can directly contain only symbolic variables and expressions built upon them. Nodes can be of five types, the first four enable interprocedural navigation, e.g. method calling and returning from them. The nodes of the last type, *inner nodes*, contain sequences of assignments. Each assignment consists of a symbolic variable on the left side and a symbolic expression on the right side. Edges can be annotated with boolean symbolic expressions to denote conditional jumps.

There are multiple ways to incorporate heap operations to CFGs and each one should be carefully considered. For example, we can add all the possible heap operations to inner nodes or create helper variables to denote heap references and add them to the expressions. We must also design the best way to express possible null dereferences. Either we can add the respective exception throws explicitly or we might omit them, effectively postponing them to the path exploration phase.

Apart from these issues, we must also extend the algorithm which produces CFGs from C# methods. It can currently work only with the static ones, so we must implement also instance method and constructor handling. Processing particular heap operations can also be complicated, for example when a field access is nested.

Path exploration is the most elaborate phase, because backward symbolic execution itself is performed. Presently, it manages the versions of all the variables along the path and transforms all the edge conditions and variable assignments to a path condition. It is capable of various complex tasks, for example passing arguments and return values to and from called methods.

When adding a symbolic heap to a symbolic execution state, we will need to transform the so far mentioned theoretical interface to a set of particular C# interfaces that reflect the inner workings of AskTheCode. These interfaces should abstract as much from the control flow as possible and focus only on the heap operations.

The last integration step comes into question when a suitable path in a code is discovered and we want to display the corresponding heap and its changes to the user. There are certain constrains involved in producing such heap model. For example, the model produced by the SMT solver is available only when creating the heap model and not later, so we cannot utilize it in a lazy manner. Apart from producing the heap model and storing it in a certain data structure, it is also important to add its visual representation to the graphical user interface (GUI).

6.3 Performance

So far, we have described only the functional requirements on the solution, i.e. how to design and assemble its parts so that it can produce correct results. However, in order to be actually usable, the tool must be able to perform the analysis in a reasonable time. The biggest hurdle in this manner is the aforementioned path explosion problem, which is inherent to the character of the analysis.

It increases with each nondeterministic choice along the analysed program path. Currently, these choices comprise only from branching, cycles and method calls. However, heap operations can also add a degree of nondeterminism due to a strongly increasing number of various input heap shapes. Therefore, when implementing the symbolic heap we must attempt to approach this problem as sensibly as possible.

As we have already mentioned, one way to alleviate this problem is state merging. Although it is currently not implemented in AskTheCode, we want to implement it in the future. Therefore, when selecting a symbolic heap technique, we should consider whether it is suitable to be extended this way, i.e. being able to merge two or more symbolic heaps into one.

When not considering path explosion, there are other performance characteristics important for backward symbolic execution as implemented in AskTheCode. From performance profiling we can see that most of the execution time is spent in an SMT solver. That is not surprising, because all the other parts serve only to gather path conditions and cannot efficiently reason about them. On the other hand, state-of-the-art SMT solvers are optimized to handle long and complicated conditions consisting of terms from various theories [32]. Not only are they able to simplify input expressions and quickly find immediately visible violations, but they can also efficiently navigate through the structure of more complicated ones. Therefore, for a lot of problems it is often more beneficial to encode them into assertions in a path condition rather than to devise specialised algorithms for them.

Having said that, we may significantly impede the algorithm performance if we overuse the SMT solver by calling him more often than necessary. It is also useful if a symbolic heap can detect certain obvious conflicts in a sequence of operations even before the SMT solver is invoked.

Chapter 7

Architecture

In the previous chapters we have gathered all the information necessary to devise the final solution. Therefore, in this one we can present the resulting overall architecture and justify all the important design decisions. First, we will evaluate all the mentioned techniques with regard to the aforementioned requirements. Second, the transformation of the selected technique to backward symbolic execution will be explained. Last, we will focus on the changes in AskTheCode required for the successful heap integration.

7.1 Technique Selection

In this section we will analyse all the techniques explained before to evaluate their compliance with the requirements. There are three important areas of requirements: the suitability of the conversion from the original forward variant, special demands on the integration to AskTheCode and performance implications.

7.1.1 Lazy Initialization

The greatest benefit of lazy initialization is its simplicity and straightforward approach. The variant for forward symbolic execution captures an unequivocal shape of a heap, enabling to model the operations in a concise way. For example, each value field of a heap location can be modelled by a single symbolic variable. Value fields are, in fact, the only source of assertions to be added to an SMT solver, because all the reference operations can be handled directly.

With a moderate amount of effort it would be possible to transform the technique to work with backward symbolic execution. It would mean to slightly alter the meaning of the graph made of locations and references. Instead of representing a known part of a heap it would express the heap locations accessed by the program so far. As mentioned in the previous chapter,

we would need to observe the order of operations on each field and match the corresponding reads and writes. Furthermore, calling `NEW` would remove all the heaps where the respective location is targeted by an unassigned reference other than the one the operation is called on.

From the integration standpoint, this technique has one specific need. Currently, an exploration path uniquely describes how to reach a certain symbolic execution state in a program. The reason is that the only source of nondeterminism is the choice of an edge on each junction in a CFG. Therefore, we would need to distinguish two paths sharing the same sequence of CFG nodes and differing only by the symbolic heaps.

When implemented, the technique would work seamlessly on small samples where we would appreciate the low number of assertions added to path conditions. However, we are concerned about its performance on larger programs, because the increased number of branches would strongly contribute to path explosion. Moreover, as lazy initialization is built upon the idea of explicitly capturing a heap layout, merging two heaps would be basically an unnatural operation. To a certain extent, these problems can be alleviated by symbolic initialization, as we will see below.

7.1.2 Symbolic Initialization

In [18] is symbolic initialization presented as an improvement of lazy initialization that reduces nondeterminism by compressing multiple heaps into one structure. This is achieved by adding a condition to each reference within a symbolic heap. Analogically to the forward variant, we suppose it would be possible to devise a similar algorithm for the backward one. The conversion is not expected to be straightforward as in the case of symbolic initialization, but it would bring us all the benefits that the technique brings.

Furthermore, we would gain certain benefits on the integration side. The main reason is that we would no longer need to deal with the extension of nondeterminism. The only requirement specific for this technique is the ability to call an SMT solver, which is straightforward to include in the interface.

Regrettably, calling it too often may severely impede performance. As we can see, not only might each `READREF` and `WRITEREF` do it, but the number of calls also increases with the symbolic heap size. The reason for these calls is to remove unreachable access paths from the symbolic heap, making it smaller. Therefore, reducing the number of calls could possibly cause larger symbolic expressions to be produced. Furthermore, even if we prune the symbolic heap appropriately, larger programs might cause it to grow fast. Such situation is a hurdle in general, because it implies that even the problems easy to solve could be represented in a complicated and redundant encoding. For example, each `READVAL` in a large symbolic heap could produce a long nested chain of *ite* operators. In summary, symbolic initialization does not directly contribute to path explosion as lazy initialization, but the heap nondeterminism might have

an impact on the complexity of the produced symbolic expressions. Unlike the previous technique, we suppose that implementing state merging would be possible, thanks to the conditions contained in its structure.

7.1.3 Theory of Arrays Mapping

The biggest difference between this technique and the others is that we do not explicitly construct any shape resembling a heap structure. Instead, we use the heap theory constructs *read* and *write* to encode the operation semantics into symbolic expressions for an SMT solver. As a result, modifying the technique to work in backward symbolic execution does not require as much effort as symbolic initialization. We only need to keep in mind the rules mentioned in the previous chapter.

As far as the AskTheCode integration is concerned, there is an obvious demand to extend the used symbolic language with array theory expressions. Furthermore, the underlying SMT solver must be able to reason about them. We currently use Z3, which is highly capable in the scope of efficiently supported theories [31].

Performance-wise, this technique is not burdened by the main problems of the previous ones. It does not cause more path branching than currently present, as well as it produces only symbolic expressions of reasonable length in each operation. Having said that, it is important to note that the encoding is unable to simplify the nature of the problem. Instead, it just effectively transfers the complexity to an SMT solver. That is a justifiable approach, because they are optimized to tackle such difficulties efficiently, as known from their successful usage in analysing real-life applications [21, 24].

