Debugging Symbolic Execution

Master's Thesis Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

By: Ivo Colombo
icolombo@student.ethz.ch

Supervised by: Malte Schwerhoff
Prof. Dr. Peter Müller

Date: September 1, 2012

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Formal software verification is an evolving field of research. Enormous progress has been made in the areas of expressiveness of specification, power of underlying theorem provers or framing techniques to prove correctness of concurrent programs with shared mutable state. Still, formal verification has not yet found its way into everyday software engineering.

Our goal is to accelerate this ongoing development by providing tool support for the debugging of verified software. We have developed a concept for a powerful debugging environment for the language Chalice using the symbolic execution verifier Syxc. Also, we have implemented a debugger tool on the basis of Eclipse which implements parts of our concept.

Contents

List of Figures

List of Listings

1 Introduction

1.1 Overview

For decades, scientists have been working on the possibility of programs to be proved correct. Today, research in the field of formal software verification has come close to realizing this vision. It is now possible, with some caveats, to feed a program text to a static, formal verifier and automatically receive a formal proof of correctness - if the verifier is able to find one. Why is not every piece of software being verified nowadays?

Firstly, verification is still on the very edge of technology. Verifiers are still rather weak, that is, many everyday programs rely on properties that verifiers are too weak to reason about.

Secondly, the level of effort and expertise needed to produce verified software is very high. The reason for this is specification: Even if a verifier could produce evidence of any property a program could have, one still would need to define precisely what these properties are for the program in question. This has to be done in a way the verifier can interpret it, that is using a mathematical language with defined formal semantics, called a *specification language*. Writing good specifications for a program with current specification languages is still at least as labour-intensive as writing the program itself.

These limitations cause verification to be unfeasible for most software. Today, it is applied only in situations where the need for software that can be relied on is particularly high. This is the case for high-risk software, for example in safety critical contexts.

In this master's thesis we will try to contribute to easier and more efficient use of verifiers and thus both facilitate research in software verification and lower the bars for verifiers to be used in a broader range of contexts. We will do this by providing a tool that helps developers to debug programs and, more importantly, program specification. Tool support for *verification debugging*, as we call the fledgling discipline of systematically finding and solving problems in programs to be verified and their specification, has been introduced most prominently by VeriFast [JSP10]. While VeriFast includes both a verifier and a graphical user interface with debugging features, we focus on the latter and rely on the verifier Syxc [Sch11]. Syxc verifies programs written in the language Chalice [LM09], which includes both programming and specification language elements.

The following sections briefly introduce the workings of software verification in general, symbolic execution as our main approach to software verification and

finally the Chalice language and Syxc which provide concrete implementations of the former techniques.

1.2 Software verification

The term *software verification* or, more precisely, *static formal software verification* describes techniques to reason about properties of programs. These properties are specified in the form of assertions called *contracts*, including preconditions, postconditions and invariants. For example, functions are specified with a precondition, determining the requirements that must be met in order for the function to be safely executed, and a postcondition, stating the guarantees the function gives after it was executed. Invariants are used to specify non-varying properties. For example, class invariants specify properties that must hold for all instances of the class at all times[1]. Another example of invariants are loop invariants. They describe the non-varying state of loops, that is a property that must hold before and after every iteration.

The task of a verifier is to prove certain correctness criteria using the specification of the program to be verified. These criteria are, for example: Assuming the precondition of a function holds, the function body must establish the postcondition. Or: The invariant of a loop must be preserved by its body.

To be able to perform these proofs, not only the specification must have formal semantics but also the program code. For example, a variable assignment changes the program state in that the variable points to a new value after the assignment, but everything else remains unchanged. The definition of the semantics of statements like the assignment statement is done in the form of *inference axioms and rules*. Hoare [Hoa69] originally defined a set of basic inference axioms and rules including the composition rule. The latter is used to reason about a sequence of statements instead of just one. This is the basis to allow to reason not only about a single statement, but, for example, about a whole function body.

Now, there are two ways how to perform the proof: Knowing the state before a (composed) statement (the *pre-state*), the verifier can reason about the state after the statement (the *post-state*). This is known as *forward reasoning*. The *strongest postcondition* of the statement given its precondition is computed and can then be compared to the specified postcondition. If the computed postcondition is stronger than the specified one, we have proof for the program to fulfil its specification. *Backward reasoning* works as follows: Knowing the post-state, the verifier can reason about how the pre-state must have looked in order for the statement's state transition to result in the post-state. This time, we compute the *weakest precondition* (denoted by $WP(STMT, POST)$) of the statement ($STMT$) given its postcondition ($POST$). If the computed precondition is weaker than the specified one, we have proof that the program is correct. This technique is also known

[1] Usually, class invariants are guaranteed to hold before a method is executed and must hold again only after the method returns.

as the *weakest precondition calculus* described by Dijkstra in [Dij76]. [GC10] gives a more detailed comparison between forward and backward reasoning.

The two verification techniques used in most of today's verifiers are based on the two ways of reasoning, respectively: *Verification condition generation* (VCG) uses the backwards approach. For each function or method of a program, it computes the weakest precondition of the body given its postcondition. The verification condition, $PRE \Rightarrow WP(BODY, POST)$, is the correctness criterion of the function/method, where PRE denotes the specified precondition. This verification condition contains all information that is needed to infer the postcondition from the precondition and is solved by a single call to a theorem prover. A very similar approach could also be imagined using forward reasoning and a verification condition using the strongest postcondition. As stated in [GC10], the computation of the weakest precondition is simpler, and thus it is preferred.

Nonetheless, *symbolic execution* (SE) uses a forward approach, but with major differences to VCG. Since our work is based on symbolic execution, it is discussed in more detail in the next section.

Framing An important issue for the verification of object-oriented programs with mutable state is *framing*. Postconditions of methods define guarantees on their post-state. Callers use these guarantees to verify their own body. In order to do this, it is particularly important for them to know exactly which heap locations were changed by the callee. Without this knowledge the caller could not rely on its previous knowledge of any heap location's value, because it could have been changed in the meantime by the callee. Framing is a general concept to avoid this. Every method includes information in its contract defining an upper bound on the set of heap locations it depends on. Examples of framing techniques are *separation logic* [Rey02] or *implicit dynamic frames* [SJP09a]. Both of them consider heap locations as protected from any access by default, only allowing it if a method has the corresponding permissions. Method can require access permissions by including an appropriate access predicate in their precondition and return them to the callee using the postcondition. This way, reasoning within a method body can be done by looking only at a partial heap, that is the part for which access permissions are available.

Data abstraction Both *separation logic* and *implicit dynamic frames* introduce the concept of predicates[2] as a means of data abstraction into their approach. Predicates define an assertion they abstract from. In turn, they can be used in assertions again.

[2]Not to be confused with access predicates as a part of assertions

1.3 Symbolic execution

Symbolic execution was first described by King in 1976 [Kin76]. His main idea was to represent (infinitely many) input values of a program by a symbolic value and thereby to overcome the problem of not being able to test a program with infinite input state. The symbolic execution of a method thus maintains a symbolic state that is modified by each statement. At the end of the method's body, the computed symbolic state represents the strongest postcondition of the method body, given its precondition. If this state satisfies the specified postcondition, the method is successfully verified.

If-then-else statements in the code need special treatment in symbolic execution, because they split the knowledge base into two branches: One branch can assume the condition to be true and continue the execution[3] with the *then*-branch. The other branch assumes the condition to be false and continues with the *else*-branch. In symbolic execution, unlike in abstract interpretation, the execution is usually not merged after both branches have been executed, but both branches execute the remainder of the method. This way, every path through a method body is executed independently. All the knowledge collected during the execution of a path is contained in the so-called *path condition*.

This approach is different from VCG in that the theorem prover is not called just once with a large formula to solve, but it is called very frequently, adding or asserting knowledge in the form of small formulae. Refer to [KMS12] for a more detailed comparison between symbolic execution and VCG.

Symbolic state [SJP09b] describes an approach for symbolic execution with implicit dynamic frames. The symbolic state is composed by a store of local variables, a current heap, an old heap and the path condition. The heaps consists of *chunks*, of which two basic types exist: *Field chunks* represent heap locations for which access permissions are currently present. *Predicate chunks* represent the assertion of their underlying predicate, which may again contain access permissions to heap locations. Also, predicate chunks require permissions to be accessed.

Heap chunks in the symbolic state are directly managed by the verifier and not by the theorem prover. Heap properties (e.g. existence of access permissions to a certain heap location) can therefore often be established even without calling the theorem prover. [KMS12] discusses advantages and drawbacks of the separation of heap and path condition.

Symbolic values are represented by *terms*. For example, in a symbolic state, every local variable is assigned a term representing its value. Similarly, field chunks are assigned a term representing the value of the corresponding heap location. Terms can be literals, for example integers, but they can also be just symbols, representing an unknown value that is possibly restricted by the path condition.

[3] In this document, whenever we talk about execution we refer to symbolic execution unless otherwise specified.

E.g. the term $t_1$represents an unknown integer and the path condition contains the formula $t_1 > 100$, restricting t_1 to values greater than 100.

The distinction between current and old heap allows the specification of, for example, a method to refer to the state before the method body has been executed.

Execution In the approach described by [SJP09b], the actual symbolic execution now reduces programs to four basic operations on the symbolic state:

1. *Evaluation* of expressions

   Finding the term that represents the value of the evaluated expression. For example, a variable access expression evaluates to the term associated with the corresponding local variable in the symbolic store. Evaluation is the only operation that does not modify the symbolic state.

2. *Execution* of statements

   For every kind of statement, the exact action is defined in a rule. For example, an assignment to a field updates the corresponding field chunk with a new term. For this to be a valid operation, the permissions to access the field must be available.

3. *Consumption* of assertions

   Consumption has slightly different meanings for access predicates and first-order logic formulae, which are both potentially contained in an assertion. Consuming access predicates results in the removal of the permissions to the corresponding chunks. If the permissions to the chunk are not available, this indicates an error. The consumption of a first-order logic formula results in a call to the theorem prover to check if it holds, given the knowledge of the path condition.

   As an example, a method's execution completes with the consumption of its postcondition. If this consumption succeeds, the method is successfully verified.

4. *Production* of assertions

   Production is the counterpart of consumption. In case of access predicates, it adds chunks to the heap, and in case of first-order logic formulae it adds them to the path condition and thus to the knowledge of the theorem prover. For example, at the beginning of the execution of a method, its precondition is produced into an empty symbolic state.

1.4 Chalice

Chalice [LM09] is an object-oriented research language for concurrent programs. It implements the implicit dynamic frames [SJP09a] technique using a permission

model to restrict read and write access to heap locations. We briefly introduce the language syntax with an example (listing 1.1): A simple linked list implementation. The following numbered list contains explanations to the parts of the code in listing 1.1 marked by the corresponding number.

1. **Predicates**

   A predicate is a way of abstracting data to hide internal details and a means of specifying recursive data structures. Their body is an assertion, for example a conjunction of access predicates (like `acc(value)`) and regular logic formulae (like the implication on line 7). They can also recursively contain predicates: `next.valid` is short for `acc(next.valid)` and denotes access permissions to `next`'s predicate `valid`.

2. **Methods**

   Methods are defined by their signature, contracts and body. The keywords `requires` and `ensures` indicate the start of a precondition and a postcondition, respectively. Here, method `add` requires `valid` and ensures `valid`. This means that the method is granted all the permissions contained in the predicate `valid` and it returns them again. Additionally, it ensures that the (new) size of the list is equal to the old size increased by 1. The `old` keyword causes the expression in parenthesis to be evaluated in the *old* state instead of the current one.

3. **Unfold**

   When granted access to a predicate, the permissions contained within it are not directly available until the predicate is unfolded using the `unfold` statement.

4. **Fold**

   The `fold` statement is the counterpart of `unfold`. It collects all permissions contained in the corresponding predicate and replaces them by a predicate permission.

5. **Functions**

   Functions in chalice are always side-effect free. Their body consists of a single expression that is evaluated to obtain the return value. Like methods, they can have pre- and postconditions. As a difference from methods, functions always return all permissions they have obtained from the caller, without explicitly specifying them in the postcondition.

6. **Unfolding**

   The `unfolding` expression unfolds a predicate permission. In contrast to the `unfold` statement, it does this only to evaluate another expression (after the

```
class LinkedList {
    var value: int
    var next: LinkedList

    predicate valid {   // (1)
        acc(value) && acc(next)
        && (next != null ==> next.valid)
    }

    method add(v: int)  // (2)
        requires valid
        ensures valid
        ensures size() == old(size()) + 1
    {
        unfold valid    // (3)

        if (next != null) {
            call next.add(v)
        }
        else {
            next := new LinkedList
            next.value := v
            next.next := null

            fold next.valid // (4)
        }

        fold valid
    }

    function size() : int   // (5)
        requires valid
        ensures result > 0
    {
        unfolding valid in
            (next != null ? next.size() + 1 : 1)    // (6)
    }
}
```

Listing 1.1: Introductory example of the Chalice language: LinkedList objects hold an integer value and a reference to the next node, which is again a LinkedList object. The `next` reference of the last node of the linked list is `null`.

keyword `in`) which depends on the permissions contained in the predicate. The latter expression defines the value of the `unfolding` expression.

Concurrency features Chalice supports multi-threaded programming through a fork-join mechanism, monitor locking and channels for inter-thread communication. Chalice programs can be proved to be free of race conditions and dead-locks. In our work, the concurrency features of Chalice are not covered specifically.

Permissions model We have already seen access predicates like `acc(value)` in our example. The meaning of it is "100% access permissions for `value`". Chalice uses a permissions model with read and write permissions. A thread must have 100% permissions on a heap location in order to perform write operations on it, but only a fraction of it to read it. Refer to [HLMS11] for a detailed explanation of the permissions model.