A subtle disadvantage of the lack of an explicit heap structure is the inability to eliminate SMT solver calls in the situations where immediately visible conflicts are encountered. We argue that this disadvantage certainly does not outweigh the previously mentioned advantages, as SMT solvers should be able to discover these conflicts quickly on their own. Furthermore, we can devise certain techniques to alleviate this problem, as we will see in the next chapter.

7.2 Symbolic Heap

As we can see, each mentioned technique has certain advantages and disadvantages. To implement the symbolic heap in AskTheCode correctly, any of them can be used if properly transformed. However, we want to ultimately make the tool usable for real applications, so we must strive to maximize the size of the possibly analysed programs. We can see that both lazy initialization and symbolic initialization are bound to have problems with longer paths containing many heap operations. Although the nature of the problem stays the same with array theory, it does not increase the number of SMT solver calls and we have a better chance of its optimizations to be involved.

Note that the expected benefits gained from the last mentioned technique are not founded on any exact data. The reason is that obtaining them would mean to implement all the techniques and then measure how they perform. Such comparison is beyond the scope of this thesis; therefore, we need to select one technique based on the aforementioned preliminary theoretical analysis. As a result, we decided to base the symbolic heap in AskTheCode on the theory of arrays. Having said that, we realize that it is not a good practice to couple the selected technique with the tool tightly and that we might want to implement other techniques in the future. Therefore, we add it via an interface, which can be later implemented by another techniques as well.

In this section we will explain the fundamentals of the resulting symbolic heap in a formal way, while the corresponding implementation details will be described in the next chapter. The structure is identical to the forward variant, we distinguish it only by adding the letter B to its name to state it is backward:

$$h^{AB} = (\varphi\ \eta\ \alpha\ l)$$

$$\eta : V_C \mapsto \Sigma_v \cup \mathbb{Z} \qquad \alpha : F \to \Sigma_v \qquad l \in \mathbb{N}$$

$$h_0^{AB} = (true\ \eta_0\ \alpha_0\ 0) \qquad \forall c \in C\ (\eta_0(null_c) = 0) \qquad \forall f \in F\ (\alpha_0(f) = f_0)$$

Again, the environment η maps reference variables to the union of integers and symbolic integer variables. Each field $f \in F$ is assigned a corresponding array symbolic variable in the field map α. Finally, l contains the count of the objects created using $\texttt{NEW}_{\leftarrow}$ so far. Note that also the initial heap h_0^{AB} is the same as h_0^A. The first difference we can see is in the helper *init* function:

- $init(\eta, v) = \begin{cases} \eta, & v \in \eta^{\leftarrow} \\ \eta[v \to fresh_{\Sigma_v}()], & v \notin \eta^{\leftarrow} \end{cases}$

The original function apart from mapping an unknown reference to a new symbolic variable also adds certain constraints to the path condition, assuring a valid input heap. In our case, however, we cannot add these constraints to the path condition permanently, because the reference can still be remapped to a positive integer value by $\texttt{NEW}_{\leftarrow}$ during the further exploration. Therefore, we will need to utilize the concept of *assumptions*. Simply speaking, an assumption is a condition temporarily added to an SMT solver for the immediately following solving, after which it is removed. In order to enable a symbolic heap to produce them, we added a simple operation $\texttt{GETASSUMPTIONS}_{\leftarrow} : H \to \Sigma_f$ to the interface. Below are defined all the operations of the symbolic heap implemented using the theory of arrays. To symbolize that their semantics corresponds to backward symbolic execution, we enhanced their names with the $\leftarrow$ symbol subscripts.

- $\texttt{NEW}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v) = \{(\varphi'\ \eta'\ \alpha\ l')\}$

$$l' = l + 1 \qquad \eta' = \eta[v \to l']$$

$$\varphi' = \varphi \wedge \varphi_e \wedge \bigwedge_{f \in F_{t(v)} \cap F_C} read(\alpha(f), l') = 0$$

$$\varphi_e = \begin{cases} l' = \eta(v), & v \in \eta^{\leftarrow} \\ true, & v \notin \eta^{\leftarrow} \end{cases}$$

- $\texttt{EQ}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v_1, v_2) = \{(\varphi'\ \eta'\ \alpha\ l)\}$

$$\eta' = init(\eta, v_1, v_2) \qquad \varphi' = \varphi \wedge (\eta'(v_1) = \eta'(v_2))$$

- $\texttt{NEQ}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v_1, v_2) = \{(\varphi'\ \eta'\ \alpha\ l)\}$

$$\eta' = init(\eta, v_1, v_2) \qquad \varphi' = \varphi \wedge (\eta'(v_1) \neq \eta'(v_2))$$

- $\texttt{ASSERT}_{\leftarrow}((\varphi_h\ \eta\ \alpha\ l), \varphi) = \{(\varphi'\ \eta\ \alpha\ l)\}$

$$\varphi' = \varphi_h \wedge \varphi$$

- $\texttt{ASSIGN}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v, v_{in}) = \{(\varphi'\ \eta'\ \alpha\ l)\}$

$$(\varphi'\ \eta') = \begin{cases} (\varphi \wedge \eta(v_{in}) = \eta(v)\ \eta[v \to \eta(v_{in})]), & \{v, v_{in}\} \cap \eta^{\leftarrow} = \{v, v_{in}\} \\ (\varphi\ \eta[\{v, v_{in}\} \to \eta(v_d)]), & \{v, v_{in}\} \cap \eta^{\leftarrow} = \{v_d\} \\ (\varphi\ \eta[\{v, v_{in}\} \to \eta(v_f)]), v_f = fresh_{\Sigma_v}(), & \{v, v_{in}\} \cap \eta^{\leftarrow} = \emptyset \end{cases}$$

- $\texttt{READREF}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v_{out}, v, f) = \{(\varphi'\ \eta'\ \alpha\ l)\}$

$$\eta' = init(\eta, v_{out}, v)$$

$$\varphi' = \varphi \wedge (\eta'(v) \neq 0) \wedge (\eta'(v_{out}) = read(\alpha(f), \eta'(v)))$$

- $\texttt{WRITEREF}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v, f, v_{in}) = \{(\varphi'\ \eta'\ \alpha'\ l)\}$

$$\eta' = init(\eta, v, v_{in})$$

$$\alpha(f) = f_i \qquad \alpha' = \alpha[f \to f_{i+1}]$$

$$\varphi' = \varphi \wedge (\eta'(v) \neq 0) \wedge (\alpha(f) = write(\alpha'(f), \eta'(v), \eta'(v_{in})))$$

- $\texttt{READVAL}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v, f) = \{((\varphi'\ \eta'\ \alpha\ l)\ t)\}$

$$\eta' = init(\eta, v) \qquad t = read(\alpha(f), \eta'(v))$$

$$\varphi' = \varphi \wedge (\eta'(v) \neq 0)$$

- $\texttt{WRITEVAL}_{\leftarrow}((\varphi\ \eta\ \alpha\ l), v, f, t) = \{(\varphi'\ \eta'\ \alpha'\ l)\}$

$$\eta' = init(\eta, v)$$

$$\alpha(f) = f_i \qquad \alpha' = \alpha[f \to f_{i+1}]$$

$$\varphi' = \varphi \wedge (\eta'(v) \neq 0) \wedge (\alpha(f) = write(\alpha'(f), \eta'(v), t))$$

- $\texttt{GETASSUMPTIONS}_{\leftarrow}((\varphi\ \eta\ \alpha\ l)) = \varphi_a$

$$fields(v_s) = F_C \cap \{f \in F_{t(v)} \mid \eta(v) = v_s\}$$

$$\varphi_a = \bigwedge_{v_s\ \in\ \eta^{\rightarrow}\ \cap\ \Sigma_v} v_s = 0\ \vee\ (v_s < 0\ \wedge \bigwedge_{f\ \in\ fields(v_s)} read(\alpha(f), v_s) \leq 0)$$

The changes performed in the operations correspond to the requirements of a symbolic heap in backward symbolic execution as summarized in the previous chapter. We can see that it was indeed not necessary to change $\texttt{EQ}_{\leftarrow}$, $\texttt{NEQ}_{\leftarrow}$ and $\texttt{ASSERT}_{\leftarrow}$, as their semantics is invariant to the direction of the analysis.

On the other hand, we had to add an optional equality condition $\varphi_e = (l' = \eta(v))$ to $\texttt{NEW}_{\leftarrow}$. The reason is that there can already be certain constrains imposed on $\eta(v)$ and by simply remapping v to a new integer value l' we would allow the SMT solver to create invalid heap models where $l' \neq \eta(v)$.

$\texttt{NEW}_{\leftarrow}$ is an example of a function that cannot be called twice on the same reference, because φ_e would instantly lead to a conflict in the path condition. Therefore, we need to set a rule that each reference can obtain its value via $\texttt{NEW}_{\leftarrow}$, $\texttt{ASSIGN}_{\leftarrow}$ or $\texttt{READREF}_{\leftarrow}$ only once. It is a natural requirement, because the core symbolic execution algorithm already manages versions of scalar variables and can extend it to reference variables as well. As a result, a full variable name will consist of an identifier and a version. Notice that this rule resembles static single assignment form (SSA) used in compilers for data-flow analysis, but it is much stronger, as multiple assignments on the same location in the program must not be to the same version of the corresponding variable.

Another operation coping with a similar problem as $\texttt{NEW}_{\leftarrow}$ is $\texttt{ASSIGN}_{\leftarrow}$. There are three possible cases differing by whether $\eta(v)$ and $\eta(v_{in})$ are already defined. The case when both are defined is the most complicated one, because we again need to force their equality so that the constraints put on each one are merged. Otherwise, we just ensure that $\eta(v) = \eta(v_{in})$, possibly by obtaining a fresh symbolic variable.

Because we do not need to reason about an exact heap structure, $\texttt{READREF}_{\leftarrow}$, $\texttt{WRITEREF}_{\leftarrow}$, $\texttt{READVAL}_{\leftarrow}$ and $\texttt{WRITEVAL}_{\leftarrow}$ were in fact only simplified due to the lack of possible null dereference errors during backward symbolic execution. Instead of producing an error state, they just constrain the corresponding reference not to be null. Notice also that in $\texttt{WRITEREF}_{\leftarrow}$ and $\texttt{WRITEVAL}_{\leftarrow}$, α and α' were swapped to match the backward semantics.

The aforementioned GETASSUMPTIONS$_{\leftarrow}$ operation simply replaces the work of the original *init*. As all the references mapped via η to symbolic variables refer to an input heap, we need to extend the path condition to secure that. For each field $f \in F$ the references within the input heap are represented by $\alpha(f)$ instead of f_0 as in the forward variant.

7.3 Integration

In this section we will explain the changes in AskTheCode needed to integrate the aforementioned symbolic heap in a high-level manner. Modification that are from this perspective too subtle or low-level will be mentioned in the next chapter.

7.3.1 Control Flow Graphs

The first important design decision was how to incorporate heap operations into CFGs. We started by selecting a representation for local reference variables. If we wanted to introduce reference variables as a completely new entity, we would need to reimplement the whole logic already present for scalar symbolic variables. They are already contained in a variety of places, e.g. assignments or method calls. Furthermore, in the path exploration algorithm the mechanism of properly handling their versions is implemented, enabling such advanced scenarios as recursive calls of the same method.

Therefore, reusing the infrastructure built for scalar symbolic variables is the most natural choice. However, not all the symbolic heap techniques really map reference variables to symbolic variables of a particular sort. As a result, we decided to create an artificial sort called `Reference` to be used for all reference variables. It is the task of the path exploration algorithm to translate the manipulation with them to respective heap operations, so that they are not directly passed to an SMT solver.

Thanks to this representation, we have successfully incorporated the ASSIGN$_{\leftarrow}$ operation. Also EQ$_{\leftarrow}$ and NEQ$_{\leftarrow}$ are straightforward to implement this way, because they correspond to the $=$ and $\neq$ conditions. Given that ASSERT$_{\leftarrow}$ and GETASSUMPTIONS$_{\leftarrow}$ are not handled on the CFG level, we need to choose how to represent the remaining operations, namely NEW$_{\leftarrow}$, READREF$_{\leftarrow}$, WRITEREF$_{\leftarrow}$, READVAL$_{\leftarrow}$ and WRITEVAL$_{\leftarrow}$.

We can see that except NEW, all of them can possibly cause null dereferencing, which we also had to decide how to approach. As already mentioned, in backward symbolic execution we can omit this option, so it would be sufficient to add conditions that restrict the corresponding references not to be null. However, we do not want to bind our CFG representation to a particular type of analysis, because in the future we might add other ones, such as forward symbolic execution or bounded model checking. As there might be

substantial differences in the ways how these techniques handle heap operations, we decided not to explicitly include null dereference exception throws. Another practical reason is that each CFG would significantly increase its size and complexity with every heap-related operation if we chose to do that.

Next, we will focus on the field read and write operations. Basically, there are two general approaches for both of them. Either we add another operation kinds to CFG inner nodes or we enhance assignments with a special kind of symbolic expression to represent a location in a heap. Choosing one over another does not have any dramatic implications, because we will need to implement their handling in the path exploration anyway. Eventually, we selected the first approach, because it is more explicit and convenient from the software design perspective.

Regarding instance methods, the semantics of their calling is almost the same as the one of static methods' with an additional instance parameter. The only difference is that high-level languages and runtime systems usually throw exception when this parameter is null. Therefore, it was sufficient to mark instance method call sites with a special flag so that the path exploration algorithm knows that the first parameter must be constrained not to be null.

To implement the `new` operator we added another call site flag to mark that memory allocation using $\texttt{NEW}_{\leftarrow}$ must be involved to set the value of the first parameter, ignoring the actually provided argument. This approach enables us to nest constructors of an inherited and a base class, where the former can be called with the memory allocation flag and the latter only with the instance method call flag. Note that while class inheritance itself is supported, virtual method calls are not, because we do not reason about object runtime types as of yet. Instead, each reference is strongly typed.

The extension of the algorithm that translates C# methods to CFGs was not much interesting from the high-level perspective. The changes made in the particular classes will be mentioned in the next chapter.

7.3.2 Path Exploration

In our previous work we described the core algorithm of our backward symbolic execution implementation in depth [1]. Here, we will only focus on the particular points where it was extended to handle a symbolic heap. In fact, to a significant extent we only need to illustrate how the design decisions made in the previously mentioned areas determined this algorithm.

As already mentioned, the algorithm maintains a version number associated with each symbolic variable, which is incremented after each assignment to that variable. It does not happen only in the case of assignments present in inner nodes, but also when an argument is passed to a method or a value is returned from it. The variables of the special `Reference` sort are no exception for these rules concerning versions. Their main difference is that instead

of expressing an assignment by an equality added to a path condition, the $\texttt{ASSIGN}_{\leftarrow}$ operation is called on a symbolic heap.

Furthermore, a reference variable's version must be incremented also in the cases when it is assigned a newly allocated object. This is done by calling $\texttt{NEW}_{\leftarrow}$ on an instance variable while processing a method call annotated with the corresponding flag. The instance method call flag causes $\texttt{NEQ}(h, v, null_{t(v)})_{\leftarrow}$ to be invoked for a reference variable v passed as the first parameter of such method, representing the instance.

Another situations in which $\texttt{EQ}_{\leftarrow}$ and $\texttt{NEQ}_{\leftarrow}$ are called is naturally when an edge condition is $v_1 = v_2$ or $v_1 \neq v_2$, respectively, for two reference variables v_1 and v_2. In the next chapter we will explain how to handle situations when the result of such expression is assigned to a boolean variable instead of being an edge condition.

Producing $\texttt{READREF}_{\leftarrow}$, $\texttt{WRITEREF}_{\leftarrow}$, $\texttt{READVAL}_{\leftarrow}$ and $\texttt{WRITEVAL}_{\leftarrow}$ is quite straightforward, because they are represented as operations in inner nodes. To distinguish between references and values it is sufficient to examine the sorts of relevant expressions.

$\texttt{ASSERT}_{\leftarrow}$ is called whenever a path condition is extended by a different source than a symbolic heap. Finally, $\texttt{GETASSUMPTIONS}_{\leftarrow}$ is used before an SMT solver is invoked. Note that the process of handling a heap condition is significantly simplified in this chapter. In reality, a path condition is held inside an SMT solver using an assertion stack, which can be manipulated using push and pop operations. In the next chapter we will see how we optimized the handling of a symbolic heap to match this semantics.