Modular verification The verification of Chalice programs is done modularly. This means that there is a set of *verifiable elements* in a program which can be verified independently. These are: Methods, functions, predicates, monitor invariants and channels (refer to [LMS10]), whereas only methods and functions need to be verified with respect to their specification. The verification of predicates, monitor invariants and channels only consists of a well-formedness check [Sch11].

In this report, we usually use methods as examples. But most concepts apply as well to the execution of the other verifiable elements.

1.5 Syxc

Syxc[Sch11] is an automatic verifier for the Chalice language that uses the approach of symbolic execution described by [SJP09b]. Syxc uses the theorem prover Z3 [dMB08]. Our debugging tool uses Syxc as its backend verifier. Our work therefore heavily depends on the internals of Syxc. For that reason we will discuss it in depth in the next chapter.

1.6 Debugging

Debugging in our sense is the discipline of searching for the cause of a failure in a program. Using tool support for debugging is well-established in the world of software engineering when it comes to concrete executions. Interactive debuggers can support a systematic approach to debugging that bases on observation, hypotheses and predictions [Zel07]. The same is valid for symbolic execution and program verification.

If a proof fails, this can have various reasons. Generally, these are either wrong implementation, wrong specification or underspecification. Without tool support,

it is left completely to the programmer to find the problem using the data reported by the verifier.

In case of a failure, Syxc reports the location and type of the failure. Syxc is also able to output the description of every elementary step taken and all intermediate symbolic states, both in a flat textual representation. Since the number of steps and the size of intermediate states can be quite high even with trivial programs, it is hard for the programmer to find the information that is relevant to find the cause of a problem. A debugging tool can help structuring this information and pointing the user to the relevant data.

1.7 Goal

The goal of this master's thesis is to design a debugger for Syxc that showcases and explores possibilities of how to help programmers in finding the cause of verification failures. This includes finding a suitable (graphical) representation of symbolic state, identifying common debugging situations and helpful interactions between developer, debugger and verifier, and finally implementing a corresponding GUI prototype.

2 Symbolic execution with Syxc

To give the reader an intuition of how Syxc works, we extend our introductory Chalice program in listing 1.1 with a new method `setValue` that is simple enough to demonstrate its symbolic execution: `setValue` performs a task as simple as (re-)setting the value of the current list node. Listing 2.1 shows its code, together with the already known predicate as a reminder.

```
class LinkedList {

    ...

    method setValue(v: int)
        requires valid
        ensures valid
    {
        unfold valid
        value := v
        fold valid
    }

    predicate valid {
        acc(value) && acc(next)
        && (next != null ==> next.valid)
    }

    ...
}
```

Listing 2.1: Extended introductory LinkedList example

The following sections use the example to highlight different aspects of the symbolic execution and introduce concepts that are important for the understanding of the debugger we are going to design later on.

2.1 State transitions

Let us now go through the symbolic execution of `setValue`. We will use a syntax for heap chunks introduced in [Sch11]. Figure 2.1 contains the body of `setValue`

annotated with a (simplified) symbolic state after each statement.

1. We start with an empty heap and a store with the local variables `this` and v pointing to two freshly created terms. Let's call them t and t_v.

2. The precondition is produced. This results in a new predicate chunk, let's call it $t.valid[t_{valid}]$.

3. The predicate `valid` is unfolded. This includes the consumption of the predicate chunk and the production of the predicate's body. Now, the implication in the predicate's body is causing the execution to split up. This is necessary because the implication is not pure, that is, it contains an access predicate in the consequent, namely another predicate `next.valid` and Syxc's heap chunks do not support such conditional permissions. Syxc continues with two different symbolic states, one with the produced predicate chunk $t.next.valid[t_{nextvalid}]$ (and the assumption $t_{next} \neq null$) and one without it (with the assumption $t_{next} = null$). This results in two different execution paths. Figure 2.1 shows both of them in sequence, but up to the `unfold` statement they are equal.

4. The assignment of v to the field `value` causes the field chunk $t.value \rightarrow t_{value}$ to be updated to $t.value \rightarrow t_v$, since $t_v$is the value of v. Note that this is the same for both paths, but still there are different heaps for the two paths.

5. The predicate `valid` is now folded again. First, the predicate body is consumed. Note that here, although we have the same impure implication again, the paths are not split up any more. This is because the path condition π contains an assumption in both paths that limits the evaluation of the implication to one case each. In path one, where the antecedent was assumed to be true, it is not possible to now assume it to be false, since this would be contradictory. In path 2, the same applies the other way round.

   Once the predicate body is consumed, the predicate chunks $t.valid[(t_v, t_{next}, t_{nextvalid})]$ in path 1 and $t.valid[(t_v, t_{next})]$ in path 2 are added to the heap. Note that these two chunks are not the same, because they are abstractions of different data. The terms in brackets indicate what data the chunks abstract from. In the first case it is the new value of v, t_v, the value of `next`, t_{next}, and $t_{nextvalid}$, which is describing the unknown abstracted data of `next.valid`. In the second case, `next` is $null$, there is no predicate `next.valid` and therefore the predicate valid does not include anything more. These terms in brackets are called *snapshots*. Every predicate chunk has a snapshot. If nothing is known about the abstracted data, a fresh term is created as a snapshot, like it was done in (2) when producing the chunk $t.valid[t_{valid}]$.

6. The postcondition is consumed. In both paths the heap contains a corresponding predicate chunk $t.valid[...]$ (the snapshot does not matter here), thus both paths are successfully verified and so is the whole method.

```
Path 1 {
        h : ∅  // (1)
        γ : this → t, v → t_v
    (produce precondition)      // (2)
        h : t.valid[t_valid]
    unfold valid        // (3)  ⟵ execution path splits
        h : t.value → t_value, t.next → t_next, t.next.valid[t_nextvalid]
        π : t_next ≠ null
    value := v      // (4a)
        h : t.value → t_v, t.next → t_next, t.next.valid[t_nextvalid]
    fold valid      // (5a)
        h : t.valid[(t_v, t_next, t_nextvalid)]
    (consume postcondition)     // (6a)
        h : ∅
}

Path 2 {
        h : ∅ // (1)
        γ : this → t, v → t_v
    (produce precondition)  // (2)
        h : t.valid[t_valid]
    unfold valid    // (3)
        h : t.value → t_value, t.next → t_next
        π : t_next = null
    value := v  // (4b)
        h : t.value → t_v, t.next → t_next
    fold valid  // (5b)
        h : t.valid[(t_v, t_next)]
    (consume postcondition) // (6b)
        h : ∅
}
```

Figure 2.1: Symbolic execution of method `setValue`

2.2 Continuations

Let us take a closer look at the splitting of the execution and how it is achieved. At the beginning, there is no indication of the necessity to later split the execution,

since the method's body does not contain any conditionals. However, the symbolic execution algorithm must always be prepared to react to conditionals. Splitting the execution results in the remainder of the method being executed twice, but with different initial states. A way to do this is the implementation of the *continuation-passing style* [FW08]. At any point in the execution, the remainder of the method can be executed by calling the *continuation* function. In turn, the execution of every statement (and also that of expressions and assertions) takes a continuation function as an argument, telling it what to do next.

Returning to our example, figure 2.2 illustrates the execution of our example method in continuation-passing style. The $Q_i(\sigma_i)$ = { ... } denote continuation functions that are passed to the execution of the statements as arguments. Each continuation takes an argument σ_i, which is used to the symbolic state as the continuation. In each case, the continuations contain the whole remainder of the method's execution. For example, the continuation of the production of the precondition, Q_1, contains the execution of all statements that follow. Q_5, the continuation passed to the consumption of the postcondition contains nothing but the information that the execution completed with success.

As we have seen before, the `unfold` statement causes the execution to split up. Now, the symbolic execution algorithm can just call the continuation Q_2 twice, once passing the symbolic state resulting from the *true* case and once passing the symbolic state resulting from the *false* case as its argument.

```
"produce precondition" (Q1(σ1) = {
    "unfold valid" (Q2(σ2) = {
        "value := v" (Q3(σ3) = {
            "fold valid" (Q4(σ4) = {
                "consume postcondition" (Q5(σ5) = { success })
            })
        })
    })
})
```

Figure 2.2: Continuations in the symbolic execution of method `setValue`

2.3 Execution trace

We have seen so far how symbolic state and state transitions work and how the execution can split up. This is a very high level view on the execution of our method. We want to go one step further and demonstrate the execution of one single path in more detail and from a slightly different perspective. This time, we

describe every single elementary execution step[1], namely executions, evaluations, productions and consumptions. Let us just have a look at the `unfold` statement:

1. It starts with the *execute* step of the `unfold` statement.
2. The receiver object of the predicate to be unfolded is evaluated, which is `this` in this (unqualified) case.
3. The predicate chunk is consumed.
4. The predicate body is produced. It is a conjunction. Since the operator && is binary, the three conjuncts are actually represented by two nested conjunctions.
5. The outer conjunction is produced by producing the left and then the right conjunct.
6. The left conjunct is again a conjunction, which is produced the same way.
7. The right conjunct is an implication. Its production...
8. ... consists of the evaluation of the antecedent and, ...
9. ... if the antecedent is assumed to be true (as it is the case in our path 1), the production of the consequent.

It becomes clear that these steps build a nested structure, where every step potentially consists of other steps. Listing 2.3 shows the execution of our example method, demonstrating this nested structure, where indentation represents an increase of the level of nesting. Note how the original code structure largely corresponds to the nesting structure of the steps. We call this nested structure the *(single-path) execution trace* of the first path of our example method. Like this, there is an execution trace for every path of every method, as well as all other verifiable elements.

Unfortunately, this representation only includes one single path and thus misses a lot of information about the execution of the method. Wouldn't it be possible to represent both paths in one trace? Figure 2.4 tries to do that. We heavily simplify the steps for this purpose (for example, we omit most of the evaluation steps). The first 8 lines are similar to the representation in figure 2.3. This is the part of the execution that is common to both paths. What follows is the part that is path-dependent. We place all steps belonging to the remainder of the paths into steps called "*true*-branch" (green box) and "*false*-branch" (red box), respectively. Both of these steps contain another step called "continuation" (grey boxes). These steps in turn contain everything that is execution as part of the continuation of the production step that caused the execution to split.

[1] In fact, quite a few steps of the sort "evaluate receiver this" have been skipped for the sake of brevity.

```
- verify method "setValue"
  |- produce precondition "valid"
  |  |- produce pred. access "valid"
  |- execute "unfold valid"
  |  |- evaluate receiver "this"
  |  |- consume pred. chunk "valid"
  |  |- produce pred. body "acc(value) && acc(next)
  |     |            && (next != null ==> next.valid)"
  |     |- produce "acc(value) && acc(next)"
  |     |  |- produce "acc(value)"
  |     |  |- produce "acc(next)"
  |     |- produce "(next != null ==> next.valid)"
  |        |- evaluate antecedent "next != null"
  |        |  |- evaluate receiver "this.next"
  |        |  |- evaluate receiver "null"
  |        |- produce consequent "next.valid"
  |- execute "value := v"
  |  |- evaluate receiver "this"
  |  |- evaluate value "this.v"
  |     |- evaluate receiver "this"
  |- execute "fold valid"
  |  |- evaluate receiver "this"
  |  |- consume pred. body "acc(value) && acc(next)
  |     |            && (next != null ==> next.valid)"
  |     |- consume "acc(value) && acc(next)"
  |     |  |- consume "acc(value)"
  |     |  |- consume "acc(next)"
  |     |- consume "(next != null ==> next.valid)"
  |        |- evaluate antecedent "next != null"
  |        |- consume consequent "next.valid"
  |           |- evaluate receiver "this.next"
  |- consume postcondition "valid"
     |- consume pred. access "valid"
        |- evaluate receiver "this"
```

Figure 2.3: Step-by-step execution of method `setValue`, path 1

This trace now contains the whole execution of our example method. The diverging part of the two execution paths are completely separated from each other. If we traverse it from top to bottom we obtain exactly the sequence of steps that have been executed by the symbolic execution algorithm. There is, however, a problem with this representation: The nesting structure of the steps inside the continuation steps does not reflect the code structure any more. For example, the consumption of the postcondition appears in the innermost level of nesting in the trace, but it should be on the same level as the production of the precondition according to the code structure. There is no way to reconstruct the correct nesting structure from this representation. As a consequence, there is also no way to compute the two single-path execution traces out of this trace.

```
- verify method "setValue"
  |- produce precondition "valid"
  |- execute "unfold valid"
     |- consume pred. chunk "valid"
     |- produce pred. body "acc(value) && acc(next)
        |            && (next != null ==> next.valid)"
        |- produce "acc(value) && acc(next)"
        |- produce "(next != null ==> next.valid)"
           |- evaluate antecedent "next != null"
           |- true-branch
           |  |- produce consequent "next.valid"
           |  |- continuation
           |     |- execute "value := v"
           |     |- execute "fold valid"
           |     |- consume postcondition "valid"
           |- false-branch
              |- continuation
                 |- execute "value := v"
                 |- execute "fold valid"
                 |- consume postcondition "valid"
```

Figure 2.4: Multi-path execution trace of method `setValue` (attempt)

In yet another attempt to create a execution trace containing multiple paths in figure 2.5, we overcome the shortcomings of the first attempt by placing every step at the correct level of nesting instead of nesting them in special "continuation"-steps of the branches. This representation has the disadvantage that the steps of the two execution paths overlap. This, however, is a rather cosmetic issue, since now we are able to construct the single-path traces out of this trace by filtering out all steps that belong to the other branch, respectively.