7.3.3 Model Creation

When a certain path in a program is proven to be reachable, an SMT solver produces a model, which contains interpretations for all the contained variables and their versions. From this information, an execution model is created by associating the interpretations to the appropriate nodes in the path. The used technique is basically a forward traversal through the path that associates each assigned variable with its interpretation from the model, decrementing its version. The execution model is then examined by the GUI to present the data to the user.

In order to introduce a symbolic heap into this mechanism, we need to create a heap model, i.e. a versioned heap structure graph whose node references can be stored in an execution model. The most straightforward way how to produce a heap model would be using a new symbolic heap operation accepting an SMT model. However, that would require the symbolic heap to remember all the operations performed on it. Furthermore, when state merging is introduced in the future, the symbolic heap would need the whole merged structure of operations. This would be an unnecessary duplication, be-

cause the structure is already captured in the path of the current exploration state.

Therefore, we used a concept of a *heap model recorder*, which is to be produced by the symbolic heap at the end of the exploration, hence at the start of the modelled execution. As the execution model creation progresses, the heap operations found on the path are executed on the recorder. Apart from creating a new version of the heap graph with each heap modification, it also returns the locations of the concerned references. As a result, these locations can be added to the execution model to represent the values of the corresponding reference variables.

Chapter 8

Implementation

After explaining the overall approach and reasons behind various design decisions, we will delve more deeply into the implementation itself. This chapter will mention all the important changes made to the particular AskTheCode modules on the level of class relations and responsibilities. For even more in-depth information please refer directly to the code and its changes recorded using Git.

8.1 SmtLibStandard and SmtLibStandard.Z3

The library containing the representation of SMT-LIB standard expressions had to be extended only slightly. As per the selected symbolic heap technique requirements, we added the support for array theory expressions. Moreover, when calling an SMT solver it is now possible to add a set of assumptions to it.

These changes were reflected also in the Z3 integration library. Because Z3 can only use literals as assumptions, we simulated the semantics by temporarily adding more complicated ones to an assertion stack and removing them afterwards.

8.2 ControlFlowGraphs

From the existing classes, the most influenced by the changes mentioned in the previous chapter were `CallFlowNode`, `InnerFlowNode` and `FlowGraphBuilder`. In Figure 8.1 we can see the relations between the newly added ones.
`IClassDefinition` and `IFieldDefinition` represent the items of the sets C and F mentioned earlier. Note that each field definition can be listed in more than one class definition, enabling to represent inheritance. The definitions of the `Reference` sort and the *null* variable are contained in the static `References` class, which is not displayed in the diagram.

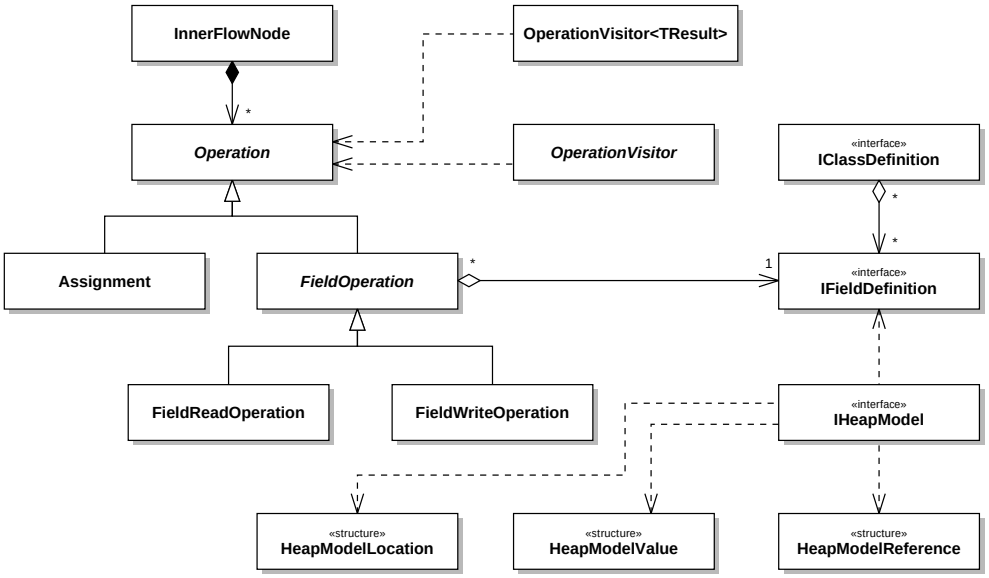


Figure 8.1: Diagram of classes newly added to *ControlFlowGraphs*

The `IHeapModel` interface and the related structures `HeapModelLocation`, `HeapModelReference` and `HeapModelValue` represent the composition of a heap model. A location contains not only an identifier of the corresponding node in the heap, but also its particular version. As a result, we can observe the changes performed on the individual locations over time.

`InnerFlowNode` now contains a list of various operations, represented by the abstract `Operation` class and its descendants. To simplify processing them, two visitors were introduced. They differ by the presence of return values in their methods.

8.3 ControlFlowGraphs.Cli

The structure of the newly added classes is depicted in Figure 8.2. Using the symbol information obtained from Roslyn, the implementations of `IClassDefinition` and `IFieldDefinition` are provided; the `TypeContext` class manages them.

We enhanced `ITypeModelFactory` to support creating value models from locations on a heap. This feature is then implemented using `ReferenceValueModel`, which is one of the classes implementing the abstract `ReferenceModel` class. The remaining ones represent reference variables in a code and null.

To take all the mentioned classes into account when processing C# code, we updated some of the existing classes. `CSharpGraphBuilder`,

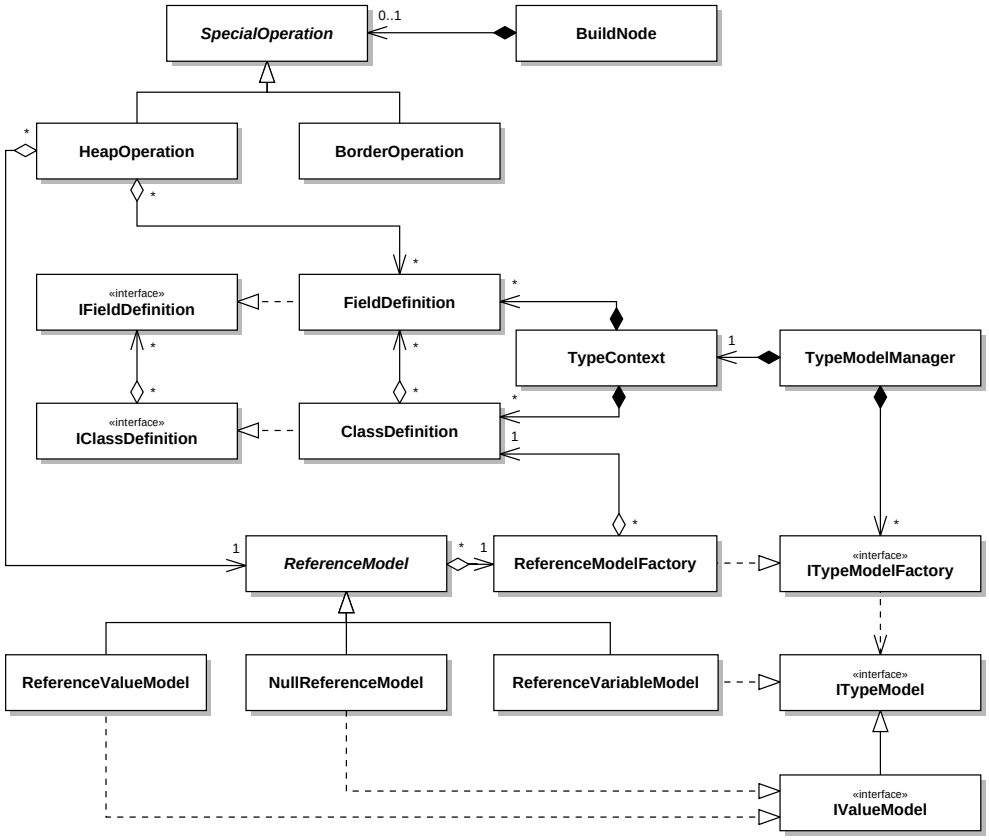


Figure 8.2: Diagram of classes newly added to *ControlFlowGraphs.Cli*

`StatementDepthBuilderVisitor` and `ExpressionDepthBuilderVisitor` are now able to process instance method declarations and heap operations. In order to mark these operations in intermediate graph nodes, the original `BorderData` class was changed to the abstract `SpecialOperation` class. Its implementations can express either interprocedural or heap operations. Non-trivial changes had to be performed also in `FlowGraphTranslator`, which serves to translate an intermediate graph to a CFG.

8.4 PathExploration

From the schema in Figure 8.3 is apparent the boundary between the path exploration algorithm and a symbolic heap implementation. It comprises four interfaces: `ISymbolicHeapFactory`, `ISymbolicHeapContext`, `ISymbolicHeap` and `IHeapModelRecorder`. Note that also `ISymbolicHeapModel` from *ControlFlowGraphs* is used. `ISymbolicHeapFactory` is responsible for creating instances of objects that implement `ISymbolicHeap` and accept an implemen-

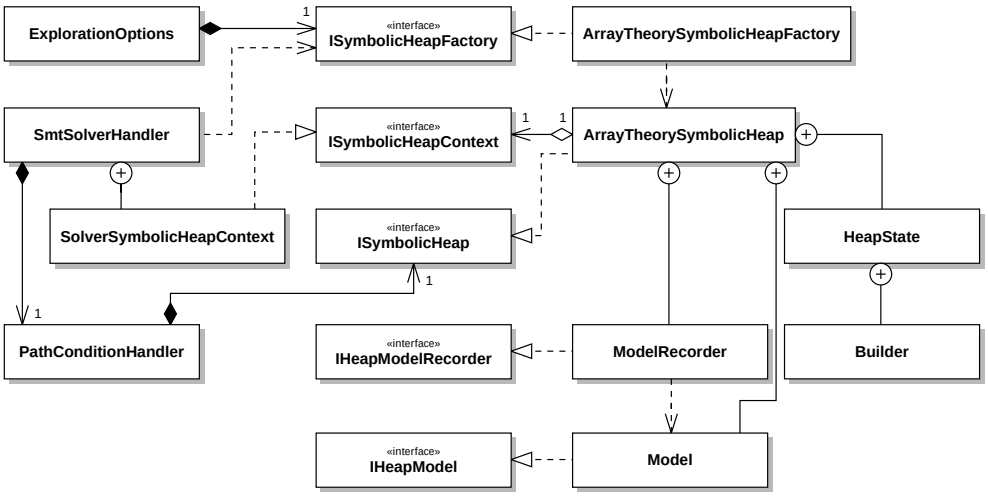


Figure 8.3: Diagram of classes newly added to *PathExploration*

tation of `ISymbolicHeapContext` for their construction. Using this interface, a symbolic heap calls certain operations on the overall algorithm, such as adding constraints to a path condition. `IHeapModelRecorder` serves to record all the heap operations happening on a certain execution path and to ultimately produce a heap model.

Because a single instance of symbolic heap is expected to be used among multiple paths differing by branching, we had to implement it in a way so that it is able to save its states along the way and return to them when necessary. The behaviour reminds the usage of an SMT solver assertion stack in the same algorithm. Therefore, we decided to implement it the same way in `ISymbolicHeap`. Using the `PushState` method, the heap saves its current state on a stack. When the `PopState` method is later called, the state popped from the top of the stack is restored.

On the path exploration side, a symbolic heap factory must be contained in `ExplorationOptions`. From there it is propagated to `SmtSolverHandler`, which creates an instance of symbolic heap and passes it to `PathConditionHandler`. It is then responsive for calling the appropriate heap operations met along the execution path and retracting them later. Any requests from the symbolic heap are processed by `SolverSymbolicHeapContext`.

The implementation of the aforementioned symbolic heap technique is present in `ArrayTheorySymbolicHeap` and its nested classes, some of which will be described here. The most important one is `HeapState`, because it contains an immutable snapshot of all the mappings, variables etc. Its `Builder` nested class is a mutable variant, which can be used to record any number of operations and produce another `HeapState` afterwards. These two

classes are based on .NET immutable collections, which provide effective operations and massive object sharing between multiple similar instances. `ArrayTheorySymbolicHeap` stores `HeapState` instances in the previously mentioned state stack and uses the builder to record the operations between two of these states.

`ModelRecorder` contains the logic of producing a heap model from an SMT model and a sequence of operations. The main idea is that we are only interested in the heap locations that were accessed at least once during the execution. The resulting heap model is stored in the `Model` class. Again, a .NET immutable dictionary is used to maximize memory reuse between different heap versions.

8.5 ViewModel

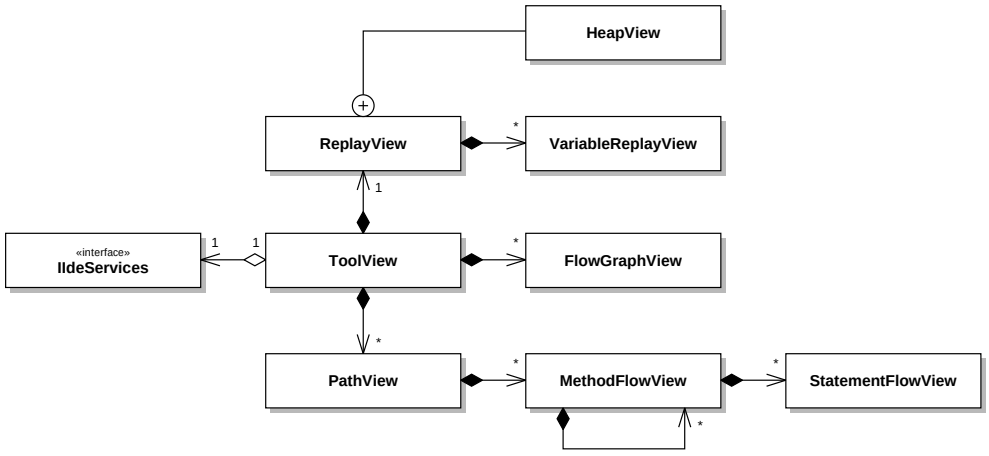


Figure 8.4: *ViewModel* class diagram

In Figure 8.4 is captured the full class diagram of this module. In comparison to the previous version, some of the classes were modified and three other were added. The most important change was performed in `MethodFlowView`. In order to make the navigation through an execution path more intuitive, we changed its structure to be displayed as a tree where each method's children represent the methods called by it.

Another attempt to improve user experience was adding a new panel, which enables to replay the currently selected execution model. One can navigate through it using commands known from debugging, such as "step into" or "step over". The panel displays the current values of all the local variables and also the overall heap shape. This logic is captured in the newly added classes: `ReplayView`, `VariableReplayView` and `ReplayView.HeapView`.

8.6 Other

A visual representation of aforementioned `ReplayView` is implemented using Windows Presentation Foundation (WPF) in the *Wpf* module as `ReplayPanel`. It is then directly used both from *Vsix* and *StandaloneGui*, which is a sample application to simplify GUI testing without the need of launching a full instance of Visual Studio.

ControlFlowGraphViewer, another sample application, was also updated to accommodate the changes performed in CFG construction and path exploration. Furthermore, it also displays a heap model when an execution model is found. The main contribution of this application is to ease debugging and together with the newly enhanced automatic tests it helps to evaluate the solution, as we will see in the next chapter.

Chapter 9

Evaluation

Having explained both the high-level and the low-level aspects of the solution, we will now proceed to evaluating whether the implementation actually fulfilled its purpose. The goal of this thesis was to extend AskTheCode with heap operation handling. To demonstrate that this functionality works well, we created a number of scenarios in the form of C# code. Firstly, we will describe their main characteristics and reasons behind them. Secondly, two helper applications that can give us a valuable insight into the algorithm will be presented. Lastly, we will summarize all the automated tests that are currently present in AskTheCode or are planned in the future.