We call this representation the *multi-path execution step* (as opposed to the single-path traces) of the method `setValue`. Note, that we can assign a branch to every step in this trace. We have already done so in figure 2.5 for the steps that belong to the two branches that resulted from the split of the execution. Additionally, we can think of a *root branch* that contains the execution before the split. Every step in the trace, that does not already belong to a branch, can be assigned this root branch.

```
- verify method "setValue"
  |- produce precondition "valid"
  |- execute "unfold valid"
  |  |- consume pred. chunk "valid"
  |  |- produce pred. body "acc(value) && acc(next)
  |     |            && (next != null ==> next.valid)"
  |     |- produce "acc(value) && acc(next)"
  |     |- produce "(next != null ==> next.valid)"
  |        |- evaluate antecedent "next != null"
  |        |- true-branch
  |        |  |- produce consequent "next.valid"
  |        |- false-branch
  |           |- -
  |- execute "value := v"
  |- execute "fold valid"
  |- consume postcondition "valid"
  |- execute "value := v"
  |- execute "fold valid"
  |- consume postcondition "valid"
```

Figure 2.5: Multi-path execution trace of method `setValue`

2.4 Execution paths

As we have seen already, impure implications are handled by the symbolic execution algorithm by splitting up the execution into two branches. Now, we want to give a more comprehensive explanation for execution splitting.

2.4.1 Global branches

The main principle is the same for other conditionals as it is for impure implications: The execution is split up into two branches, one of them assumes the condition to be *true*, the other one assumes it to be *false*. Both branches execute

their respective branch-specific code (e.g. the *then* and *else* blocks of *if-then-else* statements).

Now, whenever the branch-specific code modifies the heap, the execution of the two branches can not be joined after the execution of their respective code block without losing the branch-specific information. This is the case for impure implications, as we have seen, as well as for impure ternary expressions[2], and it is also always assumed for *if-then-else* statements[3]. Therefore, both branches execute the remainder of the method independently. We call this type of branches *global branches*. Of course, there can be other conditionals in the remainder, which causes the two branches' executions to split up again, with the consequence that, for example, sequential *if-then-else* statements can result in an exponential number of branches.

At the end of the execution of the method, every branch constitutes a distinct execution path through the whole method. In our example method, we have already identified the two execution paths.

Every execution path can be seen as the execution of a method from beginning to end (or until the first error) under certain assumptions, i.e. the conditions that lead to the execution path. This allows us to handle each path in separation, giving the user the possibility to concentrate on those of interest (usually the ones containing an error).

2.4.2 Local branches

When talking about global branches, we restricted the types of conditionals that cause them. All other conditionals, namely pure implications and ternary expressions, do not cause the execution to split up into two branches which will both run until the end of the method. We use the example in listing 2.2 to demonstrate this:

Method `setX` sets the value of the field x to the absolute value of the argument `i`. This is done as follows: A ternary expression evaluates to $-i$ if i is negative and to i if i is positive. The result is then assigned to x.

The ternary expression could be executed like the other conditionals, namely that the execution splits up into two branches. One of them assumes $i < 0$ and evaluates the ternary expression to $-i$, sets the value of x to $-i$ and finally consumes the postcondition successfully, since $-i \geq 0$ if $i < 0$. The other branch does accordingly, thus the remainder of the method is executed a second time.

There is, however a more efficient way to handle this particular conditional (also see [KMS12]): The execution is split up only to evaluate the *true*- and the *false*-expression of the ternary expression. Here, this is a trivial task. The results of these two evaluations are combined into a conditional symbolic value, $IfThenElse(i <$

[2] A ternary (conditional) expression is of the form *c ? t : f*. Its value is *t* if *c* is true and *f* if *c* is false.

[3] even if one could easily imagine if-then-else statements without any heap modifying statement in their *then*- or *else*-blocks.

```
class PositiveCell {
    var x: int

    method setX(i: int)
        requires acc(x)
        ensures acc(x) && x >= 0
    {
        x := (i < 0 ? -i : i)
    }
}
```

Listing 2.2: Example for local branching

$0, -i, i)$, which in turn is assigned to the field x. The postcondition is then checked only once, given the conditional value of x. It is left to the theorem prover to solve the formula $IfThenElse(i < 0, -i, i) \geq 0$.

Since the execution is merged after the execution of the branch-specific code, the branches are called *local branches* (as opposed to global branches) . This also means that they do not lead to separate execution paths. We refer to the splitting of an execution into local branches as *local splitting*.

2.4.3 Unreachable branches

In the discussion of our example in section 2.1 we pointed out that the unfold statement does not cause the execution to split up, although it contains an impure implication. We argued that one case contradicts the current path condition and thus only the other case is taken. Generally spoken, a condition can be always satisfied or unsatisfiable taking the path condition into account. Technically, in those cases, the execution *is* split up indeed as normal, but the contradictory branch's execution is cancelled immediately. Those branches are called *unreachable branches*.

2.4.4 Branches within branches

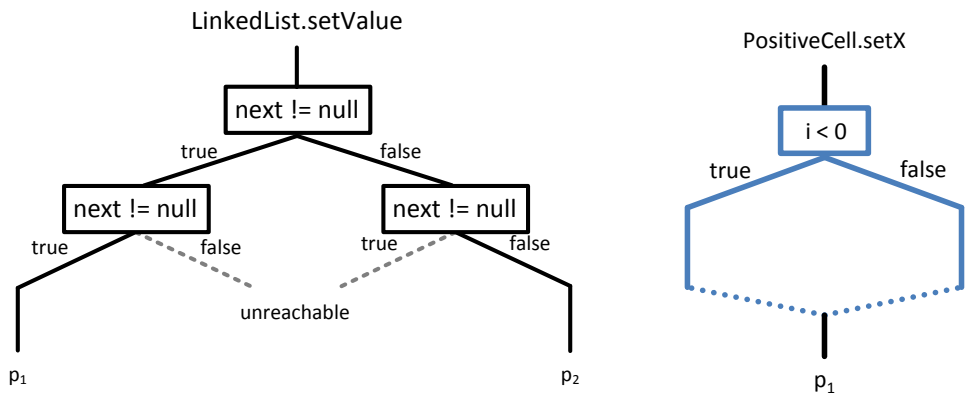
We have already seen how global branches can split up repeatedly, thus creating an exponential number of them. It is also possible for local branches to contain other local branches. This situation can be found if a branch of a pure conditional contains another pure conditional, as, for example, in the boolean expression `(c ? true : a ==> b)`, where a, b and c are boolean values.

The most interesting case is the nesting of global branches within local branches. Coming back to our original LinkedList example in listing 1.1, we can identify such a case: In the function `size`, the conditional expression `(next != null ? next.size() + 1 : 1)` is evaluated. Since this is a pure conditional, the

execution is only split locally. The *true*-branch contains a recursive call to `size`, however with the receiver `next`. The evaluation of the function's body contains an unfolding of the predicate `valid`. The discussion of the method `setValue` already revealed the global splitting that is involved with unfolding this predicate due to the impure implication it contains. What is the effect of the execution being split up globally at that point? Here, the term *global* must be interpreted in the context of the surrounding local branch only. This means that these nested global branches do not contain the execution of the whole remainder of the function to be verified, but only the execution of the local branch they are contained in, up to the point where the two local branches are joined again.

2.5 Execution trees

All the execution paths of a method together form an *execution tree*. It describes the structure of the execution of a method. Figure 2.6a shows the complete execution tree of our example method `LinkedList.setValue`.



(a) Complete execution tree of example in listing 2.1

(b) Complete execution tree of example in listing 2.2

Figure 2.6: Execution tree examples with p_i being the execution paths.

The symbolic execution algorithm builds the execution tree in a depth-first manner, verifying one execution path at a time. If an error occurs during the execution of one of the paths, the algorithm stops the execution of the corresponding method and the remaining paths are left unexplored. This means that after one run of the algorithm there may be some paths that could be verified, one unverified, i.e. erroneous path and some paths that have not been explored and hence are unknown. The paths can be uniquely identified by the leaves of the execution tree.

2.5.1 Local branches in trees

Because pairs of local branches are joined again during the execution, they always belong to the same execution path. The execution tree in its original form only shows execution paths. Therefore, local branches do not show up in the execution tree. However, the tree can be extended with local branches that at some point join into the same path again. Figure 2.6b gives an example. The blue box represents the condition leading to the local branches, the blue edges represent the local branches and the blue dotted lines symbolize the joining of the local branches. Note that this tree consist of only one path.

2.6 Symbolic state

In Syxc, a symbolic state $\sigma = (\gamma, h, g, \pi)$ is composed of a symbolic store γ of local variables, two symbolic heaps h and g (current and old heap) and the path condition π.

Symbolic values can be anything from literals (e.g. integer or boolean) to symbol terms. Symbol terms do not necessarily have a concrete correspondence in the code, as they are often created by the verifier as auxiliary terms. They are identified by a generated name. Also more complex terms like binary operations, sequences or quantifications can be symbolic values.

The *symbolic store* γ lists all defined local variables, that is, it maps variables to their symbolic value. For example, the store $(this \rightarrow t,\ x \rightarrow t_x)$ contains two local variables: $this$ maps to the symbol t and x maps to t_x. Here, $this$ is known to be of a reference type, thus t is of the same type. The type of x and t_x is not clear from this representation. Internally, every term is of some type. However, not Chalice's type system is used, but a much smaller one understood by the theorem prover, containing only types like *int*eger, *boolean* or *ref*erence.

The *path condition* π is a set of boolean terms. It contains the knowledge of the theorem prover about the current execution path. For example, when the execution is split up due to a conditional, the condition is assumed to be true in one branch and it is assumed to be false in the other branch. This means that the path condition of the first branch contains the condition in its original form and the path condition of the second branch contains its negation.

A *symbolic heap* can be seen as a view on the memory heap where only a very restricted part of the information is available. In fact, symbolic heaps are partial heaps whose shape is determined by the permissions they hold. For example, $(t.x \rightarrow t_1\#(100,0),\ t.valid[t_s]\#(50,0))$ is a heap with two heap chunks. $t.x \rightarrow t_1\#(100,0)$ is called a *field chunk*. It represents the heap location of a field x of an object t and maps it to a symbolic value t_1. The chunk also includes information about the permissions currently available to access the heap location $t.x$. Here, $(100,0)$ stands for 100% write access and 0 epsilon permissions. The existence of a field chunk in a heap implies at least read permissions to the corresponding

heap location.

$t.valid[t_s]\#(50,0)$ is a *predicate chunk*, the second type of heap chunks. t_s is the snapshot term, defining the data, the predicate chunk abstracts from. If nothing is known about the snapshot, for example if it is an arbitrary symbol term and the path condition does not specify any constraints on its symbolic value, this means that there is no information available about the abstracted data. On the other hand, *combine* terms are used to describe snapshots that contain such information[4]. For example, if we have a predicate `p {acc(x) && acc(y)}` and a predicate chunk $a.p[combine(t_x, t_y)]$, the snapshot contains the information that the field $a.x$ has the value $t_x$and the field $a.y$ has the value t_y. If at some point in the execution, the predicate chunk $a.p[combine(t_x, t_y)]$ is unfolded, the resulting field chunks for x and y will use the values $t_x$and t_y. We can also imagine the notion of *nested chunks* of such a predicate chunk. Nested chunks, as opposed to *direct chunks*, are not directly available because they are abstracted by a predicate chunk. Apart from that, they are equal to direct chunks. Whether a predicate chunk contains nested chunks, depends on whether the knowledge about them is available within the snapshot.

As stated above, a symbolic state consists of two symbolic heaps, namely a current and an old heap. The *current heap* is the one used for all heap operations, such as adding or removing chunks, or asserting a permission predicate. The *old heap* is a backup of the state of, for example, the execution of a method directly after the precondition was produced. In the postcondition, the method can refer to this state using the `old` keyword. For example, `old(this.x)` refers to the old value of a field `this.x`. The evaluation of this expression constructs a new symbolic state that contains the same store and path condition as the current state, but uses the latter's old heap as its current heap. The inner expression, that is `this.x`, is then evaluated in the context of this constructed state.

[4]Combine terms are only used if a predicate abstracts from more than one heap chunk. Otherwise, a single snapshot term is enough.

3 Conceptual design of a symbolic execution debugger

3.1 Introduction

The symbolic execution approach used in Syxc allows the debugging to be somewhat similar to conventional (i.e. concrete execution) debugging in the aspect of execution control. E.g., the concepts of stepping in, out and over can be transferred directly to symbolic execution debugging.

There are a few outstanding points of symbolic execution and program verification when it comes to debugging:

- Symbolic execution is done modularly, i.e. every method is executed and verified independently. This has the advantage that we can focus on one method at any time and thus have a much smaller debugging scope than with conventional debugging. Also, there is no restriction for entry points to be considered (such as *main* methods).

- Symbolic execution abstracts the system environment and all application domain parameters and thus is independent from them.

- Intermediate states are easily recordable and reversible. This is because, unlike the concrete execution of a compiled program, the symbolic execution does not have to deal with low level concerns like CPU state.

- However, symbolic execution is more time-consuming than the concrete execution of (compiled) code (apart from loops which are executed only once during verification).

Taking these points into account, the goal is to give the user an experience that both resembles conventional debugging and takes advantage of additional possibilities. In this chapter, we develop a concept for a debugging tool for symbolic execution. In the design of the debugger we follow a few generic principles which are listed in figure 3.1.

1. **Adjust the level of detail**

   The user should always have the choice to select the level of detail of the information that is displayed. The amount of data is too large to be comprehended otherwise.

2. **Use the user's language**

   Whenever possible a language that the user feels at ease with or even the user's own words should be used (instead of internal representations). This reduces the need for the user to translate and thus allows a better interaction. Here, what we refer to is the programming language, that is Chalice.