9.1 Scenarios

In order to determine that a certain piece of software works correctly, it is usual to prepare a set of scenarios that it must be able to handle. The scenarios should cover as many of the theoretically possible situations as possible, focusing both on corner cases and common events. It is also often beneficial to craft some of the scenarios according to the inner workings of the software so that certain potentially problematic situations are handled.

In our case, we decided to prepare a large set of C# code samples of various complexity. Some of them exercise only small number of straightforward heap operations, whereas others use more complicated ways to induce certain situations. Two samples are provided in Listing 9.1. Notice that while the reasoning behind the assertions in the `SimpleComparison` method is easily understandable, the purpose of `DelayedComparison` is not immediately apparent. In this case, we want to demonstrate that the method `GetEqualityExpression` of `ISymbolicHeap` is implemented properly and also that there are cases where it is necessary. More detailed description of all the samples is provided directly in their code.

Where possible, we reused scenarios available from literature, such as Listing 3.1, which is presented in [19]. Regrettably, most of the benchmarks and

Listing 9.1: Sample C# scenarios containing heap operations

```
1  public static void SimpleComparison(Node a, Node b) {
2      if (a == b) {
3          Debug.Assert(a.next == b.next);
4      } else {
5          a.value = 5;
6          b.value = 10;
7
8          Debug.Assert(a.value != b.value);
9      }
10 }
11
12 public static void DelayedComparison(Node a, Node b)
13 {
14     bool wereEqual = (a == b);
15     a = new Node(0, b);
16
17     if (wereEqual)
18     {
19         Debug.Assert(a != b);
20     }
21 }
```

competitions regarding software verification usually approach the problem in a more low-level way, utilizing pointers [33].

9.2 Insight

By progressively evaluating the tool against the created scenarios, we were able to validate our expectations and discover several critical problems, which were subsequently repaired. For these purposes it is natural to utilize debugging capabilities of an IDE. Apart from that, we also used two specialized applications created just for this occasion.

The first one, called *ControlFlowGraphViewer*, serves to inspect the intermediate steps of building a CFG from a C# method, to run the core path exploration algorithm and to display its resulting execution model. The CFGs used for path exploration can also be obtained by programmatically constructing them, which was useful in the early stages of implementation when the C# code translation was not yet available. In Figure 9.1 we can see a screenshot of the application, enhanced with the heap model interactive display. We will not delve into the GUI, because it was not meant to be accessible to an end user; moreover, it is not as far as stable as AskTheCode itself.

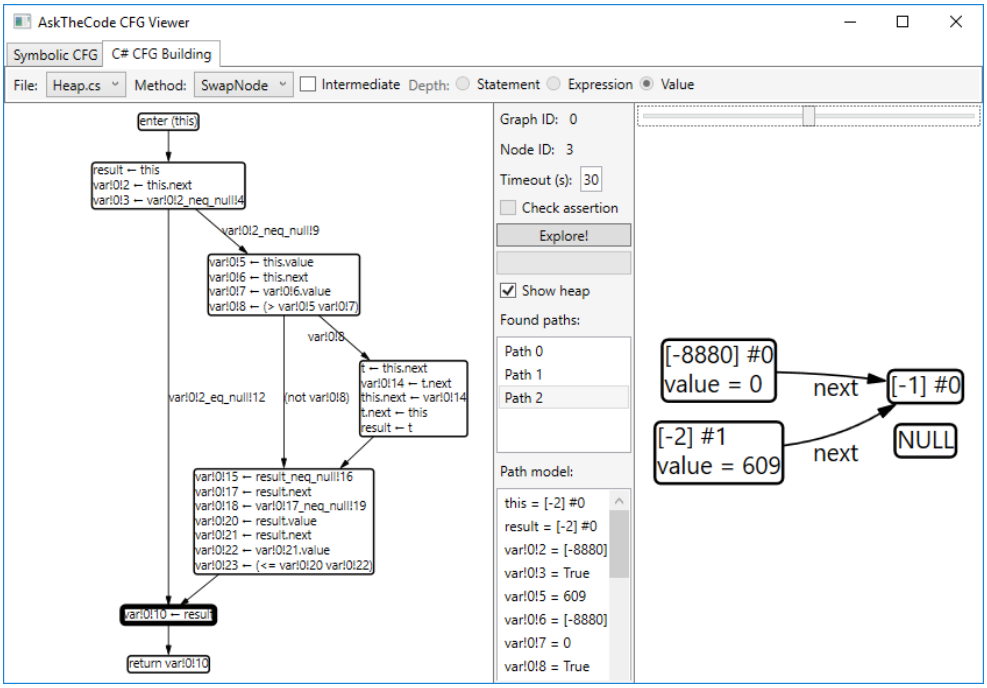


Figure 9.1: User interface of the *ControlFlowGraphViewer* application

Microsoft Visual Studio contains a useful way how to debug development versions of its extensions. It installs such extension to an experimental instance of itself and launches it with debugging attached. Although it allows to debug the extension in an authentic environment, it tends to be cumbersome because of the overhead involved in launching and operating the experimental instance. Therefore, we use an application called *StandaloneGui* to simulate the much more complex environment of Visual Studio in a simple WPF window, which can be seen in Figure 9.2 This approach enabled us to debug the GUI efficiently and minimized the need of using the experimental instance.

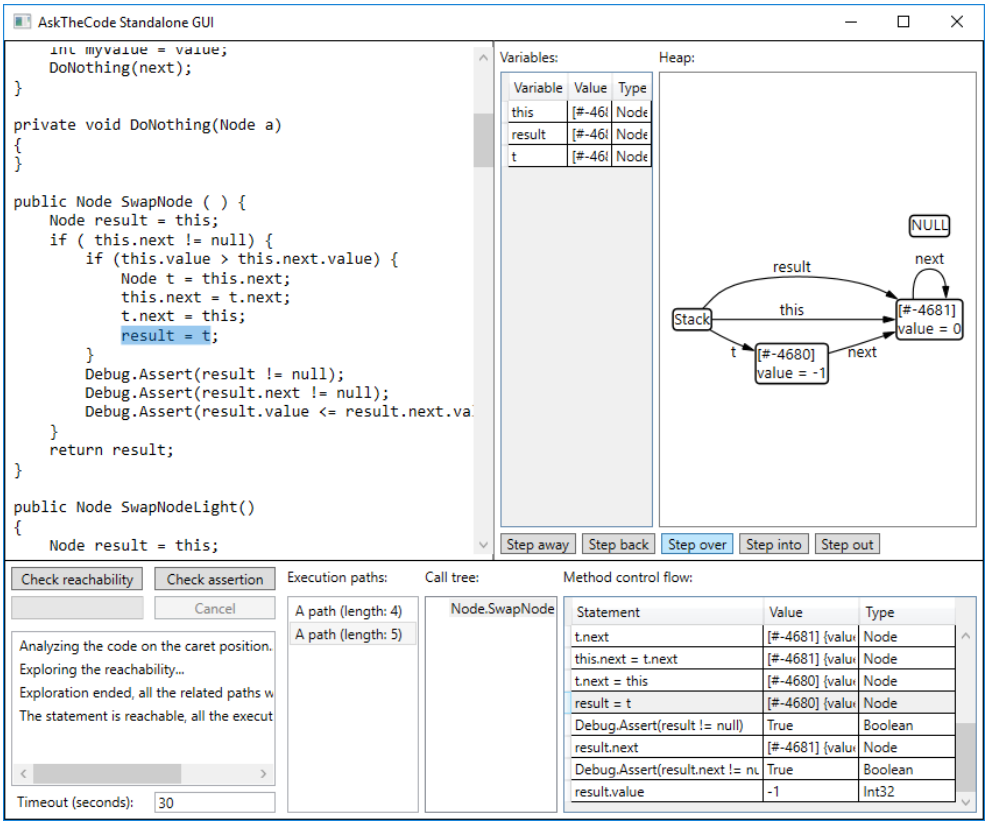


Figure 9.2: User interface of the *StandaloneGui* application

9.3 Tests

So far we have mentioned only the manual usage of the mentioned scenarios. However, their greatest benefit lies in the possibility to use them for automated testing. To comply with the common test terminology, they correspond mainly to acceptance tests, but can fulfil also the roles of integration and regression tests. From all the tests present in AskTheCode, they can cover the broadest range of situations and are the most straightforward to manage. The reason is that they are simply a collection of C# classes and we can simply modify and extend them. When a problem arises that breaks a test, we can use the previously mentioned applications to discover its exact source and fix it.