3. **Reuse known concepts**

   Do not reinvent the wheel. Using wholly new concepts forces the user to start a new learning curve and still does not guarantee the new concepts to be superior to existing, well understood ones.

4. **Compare *before* and *after***

   Showing the effect of every action by contrasting pre- and post-state is a very simple yet effective way to help the user understand any kind of transition.

Figure 3.1: Design principles

3.2 Feature overview

In the following subsections we present a short overview of the main features of our debugger concept. Later, when presenting its user interface in section 3.4, these features are discussed in more detail and more specific features are introduced in addition.

3.2.1 Decomposition

Programs and their execution can be decomposed at many levels. Each program consists of channels and classes, the latter of which in turn consist of class member. As we have seen in the previous chapter, those elements that are verifiable have an execution tree in which every path from the root to a leaf node represents an execution path. Execution paths, in turn, represent a (single-path) execution trace in the form of a tree of steps. Every step in that step tree holds a pre- and a post-state. Finally, the symbols, terms and formulas used to describe a state are the elementary building blocks of our hierarchy. Figure 3.2 shows the

decomposition of the complete hierarchy from the upper left corner in a clockwise direction.

As a fundamental feature, our debugger provides means to navigate through this hierarchy in all directions:

- across the levels: adjusting the level of detail of the available information, for example, selecting a method, then selecting an path of its tree and selecting a step out of the path's trace.

- along the levels: shifting the view to another partition of the available information, for example changing from one method to another, or stepping through an execution trace.

This hierarchical view on a program facilitates a deep understanding of a program and it allows the user to find important information quickly, both of which is essential when it comes to finding the reason for errors. This is also in accordance to design principle 1.

Figure 3.2 also indicates elements at all levels that are part of or consist of an erroneous element (coloured red). Down to a certain level, an error can be located automatically by the verifier. E.g. it is always known in which element, in which execution path and in which step an error occurred. Depending on the kind of error, the verifier can also give information about which part of the state is causing the error, e.g. missing permissions to access a particular heap chunk. In any case, information about errors should be available in an appropriate form at all possible hierarchical levels. For example, it is very common to mark the location of an error in the code at an appropriate position. But also, the information about which path, step and heap object is causing the error should be available to the user immediately as soon as the they navigate to the corresponding level of the decomposition hierarchy.

Debugging work flow The discussion about decomposition leads us to a proposed general debugging work flow: The user starts a debugging session with a program containing errors. After choosing one of the errors the user approaches its point of origin through our decomposition hierarchy (navigation across levels). On each level of detail, it is probably necessary to inspect nearby elements (navigation along levels) to gain more insights about the error's cause. Ideally, when arriving at the highest level of detail, the user should have developed enough understanding to construct a hypothesis of why the error occurs. If so, the user can think about a solution for the error and return to the program level of our hierarchy to implement a fix. Figure 3.2 illustrates this cyclic work flow using red and green arrows, symbolizing the concern for the error and the solution, respectively.

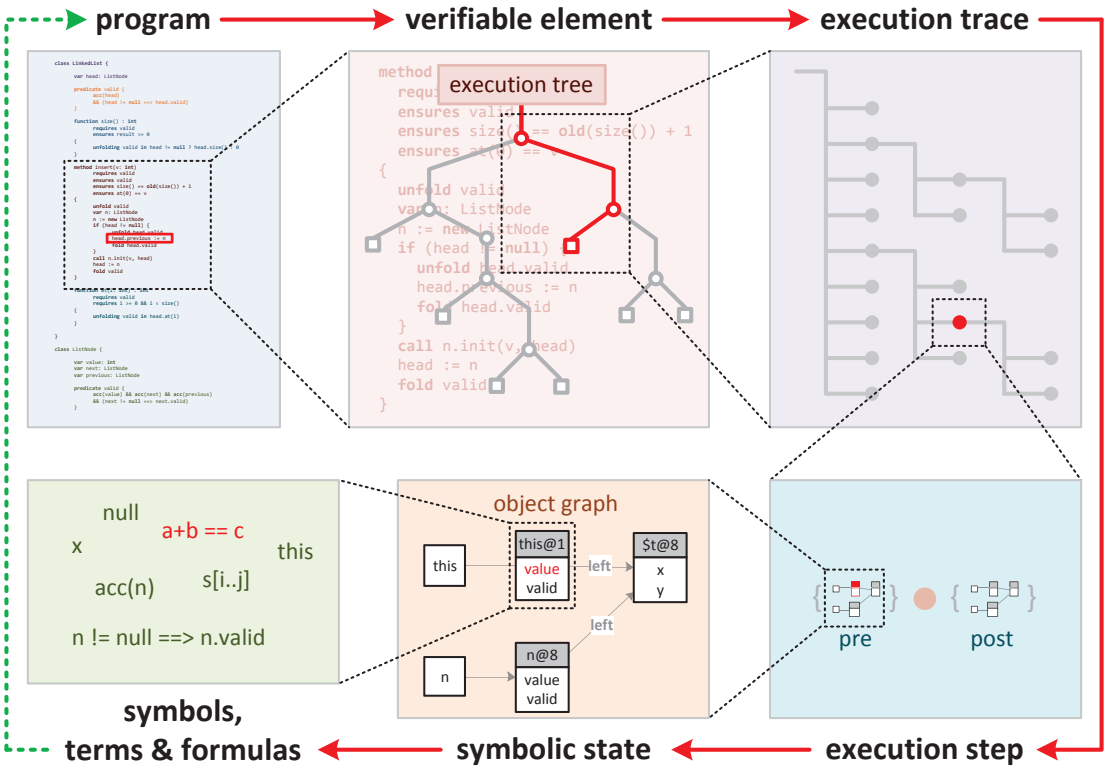


Figure 3.2: Program decomposition

3.2.2 Stepping

The developer wants to step through the program while being presented with the state at each program point of interest so that they can understand the effect of every code fragment. This gives the user the possibility to view all the available state data in a structured way by connecting it to what the user knows best: the source code.

Stepping is a well-known feature of conventional debuggers and as such a great entry point into the world of symbolic execution debugging.

3.2.3 State inspection

State inspection, i.e. the possibility for the user to see intermediate internal execution states in a human readable representation, is a feature also very commonly used in conventional debugging, e.g. through a "Variables" view, watch expressions or tooltips that display the value of a variable in the code editor when placing the mouse cursor over it. All of these ideas can be directly transferred to symbolic execution debugging. However, this report focuses on the representation of symbolic state (and in particular, the symbolic heap).

The importance of state inspection in combination with stepping for debugging

purposes becomes clear when we think of a block of code as a function of input values computing output, or, in other words, the transition of an initial state to a final state. To understand this transition, the code is decomposed into its constituents which may be further decomposable. Every building block can then again be seen as such a state transition, between each pair of which there will be an intermediate state that represents an intermediate result of the computation.

3.2.4 State manipulation

When inspecting intermediate states a problem can often be narrowed down to a certain missing or wrong value or path condition. In such situations the most intuitive action to take is to change or complement the according intermediate state so that it meets the requirements to solve the problem. This of course does not fix the broken code. Nevertheless, if the code can be verified taking these changes into account, we can shift attention to the question of how to achieve the same effect changing the code. This may be a much easier task to fulfil than to solve the initial problem directly.

3.3 Verification process

Before we discuss the frontend of our debugger concept, let us sketch the use of the verifier in the backend.

At the beginning of a debugging session, after a source file has been chosen by the user, the file is parsed and the abstract syntax tree is built. At this point the verification status of every verifiable element is unknown. Now, the program should be verified, that is symbolically executed with all proof obligations discharged. Since symbolic execution is done modularly, the user usually selects one or more verifiable elements to be verified at once.

One could imagine the situation where the user is only interested in the effect of a specific line of code. In conventional debugging, the user would set a breakpoint on that line of code and start the execution. The program would execute until it reaches the breakpoint and then pause, letting the user inspect the intermediate program state and later decide to resume the execution.

This approach can be directly transferred to symbolic execution. However, we identify the following weak point:

The responsibility to initiate the execution is assigned to the user. For example, after pausing the execution at some point, there is a (presumably high) probability that the user will want to inspect the remainder of the method. After the user finally chooses to resume the execution, they are kept waiting again until the method has been executed to the end. Instead, while the user is inspecting the intermediate state, which usually is not a very resource-demanding task, there would be time and capacity to execute the remainder of the method. Considering

that the execution of non-trivial methods is very time-consuming, the waiting time induced for this reason must not be neglected.

To overcome these inefficiency, we suggest a more adaptive approach to managing the verification of a program:

1. The verification is done in a background task, allowing the user to still interact with the user interface while verifying.

2. The order of verification of all verifiable elements is adapted dynamically to the user's demand. For example, if the user chooses to debug a certain method which has not yet been processed, the background task pauses the execution of the currently scheduled element and immediately starts verifying the method requested by the user. Later, the interrupted execution is resumed from where it was left.

3. The user can *simulate* the execution of a method which has been verified earlier, while some other method is being verified. Simulating an execution is equivalent to navigating within the execution's recorded trace. Since it requires no complex computation, simulation can be done in virtually no time. Except, in the case of a method that has not already been executed completely, its simulation obviously needs to be synchronized with the execution.

4. Execution tree and trace are built up gradually in synchronization with the background execution. This means that even if the execution of a method has not yet been completed, parts of the execution tree and trace can already be shown to the user.

5. Simulation and execution are completely decoupled. This means, for example, that the user can pause the simulation of a method at any time while execution of the method continues in the background. If the user later decides to resume the simulation the execution may already have finished and the simulation can take place in its natural speed.

There is a possible use case that still potentially causes the user to be kept waiting unnecessarily: In the case of conditionals, the user might be interested in the second branch only. As the verification algorithm traverses the execution tree in depth-first manner, the first branch is completely executed before the execution of the second branch is started. Thus, if the method has not previously been executed anyway, the user will have to wait until the first branch has been completed before the second branch can be inspected. This problem could be approached by providing the user with a way to control which branches are executed first.

One could imagine a more sophisticated mechanisms to manage the use of the verifier that automatically re-verifies verifiable elements whose verification is affected by modifications of the code (and only the affected ones). For example, this could be done as soon as the user applies the modifications or (alternatively) saves

them. This mechanism resembles the automatic, incremental build mechanism of modern development environments rather than an execution of the code.

3.4 User interface sketches

Following the principles in figure 3.1, the previously introduced concepts are now to be presented to the user. Below, the main ideas behind the user interface are discussed, followed by a list of detailed descriptions of the UI's core elements.

One of the main ideas is to give the user the possibility to decide on the level of detail to be shown, according to principle 1. This is done through a number of GUI elements that show different abstractions of the code and its execution at all levels of the hierarchy presented in figure 3.2:

The *execution tree view* gives an overview of the execution. It is used to determine an execution path, that is to be inspected in more detail. The *execution trace view* shows the execution trace of this path. It can again be used to select a (composite) execution step which is then further analysed using the *state view*.

Another important principle is to stay as close as possible to the user's language, that is the programming language (principle 2). Therefore, the goal is to use the Chalice's syntax instead of internal intermediate representations whenever possible. This applies to almost all of the GUI elements. The code written by the user, thus the user's own use of the language, plays a key role in the user interaction, and thus the *code view* is a GUI element of major importance.

Principle 3 suggests doing things the way the user is used to. Besides following common user interface conventions, this can be accomplished by using elements of the domain of conventional debugging.

3.4.1 Outline view

The goal of the outline view is to give a summary of the code structure of a program, i.e. the top-level declarations (classes, channels) and class members (fields, predicates, functions, methods and monitor invariants), and the verification status of those elements that are verifiable. Figure 3.3 shows the structure of an example file.

Apart from providing an overview of the program's structure, this view is also used to select an element to be further examined. The currently selected element is highlighted with a yellow background as in figure 3.3. Selecting an element causes its execution trace and tree to be displayed in the corresponding views and the code view to scroll to the element's location in the code.

The outline view described in section 3.4.1 is a good element to visually integrate the current state of the verification process. Figure 3.3, an example of the outline view, also shows two buttons to pause the process and to restart the verification. Selecting a declaration as described in section 3.4.1 also causes the background process to prioritize its verification.

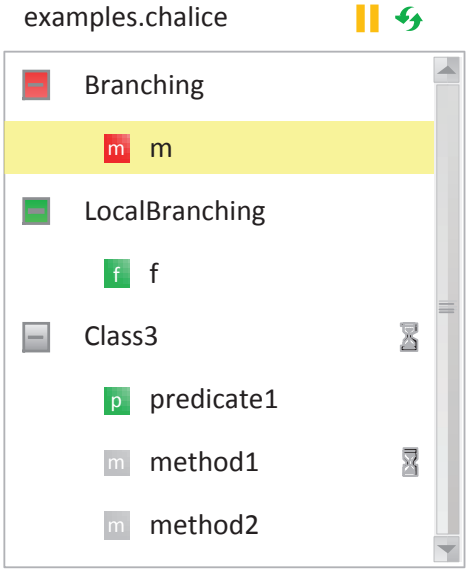


Figure 3.3: Outline view. Square symbols denote the type of a declaration (m=method, f=function, p=predicate) and its verification status (green=verified, red=error, grey=unknown/not yet verified). The hourglass symbols designate the elements currently being verified.

3.4.2 Simulation control

The following paragraphs show different possibilities offered to the user to control the simulation. The simulation is only a traversal of a recorded execution trace for which we provide means to quickly navigate through in a targeted manner. The actual symbolic execution algorithm is run beforehand by the verification process (discussed in section 3.3).

3.4.2.1 Starting and resuming the execution

Play buttons (▷) are displayed in the code editor near the definition of every verifiable class member. Clicking on one of them starts the execution of the corresponding class member. The execution is stopped either at the end of the execution trace or at the first breakpoint. Note that this is equivalent to jumping to either the last step or the breakpoint step since the execution is only simulated.

If the execution is paused (e.g. at a breakpoint) it can be resumed using a corresponding button in the tool bar.

3.4.2.2 Breakpoints

Like in conventional debuggers, breakpoints can be defined in the code view which cause the execution to pause as soon as it reaches a certain line.

3.4.2.3 Execute to cursor

A special form of setting a breakpoint is to place the cursor at a certain location in the code view and use the option *execute to cursor* (offered e.g. as a toolbar button, a context menu item and/or a keyboard shortcut) which will then execute the code until it reaches the cursor position.

3.4.2.4 Stepwise execution

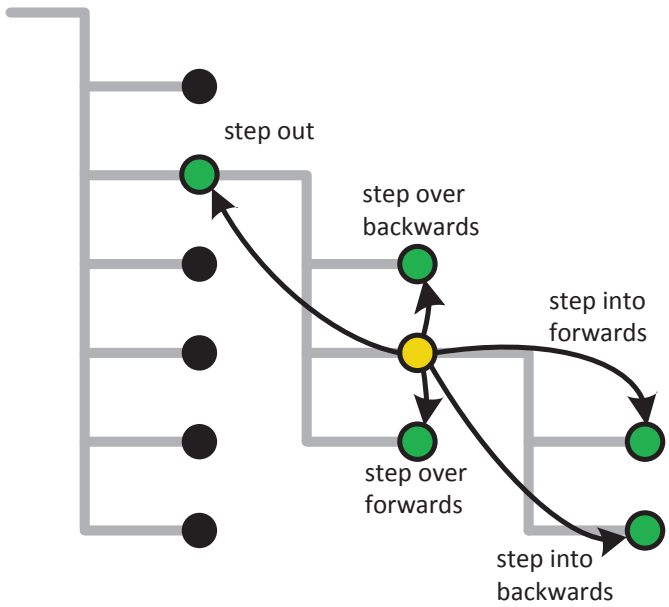


Figure 3.4: Operations for stepwise navigation within a step trace

The concepts of stepping into, out, and over are well-known to developers and applying them to verification debugging is straight-forward. Figure 3.4 shows all possible operations. *Step over* advances to the next step at the same level of nesting in the execution trace (or the next outer level if there is no next step at the current level). *Step into* is only available if the current step is composite. It advances to the first child step. *Step out* is only available if the current step is not at the outermost level of nesting. It returns to the parent of the current step.

As the full execution trace is recorded during execution and thus every step is revertible, the same operations can as well be executed in backwards direction. *Step over backwards* returns to the last step on the same level. *Step into backwards* jumps to the last sub-step of the current step. *Step out* is orthogonal to that axis, thus there is only one version of it.

Buttons for each of the described operations are provided in the user interface.

3.4.2.5 Mapping code locations to steps

Using a context menu item in the code view, the user can find the step corresponding to the execution, evaluation, consumption or production of a certain code element. As there may exist multiple steps related to a code element, for example in case of branches, the first occurring step in the currently selected execution path is chosen.

If a code element is selected by the user that is not contained in the currently selected execution path, the user will be asked to switch to an execution path containing that code element.

3.4.3 Execution tree view

The execution tree of a method is visualized using a conventional graphical representation of trees. Every condition denotes a node and branches are shown as edges. A condition splitting the execution results in two branches, one for the *true* and one for the *false* case. Accordingly, the condition's node is connected to its children through the two branches' edges. By convention, the edge on the left-hand side represents the *true* branch and the edge on the right-hand side represents the *false* branch. Single steps of the execution trace are not shown as they are on a higher level of detail.

Figure 3.5a gives an example of an execution tree view.

The execution tree view fulfils multiple tasks:

1. It helps to understand the structure of the execution trace of a method.
2. It provides an overview of the method's execution paths and their verification statuses. In case of a failure, it gives a rough hint on the failure's location.
3. It provides the interface to select the execution path to be further examined. The execution trace of this path can then be shown to the user. The tree view can even be used to select only a single code block which is of special interest, with the effect of an even shorter and thus clearer trace.

As the execution trees can grow very rapidly with increasing complexity of the program being verified, it is important to apply principle 1 even within the execution tree view. There are two ways to do this: collapsing sub-trees and zooming in on sub-trees, both described below.

3.4.3.1 Collapsing

In a deeply nested execution tree one might not be interested in every single branch. Collapsing the sub-trees whose details are not of interest helps focusing on the important parts of the tree.

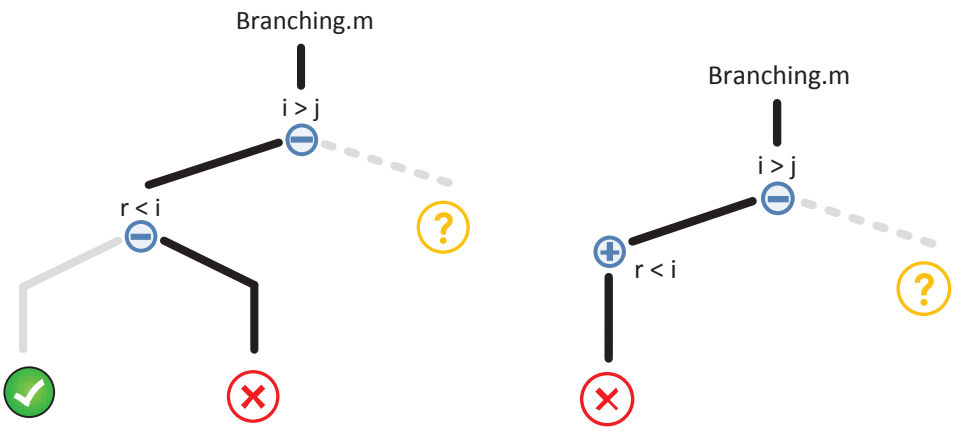
Figure 3.5a shows the whole, fully expanded tree with the erroneous path $i > j \wedge \neg(r < i)$ selected. As the tree is traversed in depth-first order and the execution

is stopped after the first error, all paths that come after the error are not explored and their status is unknown. By clicking a $\ominus$ symbol the corresponding sub-tree can be collapsed.

In figure 3.5b the sub-tree of condition $r < i$ is collapsed, resulting in a single edge representing both branches $r < i$ and $\neg(r < i)$. The sub-tree is part of an aggregated execution path $i > j$ with the verification status *error* as one of its underlying paths contain an error. Clicking the $\oplus$ symbol results in the expansion of the sub-tree.

Ideally, a collapsed tree still remembers the collapse states of its own sub-trees, so that they can be restored after the tree is expanded again.

The presumably most common application of this feature in case of a failed verification is to expand the sub-trees that contain the error and to collapse all others. Therefore, a shortcut for this operation may be worthwhile.



(a) Fully expanded execution tree with selected execution path $i > j \wedge \neg(r < i)$

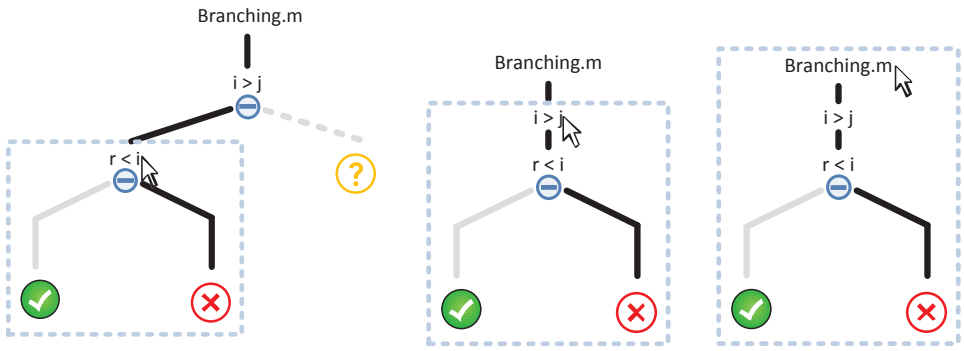
(b) Partially collapsed execution tree with selected (aggregated) execution path $i > j$

Figure 3.5: Different views of an execution tree. Symbols $\surd$, $\times$ and $?$ denote the verification status of the corresponding execution path, namely verified, erroneous and unknown, respectively. Black edges denote the currently selected execution path, whereas all other edges are grey. Dashed edges denote paths not (yet) executed, e.g. due to a prior error.

3.4.3.2 Zooming

Zooming in on a sub-tree of the execution tree facilitates concentrating on a smaller part of the tree in full detail. Figure 3.6 shows how the support for zooming could look. Both figures 3.6b and 3.6b show how to zoom out, where the former aims to zoom out one level and the latter to the top level. In this case,

the results of both actions are equivalent because there is only one level above the currently zoomed-in sub-tree. Note the dashed frame that indicates the scope of the part of the execution tree that is to be shown when clicking.



(a) Fully zoomed out. The dashed frame appears on mouse over the sub-tree condition. Clicking on it will zoom in on this sub-tree.

(b) Zoomed in on the sub-tree $r < i$. Clicking the $i > j$ condition above the sub-tree will zoom out to the next higher level, showing the sub-tree containing the root of the currently shown sub-tree as its child.

(c) Zoomed in on the sub-tree $r < i$. Clicking the method identifier label (which is the root node of the execution tree) will zoom out to the outermost level, showing the whole tree.

Figure 3.6: Demonstration of sub-tree zooming

Of course, regular zooming, for example via mouse wheel, is also a valid option. An advantage of the zooming mechanism described above is that the path to the current sub-tree is still visible in a very compact way.

3.4.3.3 Path selection

As stated above, the execution tree view is used to make the selection of an execution path for further analysis. This is done by clicking on one of the verification status symbols at the tree leaves. Unexecuted paths can not be selected.

3.4.3.4 Local branches

Except for the graphical presentation, local branches can be treated very much the same way as normal branches, i.e. they can be collapsed and zoomed in on.

3.4.4 Execution trace view

A trace is preferably displayed in a tree view control which reflects its nested structure.

The multi-path execution trace of a method can grow very rapidly and displaying it as a whole would quickly exceed the limits of what is still practical for the user to obtain a good overview. In that respect, tree view controls help by giving the user the possibility to control the level of detail that is shown by collapsing and expanding sub-trees.

However, showing multiple execution paths of a method at the same time adds too much complexity to the trace: In section 2.3 we have seen the example of a multi-path trace and its disadvantage that the paths are overlapping each other. This is not suitable for a presentation to the user. Instead, the user will only be presented single-path traces, with the possibility to easily switch between paths.

```
produce precondition
   PRODUCE i != j
check postcondition
   PRODUCE r > i
execute body
   EXECUTE if (i > j)
      evaluate condition
         EVALUATING i > j
      execute then branch
         EXECUTE r := i - j
   EXECUTE if (r < i)
      evaluate condition
         EVALUATING r < i
      execute else branch
         (empty)
consume postcondition
   CONSUME r > i
```

Figure 3.7: Simplified execution trace of path $i > j \wedge \neg(r < i)$ of an example program

3.4.4.1 Path selection

The user can select the path whose trace is to be displayed using the execution tree view (as described in section 3.4.3.3). In order to give the user the possibility to switch branches directly inside the trace view, two radio boxes to choose between the *then* and the *else* branch can be shown as in figure 3.8. Of course, switching branches has consequences not only for the selection of the code block, but also for the complete remainder of the execution.

This does not hold for pairs of local branches. Both branches are always included in the trace as they are both part of the execution path (they can of course be collapsed, though).

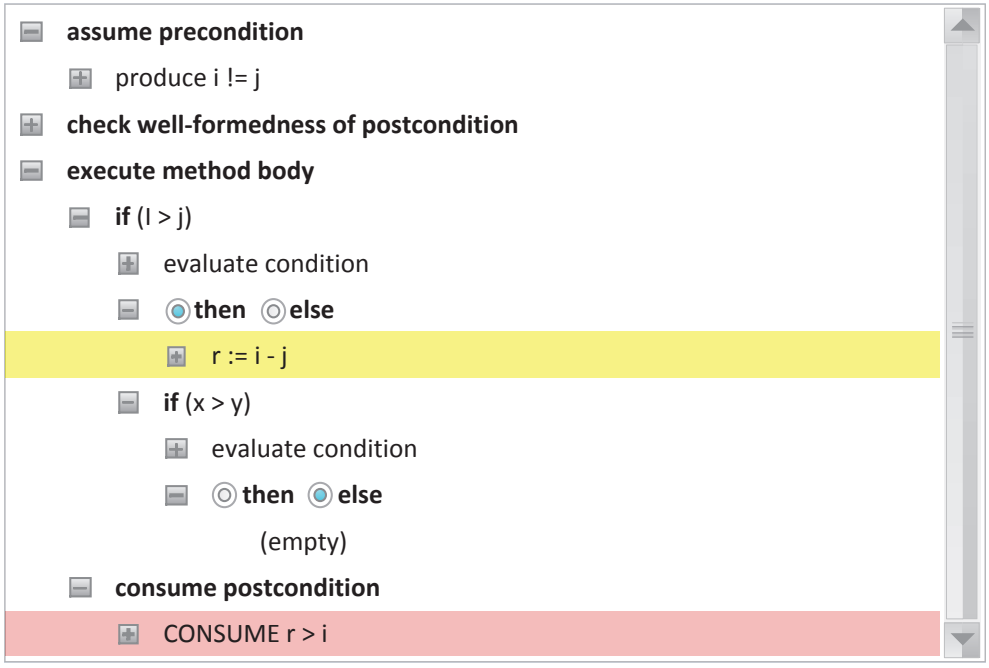


Figure 3.8: Visualization of an execution trace of the path $i > j \wedge \neg(r < i)$

3.4.4.2 Trace hopping

The execution trace view can be used to jump to any step at any level of nesting in the trace.