Regarding other tests in AskTheCode, there are also unit tests for the fundamental classes of *SmtLibStandard* and *ControlFlowGraphs*. We added also automated tests of C# code to CFG translation, which check that it throws no exception. In the future it might be beneficial to extend both the range of types and coverage of the particular tests, but we currently consider is sufficient with respect to the size and complexity of the whole project.

Some of the future tests will also include performance measurements in order to compare various algorithm optimizations. However, currently we focus mainly on extending the functionality of the tool, not on the overall speed. Having said that, it is possible to get a picture of the complexity of the tested scenarios by investigating the automated test results.

Conclusion

In our previous work, a static code analysis tool called AskTheCode was created. It is implemented as a Microsoft Visual Studio extension enabling to verify assertions in C# code using backward symbolic execution. Due to its limited capabilities it cannot be currently used in real projects, but our goal is to overcome this problem by progressively extending it.

The purpose of this thesis was to implement the capability of handling heap objects and operations upon them. We started by reminding the theory behind symbolic execution and SMT solvers. Then, three techniques of symbolic heap implementation which seemed the most promising were introduced, namely lazy initialization, symbolic initialization and the utilization of the theory of arrays. The remaining two chapters of the first part were dedicated to the existing tools that utilize some of these techniques and to the fundamentals of AskTheCode.

In the second part, we summarized all the requirements on the final solution divided into three categories: the specifics of symbolic heap in backward symbolic execution, integration challenges and possible performance hurdles. These requirements were later used to evaluate all the mentioned techniques and find the one that suits our needs the best. We selected to use the theory of arrays, mainly because it is expected to have the lowest performance footprint. Apart from the symbolic heap implementation itself, we described also various aspects of the entire solution, including the integration into the tool. The most important decision was to create a clean interface for a symbolic heap so that different techniques can be implemented later on.

To prove that our solution was designed and implemented properly, we prepared a large set of scenarios. Although they can be used manually, their greatest benefit is their usage as automated tests. Thanks to them, we can safely state that the goal of the thesis was successfully accomplished and AskTheCode made another step forward to being actually practically usable.

Bibliography

[1] Husák, R. Code Assertions Verification Using Backward Symbolic Execution. 2017, under the supervision of Jan Kofroň. Available from: https://dspace.cuni.cz/handle/20.500.11956/2087

[2] Biere, A.; Biere, A.; et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009, ISBN 1586039296, 9781586039295.

[3] de Moura, L.; Bjørner, N. *Satisfiability Modulo Theories: An Appetizer*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ISBN 978-3-642-10452-7, pp. 23–36. Available from: https://doi.org/10.1007/978-3-642-10452-7_3

[4] Davis, M.; Putnam, H. A Computing Procedure for Quantification Theory. *J. ACM*, volume 7, no. 3, July 1960: pp. 201–215, ISSN 0004-5411. Available from: http://doi.acm.org/10.1145/321033.321034

[5] Davis, M.; Logemann, G.; et al. A Machine Program for Theorem-proving. *Commun. ACM*, volume 5, no. 7, July 1962: pp. 394–397, ISSN 0001-0782. Available from: http://doi.acm.org/10.1145/368273.368557

[6] Nieuwenhuis, R.; Oliveras, A.; et al. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, volume 53, no. 6, Nov. 2006: pp. 937–977, ISSN 0004-5411. Available from: http://doi.acm.org/10.1145/1217856.1217859

[7] Kroening, D.; Strichman, O. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2016, ISBN 978-3-662-50496-3. Available from: https://doi.org/10.1007/978-3-662-50497-0

[8] Stansifer, R. Presburger's Article on Integer Airthmetic: Remarks and Translation. Technical report TR84-639, Cornell University, Computer Science Department, September 1984. Available from: http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639

[9] Goel, A.; Krstić, S.; et al. Deciding Array Formulas with Frugal Axiom Instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, New York, NY, USA: ACM, 2008, ISBN 978-1-60558-440-9, pp. 12–17. Available from: http://doi.acm.org/10.1145/1512464.1512468

[10] Stump, A.; Barrett, C. W.; et al. A decision procedure for an extensional theory of arrays. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, 2001, ISSN 1043-6871, pp. 29–37. Available from: https://doi.org/10.1109/LICS.2001.932480

[11] Barrett, C.; Fontaine, P.; et al. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available from: www.SMT-LIB.org

[12] King, J. C. Symbolic Execution and Program Testing. *Commun. ACM*, volume 19, no. 7, July 1976: pp. 385–394, ISSN 0001-0782. Available from: http://doi.acm.org/10.1145/360248.360252

[13] Sites, R. Branch resolution via backward symbolic execution. June 27 1995, uS Patent 5,428,786. Available from: https://www.google.com/patents/US5428786

[14] Kuznetsov, V.; Kinder, J.; et al. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1205-9, pp. 193–204. Available from: http://doi.acm.org/10.1145/2254064.2254088

[15] Godefroid, P.; Klarlund, N.; et al. DART: Directed Automated Random Testing. *SIGPLAN Not.*, volume 40, no. 6, June 2005: pp. 213–223, ISSN 0362-1340. Available from: http://doi.acm.org/10.1145/1064978.1065036

[16] Dinges, P.; Agha, G. Targeted Test Input Generation Using Symbolic-Concrete Backward Execution. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Västerås, Sweden: ACM, September 15-19 2014.

[17] Deng, X.; Lee, J.; et al. Efficient and formal generalized symbolic execution. *Automated Software Engineering*, volume 19, no. 3, Sep 2012: pp. 233–301, ISSN 1573-7535. Available from: https://doi.org/10.1007/s10515-011-0089-9

[18] Hillery, B.; Mercer, E.; et al. Exact Heap Summaries for Symbolic Execution. In *Verification, Model Checking, and Abstract Interpretation*, edited by B. Jobstmann; K. R. M. Leino, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, ISBN 978-3-662-49122-5, pp. 206–225.

[19] Khurshid, S.; Păsăreanu, C. S.; et al. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, Berlin, Heidelberg: Springer-Verlag, 2003, ISBN 3-540-00898-5, pp. 553–568. Available from: http://dl.acm.org/citation.cfm?id=1765871.1765924

[20] Bjørner, N. Engineering Theories with Z3. In *Programming Languages and Systems*, edited by H. Yang, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 978-3-642-25318-8, pp. 4–16.

[21] Tillmann, N.; De Halleux, J. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 3-540-79123-X, 978-3-540-79123-2, pp. 134–153. Available from: http://dl.acm.org/citation.cfm?id=1792786.1792798

[22] Păsăreanu, C. S.; Visser, W.; et al. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, volume 20, no. 3, 2013: pp. 391–425, ISSN 1573-7535. Available from: http://dx.doi.org/10.1007/s10515-013-0122-2

[23] Clarke, E. M. *Model checking.* Cambridge : MIT Press, c1999, 1999, ISBN 0-262-03270-8. Available from: http://search.ebscohost.com/login.aspx?authtype=shib&custid=s1240919&profile=eds

[24] Cadar, C.; Dunbar, D.; et al. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. Available from: http://dl.acm.org/citation.cfm?id=1855741.1855756

[25] Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed*

and Runtime Optimization, CGO '04, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2102-9, pp. 75–. Available from: http://dl.acm.org/citation.cfm?id=977395.977673

[26] MacKenzie, D. e. a. *GNU Coreutils*. 2017. Available from: https://www.gnu.org/software/coreutils/manual/coreutils.pdf

[27] Chandra, S.; Fink, S. J.; et al. Snugglebug: A Powerful Approach to Weakest Preconditions. *SIGPLAN Not.*, volume 44, no. 6, June 2009: pp. 363–374, ISSN 0362-1340. Available from: http://doi.acm.org/10.1145/1543135.1542517

[28] Lal, A.; Qadeer, S.; et al. A Solver for Reachability Modulo Theories. In *Computer Aided Verification*, edited by P. Madhusudan; S. A. Seshia, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN 978-3-642-31424-7, pp. 427–443.