3.4.4.3 Collapsing

Every composite step can be collapsed or expanded. Special shortcuts can help to find the steps of interest quickly, e.g.

- expand all steps
- expand all steps all up to a certain nesting level
- expand steps that lead to the error
- collapse all less important steps (e.g. well-formedness checks)

3.4.4.4 Current step and error location

In the trace view, the current step in the execution and the step where an error occurred are highlighted, for example with yellow and red background, respectively (see figure 3.8). If the current or erroneous step is not visible because a sub-tree containing it is collapsed, the root step of the collapsed sub-tree is highlighted instead.

3.4.4.5 Remove uninteresting steps

- The production or consumption of a conjunction is decomposed into the production/consumption of each conjunct, where the conjuncts can again be conjunctions, etc. The decomposition of conjunctions is repetitive and uninteresting, since the state is not changed[1]. All steps that only deal with decomposition can therefore be omitted.
- The evaluation of atomic expressions, such as literals and variables, are usually only technical details and the inclusion of such steps would rather clutter the execution trace than provide a benefit to the user.

3.4.4.6 Insert descriptive steps

Knowing whether a step is evaluating, executing, producing or consuming often does not reveal its actual purpose. E.g. in order to check the well-formedness of a method's postcondition, it is produced into an empty heap as part of the verification of the method. Seeing the corresponding *producing* step in the execution trace does not make this clear. Like that, there are many cases where additional information about the purpose of a step should be shared with the user. Figure 3.8 shows how descriptions could be integrated into the step trace. They can be seen as special descriptive steps that are inserted into the trace.

The following list gives some examples of valuable descriptions:

- Assume precondition
- Consume postcondition
- Check well-formedness of post-condition/predicate body
- Execute method body
- Evaluate condition
- Execute then/else branch
- Consume invariant
- Check loop invariant preservation
- Execute loop body
- Check loop invariant establishment
- ...

[1] Technically, this is not completely true because snapshot terms are created and *combine* terms are inserted into the path conditions. This state modification will appear as an effect of the first child step.

3.4.5 Code view

The code view's most important task is to simply show the user's code. Ideally, the code view is also a full-fledged code editor as found in many IDEs, providing features like syntax highlighting, code completion, quick navigation within the code, refactoring etc. All these features can be implemented and used very much the same way as in conventional debuggers, provided the support for the Chalice language is added. We will not expand on them and rather focus on aspects of the code view that are specific to symbolic execution.

In the following sections, important aspects of the code view are discussed and finally a proposal for a user interface is presented in section 3.4.5.6.

3.4.5.1 Current step

Each step of an execution trace can be assigned a direct correspondent in code. For example, the consumption of a precondition corresponds to the specification of that precondition in the *requires* block of the method. To visualize this correspondence, the code of the current step is highlighted with a yellow background.

The nesting structure in the trace often originates from the code structure and thus can be directly transferred back to the code when highlighting the code belonging to nested steps. This has the consequence that in these cases the code belonging to a composite step consists of the code belonging to the step's children (plus possibly some *glue* code). For example, the code belonging to the evaluation of a conjunction consists of the code belonging to the evaluation of the conjuncts plus the conjunction operator.

However, this is not the case with, for example, method or function calls, because they include the consumption of the callee's precondition and the production of the callee's postcondition. This would make the callee's specification a part of the code belonging to the *call* step, which is clearly not desirable.

3.4.5.2 Erroneous step

Similar to the highlighting of the current step, the code corresponding to the erroneous step is highlighted with a red background. A tooltip contains the precise error message.

3.4.5.3 Execution path

During stepping, only a single execution path is considered at a time. We will now discuss ways to visualize a path within the code view.

For this purpose let us look at the example in listing 3.1. A function `f` in class `C` is called twice in method `main`, the result of each call is stored to the field `i`. From a static perspective, these are just two calls of the same function `f`. From a dynamic perspective, the calls need to be distinguished because of the state of the target object changes between the two calls. When we analyse two evaluations of

```
class C {
    var i: int

    method main(v: int)
        requires acc(i)
        ensures acc(i) && i < 0
    {
        i := -1
        i := f()
        i := f()
    }

    function f() : int
        requires rd(i)
    {
        i > 0 ? -1 : 1
    }
}
```

Listing 3.1: Example for call instances

function `f`, we realize that in the first evaluation, the *false* branch is taken, while in the second evaluation it is the *true* branch. The two calls thus result in two completely different evaluations of the function.

Using this example, we can already make two important observations:

1. It is not enough for the visualization of a path, to show every method, function or any other declaration just once. Thus, we need to find a way to display the code of a method or a function[2] as many times as it is executed within the execution path. Let us call the different executions of a method or function *call instances* of the method or function.

2. Conditionals cause parts of the code not to be contained in an execution path. Specifically, the code specific to branch which is *not* taken, is completely ignored. For example, if the execution path contains the *true* branch of an *if-then-else* statement, the *else* block is skipped. Our visualization must include the information which code is part of the execution path and which is not. Obviously, the code that is part of the execution path is given priority.

Figure 3.9 shows a first, simplistic attempt of the visualization of the execution of the method `main`, incorporating the above observations.

[2]The same holds for the execution of the bodies of predicates and monitor invariants. Without loss of generality, we omit to mention all similar cases for the sake of brevity.

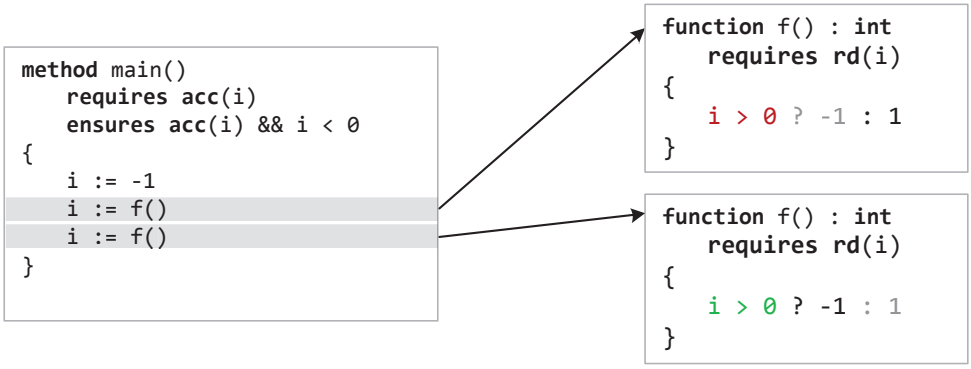


Figure 3.9: Path visualization of example 3.1 based on the idea of code bubbles [BRZ+10]

1. The declaration of function `f` is shown twice, once for each call. More precisely, each call instance of the function `f` is shown once.

2. The conditionals are coloured to indicate which part of the code is contained in the execution path and which part is not. The simple rule for the colouring: Black code is executed, grey code is not executed. Additionally, the condition is coloured green or red, indicating whether the path assumes the condition to be *true* or *false*, respectively.

We will now discuss both of these aspects from a more general perspective.

Code Bubbles We have seen that a call is not just referencing a callee declaration, but a call instance. We can further define the components that identify call instances: their declaration (for example, the function), a target object (with an object state) and, if any, the arguments. For the understanding of the execution of a call instance, this information is crucial.

Figure 3.9 presents the code of the call instances of function `f` within two separate boxes. The boxes are pointed at by arrows from the caller at the line of code, where the call is executed. The idea of this visualization is based on *Code Bubbles* [BRZ+10], which originally is a concept for a new kind of code editor opposed to the traditional paradigm "one file - one editor". Declarations are placed into *bubbles* which then can be dynamically and independently arranged on a canvas, according to any imaginable measure of (conceptual) proximity that may be helpful for the user in the current context.

Let us discuss a use case for code bubbles which is also relevant for our concerns: Calls cause the control flow to jump from the caller's to the callee's code. In a traditional, "one file - one editor" context, this has the consequence that the focus jumps randomly within a file, or, even worse, within multiple files of a project. Thus, if the user follows the control flow in an execution, the caller's declaration

is probably not visible any more. This poses a challenge for the user not to lose orientation. The approach of code bubbles provides a solution to this problem by dynamically rearranging the declarations on the canvas. For example, a method call creates a very strong relationship between caller and callee at the moment of the call. Before or after the call, other relationships between declarations may be more important. Accordingly, the caller and callee code, both within bubbles, are moved closer to or away from each other.

Our use of code bubbles focuses on the aspect of call dependencies between declarations. Additionally, we strictly require one bubble per call instance, whereas the concept of code bubbles in general allows any number of bubbles containing a declaration.

The execution of a path starts with only the executed method or function placed in a bubble. As soon as the execution calls a call instance, a new bubble containing the call instance's declaration is placed near the bubble of the caller bubble. The bubbles are connected with an arrow, symbolizing the call. This procedure is repeated for every call. As the number of bubbles on the canvas grows, it gets more important to rearrange them, so that the bubbles involved in the most recent calls are placed close to each other. Also, we can recognize a call stack, containing of those call instances which are currently executing. The bubbles of these call instances should be prominently placed on the canvas and highlighted accordingly.

Colouring of conditionals To reflect the selection of the path in the code view, code that is not part of the path is faded and conditions are coloured according to their assumed boolean value. Figure 3.10 shows examples of all types of conditionals, each of them in two formattings, one for the *true* and one for the *false* case. The rule for the colouring is the same for all conditionals: Green is used for conditions assumed to be true, where red is used for conditions assumed to be false. The black code is part of the selected execution path, while the grey code is not.

3.4.5.4 Watch expressions

Watch expressions are known from conventional debuggers. They are used to observe the value of an expression over time during an execution. Usually, the feature provides means to define watch expressions in an extra UI view. Whenever the execution is paused, the watch expressions' values are evaluated and displayed. The very same mechanism is possible for symbolic execution. There is, however, a more specific use case for which watch expressions are particularly interesting in our environment:

One of the most commonly occurring errors are missing access permissions for heap locations. In such a case, the questions are: Why are the permissions missing? Were they initialized properly? Were they consumed on the way? Or, generally speaking: In my program, which steps of the execution had an influence

a)

```
next != null ==> next.valid
next != null ==> next.valid
```

b)

```
next == null ? true : next.valid
next == null ? true : next.valid
```

c)

```
if (next != null) {
    call next.dosomething()
}

if (next != null) {
    call next.dosomething()
}
```

d)

```
if (next != null) {
    call x := next.dosomething()
}
else {
    x := 0
}

if (next != null) {
    call x := next.dosomething()
}
else {
    x := 0
}
```

Figure 3.10: Colouring of conditionals in the code view: a) implication, b) ternary *if* operator, c) *if-then* block, d) *if-then-else* block.

on the availability of the permissions in question? This could be answered by a boolean watch expression that evaluates to *true* if the permissions are available and *false* otherwise [3], giving us the desired information with a certain granularity (depending on how often the expression is evaluated).

Because this is such a frequent use case, we call these special watch expressions *permission watch expressions* and propose to add the option to display this information in both the code and the trace view. For example, a thin vertical bar coloured red and green could indicate whether a selected heap chunk can be accessed at certain locations. Figure 3.11 shows the integration of this feature into the code view (item 5). The integration into the trace view works similarly.

3.4.5.5 Code editing

The code bubbles approach adds some complexity to the task of editing code. Unlike a file-based editor, code arranged in bubbles does not have a static linear ordering any more. In particular, the same code can appear in multiple bubbles at once in case of multiple calls of a single callee, resulting in multiple call instances. The arising problems could be avoided by making code bubbles read-only and providing a separate, conventional editor. Yet, the code bubbles project [BRZ+10] equips bubbles with full-fledged editor functionality. We do not discuss this in more detail as it is not specific to symbolic execution or verification.

3.4.5.6 Putting it all together

Figure 3.11 visualizes the concepts discussed above.

1. **Code bubble frame with title bar**

   The title bar includes the symbolic name of the target object and its type. The background colour of the title bar is unique for each target object or for its type, if too many objects are involved for the colours to be distinguishable.

   A bubble contains exactly one code element, here a method. The code is presented conventionally with syntax highlighting. A vertical grey bar (*marker bar*) can be used to place markers or line numbers.

2. **Breakpoint**

3. **Erroneous step**

   Marked by a symbol in the marker bar and red background in the code.

[3] Technically, there is no way to write such an expression in Chalice because access expressions like `acc(x)` or `rd(x)` are not boolean expressions and can only be used in assertions. Instead, one would have to assert the expression and construct the boolean value depending on the outcome.

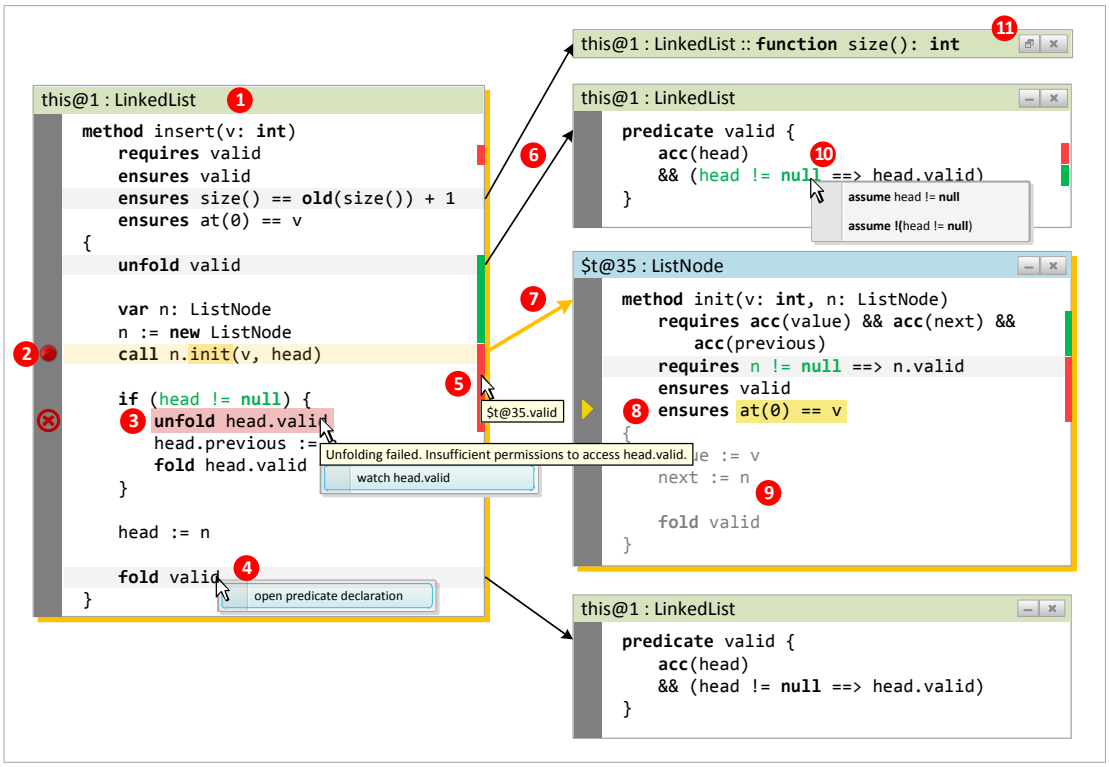


Figure 3.11: Code view

4. **Open declaration manually**

   Using a context menu item, any reference in the code can be opened, even if it is not part of the code executed so far.

5. **Permission watch expressions**

   The red and green vertical bars indicate for each line of code, whether a specified heap chunk is available in the last step completed on that line (as described in section 3.4.5.4). The heap chunk is selected using the context menu (here, the predicate chunk `$t@35.valid` referenced in the erroneous step is selected).

6. **Calls**

   Black arrows between bubbles denote calls of any kind.

7. **Call stack**

   The current call stack is highlighted with yellow colour. This includes call instances that are on the call stack (marked by yellow drop shadow) and the calls (yellow arrows).

8. **Current step**

   The current step is highlighted with yellow background and a yellow marker in the marker bar of its containing bubble.

9. **Shaded code**

   Code that is not directly relevant in the current context is shaded. Here, the body of a called method is not relevant, because only the specification of a callee is considered. Branches of *if-then-else* statements, that are not executed because they are not contained in the current execution path, are also shaded this way.

10. **Condition highlighting**

    Conditions are highlighted as described in section 3.4.5.3.

11. **Minimizing and closing bubbles**

    Each bubble can be minimized to its title bar or closed entirely using the buttons in its title bar.

3.4.6 State views

The state views have the purpose of displaying (intermediate) symbolic states. For every step within an execution trace, the pre- and the post-state is recorded. Using these, we can provide the user with an interface that gives them an understanding of each step's effect on the state.

In order to find appropriate visual representations of a state, we first analyse its elements and their relationships: As discussed in section 2.6, a state consists of local variables, a current and an *old* heap, and the path condition. We can interpret a heap as a graph of heap objects. A *heap object*, in our context, is an abstraction of an object, that is a class instance, very similar to how the symbolic heap is an abstraction of the physical heap in memory. Heap objects are not defined directly in Syxc's internal representation of the symbolic heap. Instead, their existence can only be deduced from local variables and object fields referencing them. Such a reference to a heap object is represented by a symbol term. We call this symbol term the *symbolic name* of an object. Object fields are defined by heap chunks. They hold a reference to their containing object, that is their *receiver*. Object fields of a reference type additionally hold a reference to their target object. Predicate chunks can be handled very much like object fields. For example, they are also associated with a receiver.

A unique property of object fields (including predicates) is that they are protected from unpermitted access. Since a symbolic heap is only a projection of the currently available permissions onto the physical heap, every object field in a symbolic heap is at least readable. Still, it is an important task for the visualization of a heap to include the exact amount of permissions to every field.

Local variables point to a symbolic value, which can either be of a value type (such as *integer*) or a reference type, very much like object fields. Also, local variables are the entry point for any access to the heap. In particular, the access to fields of the current object is done through the implicitly declared local variable $this$, even if the use of the keyword `this` is not enforced by the language. These observations lead us to the conclusion that local variables are heavily connected with the heaps. This will influence the way they are presented to the user. Unlike object fields, however, local variables do not need permissions to be accessed.

The path condition, consisting of boolean terms, defines additional knowledge about symbolic values. These symbolic values may also be used within the heap. For example, an integer-typed object field could have a symbolic value t and the path condition could contain a formula $t > 0$.

In the following subsections we now describe different representations of different parts of a state, which together are referred to as the state views.

3.4.6.1 Variables view

The *variables view* is a structured textual representation of local variables and a heap. It is known in a similar form from conventional debuggers. Figure 3.12 shows an example of a variables view. We can see that it visualizes a tree structure[4], where every node is associated with additional information arranged in multiple columns. Nodes represent symbolic values (including references).

The outermost level of the structure is formed by the local variables, or more strictly, their values. All other nodes are formed by the values of heap chunks. In the case of references, all fields of the reference's target object form the child nodes of the node representing the reference. Note that we also include nested chunks in our visualization. Even if nested chunks are not available for direct access, they are an integral part of the knowledge about a heap and should therefore be presented to the user. To distinguish nested from direct chunks and to symbolize the indirect nature of the former, the text of nodes representing nested chunks is coloured grey, while others are black.

The first column from the left, that also visualizes the nesting structure, gives every node a meaningful name: The outermost nodes are named after the local variables. All other nodes are named after the object fields they represent. This way, by following a path from a local variable to a nested node, we can construct the access path to the according object field. For example, the fifth node from the top in figure 3.12, named *V*, can be accessed by *this.left.V*.

The second column contains the type of the value represented by each node. This may be either a reference type, a value type or the name of a predicate. For example, *this* has the type *Tree*, *this.value* has the type *integer* and *this.left.V* is a predicate *Tree.V*.

[4]Since the outermost level consists of multiple nodes, the tree structure is only complete if we think of an imaginary root node, with the outermost level nodes as child nodes.

The third column contains the actual symbolic value of each node. In case of predicates, the snapshot is shown in by brackets.

Finally, the last column shows the permissions that are available to each field object. Note that local variables are not access protected, thus no indication of permissions is given for them. In the case of nested chunks, the given permissions represent the permissions that would be available, if its containing predicate chunk was unfolded. In our example heap, *this.right* is an example of a *partially nested chunk*. This means, that the heap contains both direct and nested chunks for the same heap location. In Syxc's heap representation, both chunks are treated independently. We combine them into one node and indicate the aspect of it being partially nested with the notation *(50,0)+(50,0)*, meaning 50% write permissions are available directly plus 50% write permissions are available indirectly.

	type	value	permissions
this	Tree	this@1	
rightV	Tree.rightV	[Combi...]	(100,0)
value	int	$t@23	(100,0)
left	Tree	$t@24	(50,0)
V	Tree.V	[$t@20]	(100,0)
right	Tree	t@2	(50,0)+(50,0)
V	Tree.V	[$t@10]	(100,0)
leftV	Tree.leftV	[$t@11]	(100,0)
rightV	Tree.rightV	[$t@9]	(100,0)
t	Tree	t@2	
V	Tree.V	[$t@10]	(100,0)
leftV	Tree.leftV	[$t@11]	(100,0)
rightV	Tree.rightV	[$t@9]	(100,0)

Figure 3.12: Variables view

3.4.6.2 Heap graph view

The heap graph view is another way of visualizing a heap. It contains the roughly the same information as the variables view, but from a different perspective. Figure 3.13 shows the graph of the exact same heap as seen in figure 3.12.

The dominant elements of the heap graph are heap objects. Their visualization loosely follows that of objects in UML diagrams: A rectangular box, containing a header (with grey background, showing the object's symbolic name) and lists of (value) object fields and predicates, divided by horizontal lines. Object fields of a

reference type are not listed within the box. Instead, they are presented as arrows connecting the receiver with the target object. Value fields, predicates as well as reference fields are annotated with their respective permissions in the same way as we have seen with the variables view.

Local variables are also included in the heap graph view. They are represented by white rectangular boxes, separated from the heap objects by a dashed vertical line. Arrows from the local variables' boxes to their target objects' boxes denote reference-typed local variables. Value-typed local variables contain their symbolic value within their box.

As in the variables view, grey text is used to indicate nested chunks. Figure 3.13 also contains an example tooltip appearing when the user places the mouse over a label indicating (partially) nested permissions. That way, the user can be provided with the information about which predicate chunk is containing the nested permissions.

In figure 3.13, we can identify a heap object with a dashed frame (with the symbolic name $\$t@22$). This symbolizes a *lost* object, that means one that is not reachable through any access path. Of course, this also has the consequence that this object can not be referenced in code within the current context. There may, however, be other references to that object that are not contained within the current (partial) heap. Since permissions, unlike references, can not be shared[5], they are irrevocably lost, though. It is therefore considered useful to know of the existence of such unreachable objects. It may indicate a problem if permissions are lost like this.

3.4.6.3 Comparison of variables view and heap graph view

When we compare the two discussed representations of the heap, we can identify one major difference: The variable view is access path based, while the heap graph view is object based.

The variable view is access path based because it follows access paths beginning with the local variables and shows every object (or rather its fields) it encounters. This has three consequences:

1. Objects that are unreachable, are not included. We have seen in the heap graph representation, that the object $\$t@22$ is not reachable. In the variable view, it does not appear.

2. Objects can be contained multiple times. For example, in our heap, the object $t@2$ is contained twice because there are two access paths for it: *this.right* and *t*.

3. Cyclic object dependencies are not detected. Such cyclic dependencies cause access paths to be potentially infinite. For example, if an object A references

[5](Read) permissions can be partitioned and distributed among different call instances, but all partitions have to be recollected to gain write permissions to a heap location again.

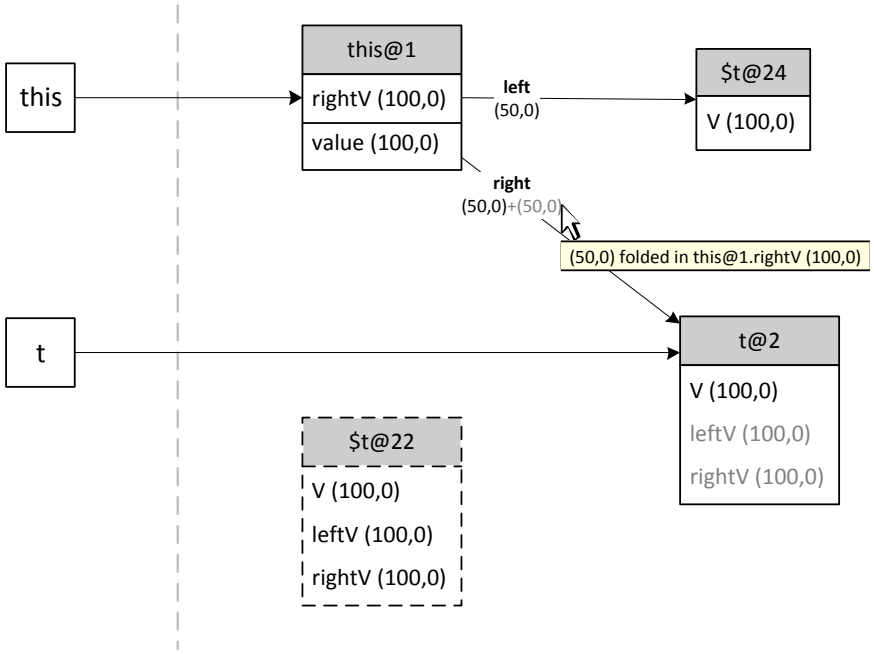


Figure 3.13: Heap graph view

the object B through the field b, and the object B references A through the field a, the variable view will show an endless access path *a.b.a.b.a.b* and so forth, provided one of the objects A and B are reachable through a local variable. This problem is solved in the variable view by expanding access paths lazily, using the possibility to display nodes in a collapsed state and let the user to expand it manually.

The heap graph view, on the other hand, is object based, meaning that heap objects are the primary source for what is displayed. The three identified problems with the variables view are all solved in the heap graph view:

1. It is not required for a heap object to be reachable in order to be included in the graph.

2. Every object is only displayed once, independent from how many access paths there are

3. Cyclic object dependencies are visualized naturally using arrows.

In summary, it can be said that the problems arising with the variables view are due to the fact that the heap has the structure of a (general) directed graph which is forced into a tree structure in the variables view. Obviously, the heap graph view is suited much better to represent a heap.

3.4.6.4 Old heap

The variables view and the heap graph view should both offer the possibility to inspect the *old* heap[6].

3.4.6.5 Path condition view

The *path condition view* has the sole task to give the user all information contained in the path condition. The latter consists of boolean terms. We have seen in section 3.2.1 that terms are at the lowest level of the decomposition hierarchy of a program and its execution. This also means, that terms can be conceptually far away from the program, which is the user's main focus. For example, symbol names generated by Syxc are usually not meaningful to the user. It is therefore particularly important for the path condition view to find connections between this internal concept of terms and the code.