[29] Fähndrich, M. Static Verification for Code Contracts. In *Static Analysis*, edited by R. Cousot; M. Martel, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-15769-1, pp. 2–5.

[30] Coverity® Scan Open Source Report 2014. 2015. Available from: http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf

[31] De Moura, L.; Bjørner, N. 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, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 3-540-78799-2, 978-3-540-78799-0, pp. 337–340. Available from: http://dl.acm.org/citation.cfm?id=1792734.1792766

[32] Cok, D. R.; Déharbe, D.; et al. The 2014 SMT Competition. *Journal on Satisfiability, Boolean Modeling and Computation*, volume 9, 2014: pp. 207–242. Available from: https://satassociation.org/jsat/index.php/jsat/article/view/122

[33] Beyer, D. Software Verification with Validation of Results. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, New York, NY, USA: Springer-Verlag New York, Inc., 2017, ISBN 978-3-662-54579-9, pp. 331–349. Available from: https://doi.org/10.1007/978-3-662-54580-5_20

Appendix A

AskTheCode User Documentation

In this appendix we will show how to install and use AskTheCode. Furthermore, we will also explain its current limitations.

A.1 Installation

The following requirements must be met in order to perform the installation:

- Microsoft Windows 7 SP 1 or newer
- Microsoft .NET Framework 4.6.1. or newer
- Microsoft Visual Studio 2017 of version 15.6.5 or newer
- Microsoft Visual C++ Redistributable for Visual Studio 2012

The installer of the Visual Studio extension is located in the `bin/AskTheCode.vsix` file on the enclosed CD. An alternative way to try AskTheCode without installing it to the IDE is to launch the standalone GUI application from `bin/ControlFlowGraphViewer/ControlFlowGraphViewer.exe`. Keep in mind, however, that this tool is mainly meant for debugging and therefore it is not as user-friendly and stable as the extension.

A.2 Usage

For the first usage of AskTheCode we recommend to use the samples from the enclosed CD. It is useful to copy them to a local folder so that it is possible to modify them. The directory path on the CD is `test/inputs/csharp` and it contains two projects. The `Sample` project consists of a low number of small code examples, whereas the samples in the `EvaluationTests` project cover the

functionality more systematically. When using AskTheCode to verify custom code, remember to check the limitations listed in the next section.

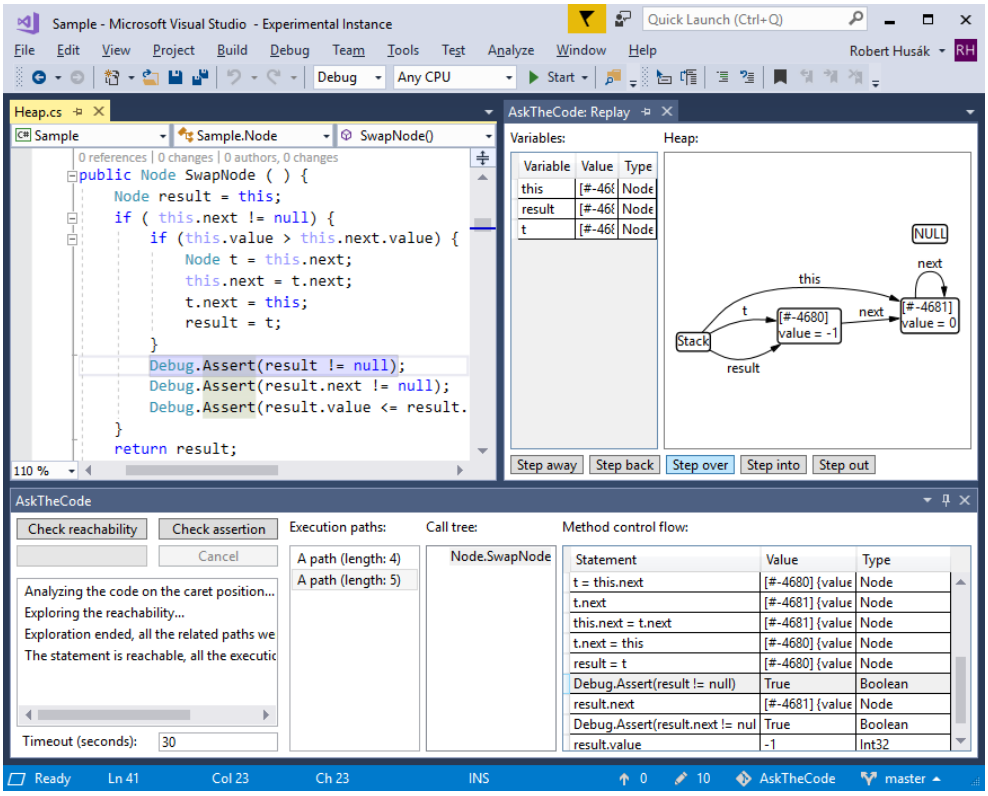


Figure A.1: AskTheCode in Microsoft Visual Studio 2017

The extension adds a command named *AskTheCode* to the *Tools* menu. Clicking on it will display the main panel, as seen in Figure A.1. In its top left corner we can wee two buttons: *Check reachability* and *Check correctness*. As their names suggest, the former simply attempts to find any execution path leading to the currently selected statement, whereas the latter is concerned only with the paths that break the currently selected assertion. Note that they consider only public methods of public classes as the possible entry points of the execution paths.

When a path exploration is started, its progress and intermediate results are displayed in the panel below the mentioned buttons. It is possible to explicitly cancel the exploration by the corresponding button, otherwise it is terminated automatically when the specified time limit elapses, if it is unable to finish earlier.

If any execution path is find during an exploration, it is shown in the *Found paths* list. When a path is selected from the list, its call tree is dis-

played alongside the statements encountered during the execution of of these methods. Figure A.1 shows also a replay panel, which allows us to traverse the currently selected execution path as if it were a program being debugged. For each point in the replay we can see the current values of all the local variables and the state of the interesting objects on the heap. To navigate throughout the replay, we can use the buttons known from debugging: *Step over*, *Step into* and *Step back*. In addition to that, there are also two operations enabling us to travel back in time. *Step back* moves us to the previous statement and *Step away* brings us back to the call site of the current method.

A.3 Limitations

It is important to remember that although by supporting heap objects we have improved the usability of AskTheCode in a significant way, it is still far from being completed. It supports only a subset of C# language constructs, powerful enough to create tests but not wide enough to be practically usable yet.

From the basic types, only `bool` and all the integral types are supported, whereas the latter are modelled using unbounded integers. Although this approach violates the soundness, it is a common trade-off used among static analysis tools. We can use the mentioned types in expressions created from these operators: `!`, `&`, `&&`, `|`, `||`, `==`, `!=`, `<`, `<=`, `<`, `>=`, `+`, `-`, `*` and `/`. Division is not modelled precisely, because it is not sound when a divisor is zero.

AskTheCode can model calls of both static and instance methods, but they must not be virtual and the parameters cannot be passed by reference. It can handle classes with their non-static fields, modifying them via references and calling their constructors. However, it cannot reason about their runtime types, so the operators like `is`, `as` or explicit casting cannot be used. The support of inheritance is only experimental and not fully completed.

The statements used in the methods are limited to assignments, other method calls, `return`, `throw new`, `if`, `else` and `while`. Note also that the tool cannot handle loops and recursion in a graceful way and will usually get stuck in an infinite loop.

Appendix B

Contents of enclosed CD

- `.git` ..... Git version control system folder
- `bin` ..... executables
  - `bin/ControlFlowGraphViewer` ..... CFG viewer application
  - `bin/StandaloneGui` ..... standalone GUI application
  - `bin/AskTheCode.vsix` ..... AskTheCode extension
- `lib` ..... external libraries
- `samples` ..... source code of evaluation applications
- `src` ..... AskTheCode sources
- `test` ..... AskTheCode tests
  - `test/inputs/csharp/EvaluationTests` ..... C# evaluation code
  - `test/inputs/csharp/Sample` ..... C# sample code
- `text` ..... thesis text
  - `latex` ..... thesis LaTeX source code
  - `thesis.pdf` ..... thesis text in PDF format
- `.gitignore` ..... files and folders ignored by Git