Figures 3.14 and 3.15 demonstrate possibilities how to do this. They show two versions of the same path condition. The raw version (3.14) presents the terms in an equivalent form to the internal representation. If combined with the knowledge of the local variables and a heap, we can obtain an optimized representation (3.15) much more meaningful to the user. We will now go through the optimizations done to achieve this result:

Let us assume a store

$$\gamma = (this \rightarrow this@1,\ next \rightarrow \$t@27)$$

and a heap

$$h = (this@1.x \rightarrow \$t@3,\ this@1.y \rightarrow \$t@4,\ this@1.z \rightarrow \$t@5)\ .$$

The first term `Equals($t@27, null)` allows to optimizations: First, the equality operator should look the same as in the programming language in order to comply with our principle 2. This gives us an optimized version `$t@27 == null`. We now consult our store γ and realize that the symbol $\$t@27$ is the value of the local variable $next$. By replacing the symbol name in our equality term with the name of the local variable, we obtain `next == null`. The second term `Not(Equals(this@1, null))` is optimized similarly. As an additional optimization, the nested operators `Not` and `Equals` are combined into the $!=$ operator.

The third term GreaterThan($t@3, 100) requires us to not only consult the store γ, but also the heap h to find an access path to the referenced symbol $\$t@3$. The access path $this.x$ is then further reduced to x, since the receiver $this$ can be assumed for unqualified references in case no equally named local variables are within the present scope. This gives us the optimized representation `x > 100`.

[6]Technically, both pre- and post-state of a step contain an *old* heap. However, they are always the same.

```
Equals($t@27, null)
Not(Equals(this@1, null))
GreaterThan($t@3, 100)
Not(Equals($t@4, $t@5))
GreaterThan(FApp(size, this@1, $t@10, List()), 0)
Equals($t@10, Combine($t@11, $t@5)))
Equals($t@11, Combine($t@3, $t@4))
```

Figure 3.14: Example of a path condition in a raw string representation

The fourth term `Not(Equals($t@4, $t@5))` is optimized with a combination of already discussed techniques. We obtain `y != z`.

The last three terms are combined into only one resulting term, as we notice from our comparison. To understand this, we first analyse the term `FApp(size, this@1, $t@10, List())`: It represents the value that is returned by the call to the function $size$ on the receiver $this@1$, with a snapshot $\$t@10$ and with an empty list of arguments. This is called a *function application term*. We have already encountered snapshots in the discussion about predicates. Snapshots with function applications work equivalently[7]. We can now see that the last two terms of our path condition are defining the snapshot used in this function application term. Since *combine* terms are binary, they are nested if more than two terms need to be covered by the snapshot. For our optimized version, we omit to mention the combine terms ($t@10$ and $t@11$), but we show a flattened version of the snapshot containing the terms $t@3$, $t@4$ and $t@5$, resolved to their canonical names, x, y and z, respectively. For the function application term, we thus obtain `size()[x, y, z]` (omitting the receiver $this$), and for the whole term `GreaterThan(FApp(size, this@1, $t@10, List()), 0)` we obtain `size()[x, y, z] > 0`. We omit the last two terms of the raw version in our optimized version.

Figure 3.16 presents a user interface for the path condition view. It demonstrates two more ideas of how to facilitate the understanding of a path condition: First, the previously shown optimized string representations can be syntax highlighted in order to obtain a representation even closer to the code. Second, filters can be applied to restrict the amount of terms displayed. In complex programs, the number of terms can be to many to show them all. Also, not all kinds of terms may be equally important to what the user needs to understand.

[7] The rationale behind snapshots in function application terms is the following: They define the state of the receiver object for which the function application term represents the return value of the corresponding function call. This is a way to express the return value of a function without explicitly evaluating the function's body. For example, this is necessary with recursive functions, since they can not be fully evaluated with a guarantee that the recursion stops after a finite number of steps. Syxc therefore evaluates recursive functions only once and the containing recursive call is abstracted using such a function application term.

```
next == null
this != null
x > 100
y != z
size()[x, y, z] > 0
```

Figure 3.15: Optimized string representation of the path condition in figure 3.14

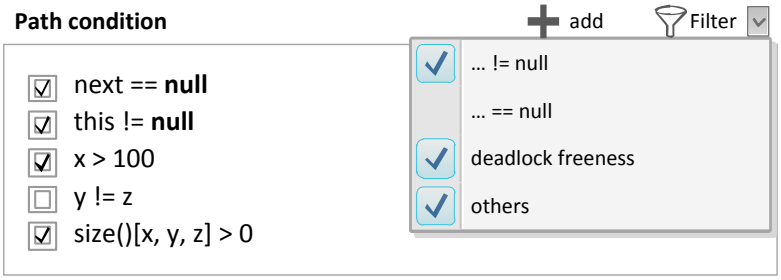


Figure 3.16: Path condition view

3.4.6.6 State diffing

Bearing in mind our principle 4, we now discuss ways to show the immediate effect of a step on the symbolic state by displaying a combination of the step's pre- and post-state and highlighting their differences.

We identify three general types of modification that can happen to a state: A new element is added, an element is removed or an element is altered. Here, the elements could be either heap objects, heap chunks, heap references or local variables. Path condition terms are never removed from the path condition or altered within a path. Therefore, only the addition of a term is a valid modification of the path condition.

The three visualizations of the state, namely variables view, heap graph view and path condition view, can now be extended to provide a comparison of pre- and post-state. This is done by showing a combined version of the pre- and post-state, containing the union of local variables and heap chunks (in case of the variables view and the heap graph view), and terms in case of the path condition view. Added, removed and altered elements within the union are marked appropriately, for example using colour.

3.4.6.7 State manipulation

The manipulation of any intermediate state during the simulation of an execution path has the consequence that the remainder of the path needs to be executed again.

The three presented state views can be extended to support state manipulations. We identify several possibilities:

Disabling path condition terms: Figure 3.16 contains check boxes for every term of the path condition. These could be used to disable a term temporarily. This means that the resulting state's path condition does not include this term any more and the execution is thus resumed without it. Still, it is kept as an entry in the user interface to facilitate the re-enabling at a later time.

Disabling heap chunks: The same effect as with disabling path condition terms could be achieved by disabling heap chunks using check boxes in the variables view.

Adding/Editing: We can also easily imagine the possibility for heap chunks and path condition terms to be added and edited. For example, the value of a heap chunk could be changed to some other symbolic value. Here, it the question arises, which language is use for the user input. Since heap chunks and terms do not have direct equivalents in the programming language[8], we must either define a new language to support the entry of chunks and terms, or, better, extend the programming language to support those aspects of chunks and terms that can not be expressed. There is also the possibility to provide something like a command line that can be used to change the state. There, the normal programming language could be used.

Graphical editing: Advanced possibilities for heap modifications could be provided by the heap graph view. For example, the connections between heap objects could be changed in a drag and drop fashion.

[8]For example, there is no syntax in Chalice for snapshots, since they are a completely internal concept.

4 Implementation of a symbolic execution debugger

Parts of the above concept have been implemented in a tool named Syxc IDE based on the Eclipse platform. It provides basic development support for the Chalice language and a debugging environment with Syxc as its backend.

SyxcIDE is mostly written in Scala [Ode11], but uses several Eclipse-based Java libraries. Syxc is written entirely in Scala.

Figure 4.1 shows an overview of the components involved in Syxc IDE and their dependencies. The Syxc verifier itself builds the base of the system. It has been extended with execution tree and trace recording facilities (section 4.1). Chalice is used by Syxc and the Chalice Language Plugin only to parse and resolve Chalice code. The Chalice Language Plugin (section 4.3) provides language support within the Eclipse IDE. The Syxc Debug Interface (section 4.2) acts as an interface between the debugging user interface and the Syxc verifier. And finally, the Syxc Debugger Plugin (section 4.4) provides UI modules specific to debugging Chalice programs and integrates them into the Eclipse IDE and the Chalice language environment.

The following sections describe important aspects of the implementation of the mentioned components.

4.1 Extensions of Syxc

4.1.1 Definitions

4.1.1.1 History

A *history* is the collection of all data gathered during the execution of one verifiable element. That includes the execution tree, the (multi-path) execution trace, all intermediate symbolic states and status information including, if any, failures and warnings.

4.1.1.2 Branch tree

During an execution, there is always an *active branch*. The execution is started with a so-called *root branch* as active branch. A branch is split into two sub-branches if the execution is split into two paths. The two sub-branches form the child nodes of the split branch, denoted by *true* and *false branch*. They are also

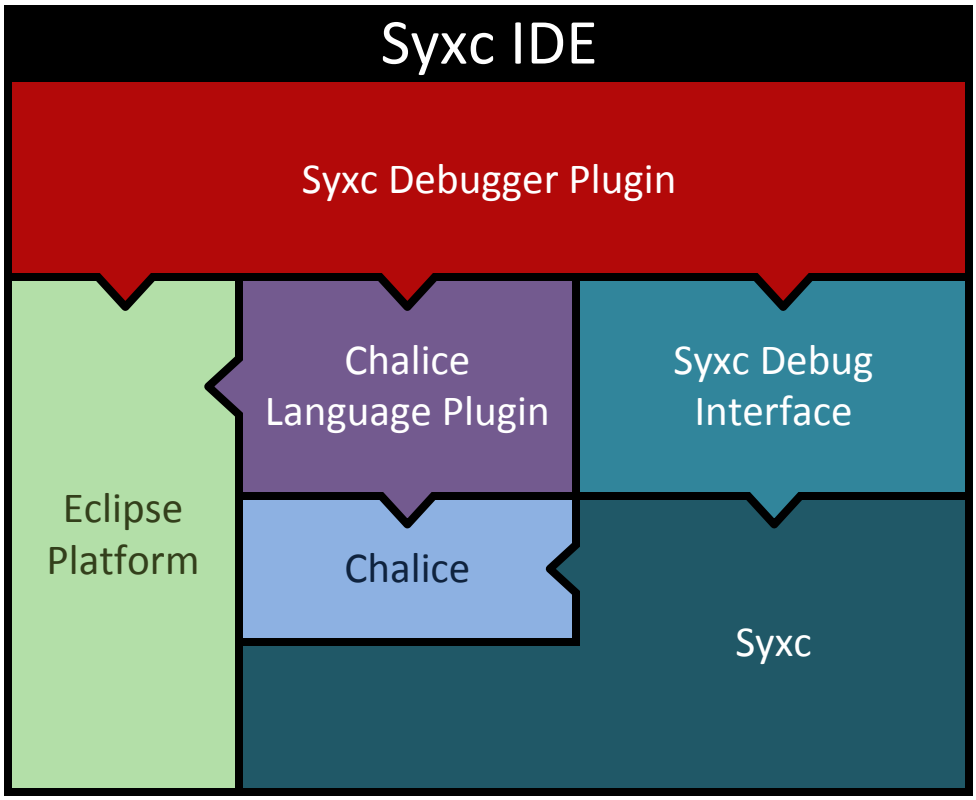


Figure 4.1: Syxc IDE components diagram, arrow symbols denote dependencies

referred to as the *twin branches* of each other. The two paths are then executed with the *true* and the *false* branch, respectively, as active branches. The complete execution thus yields a *branch tree* whose leaf nodes represent the execution paths.

Local branches within an execution path are referenced by pairs by the branch that was active at the time they were created, but they are not its child branches, thus they do not interfere with its sub-branching hierarchy. Instead, local branches form the roots of their own branch tree, which is built up in the case of global branches within the execution of the local branches, as described in section 2.4.4. We therefore also call the local branches *local roots* with respect to their *local trees*. The leaf branches of local branch trees represent *local paths*. Figure 4.2 shows an advanced example of nested branches, containing a pair of local branches (in blue), one of which contains a pair of global branches. $p_{1a}$and p_{1b} are the resulting local paths. We can also see in this figure that the joining of the local branches gains complexity in this case, as it must consider both local paths.

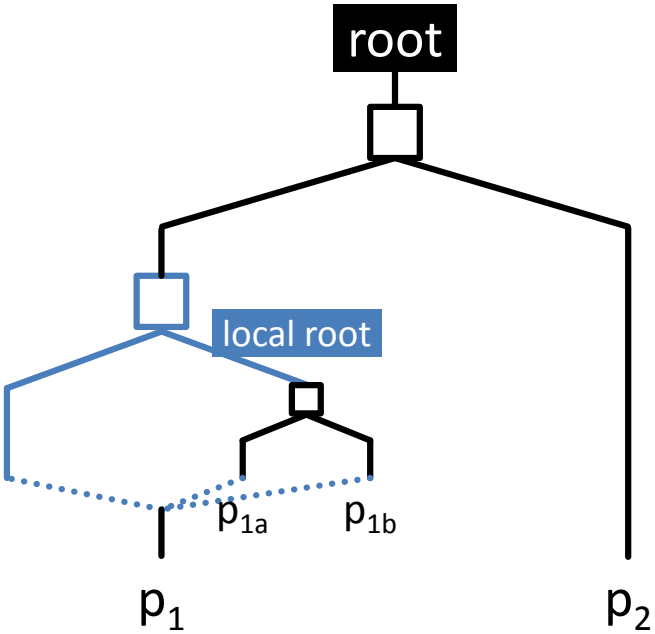


Figure 4.2: Abstract example of a nested branch tree

4.1.1.3 Step trace

A *step* is the smallest recorded unit of the symbolic execution. They contain a list of sub-steps[1]. Also, they hold a reference to the branch that was active, when they were created. The series of steps that are recorded during the execution of a verifiable element forms the execution trace, or *step trace*.

Basic step types are the four symbolic execution steps: Evaluating, Consuming, Producing and Executing. For these steps, the pre- and post-state is also recorded. Besides the symbolic execution steps, there is a number of special types of steps:

Branching steps: They mark the position of a split of the execution into two sub-branches. There is a branching step for each sub-branch, both being the first step in the trace of the respective sub-branch.

Description steps: They are inserted into the trace to conceptually group a number of successive steps and provide a textual description for better understanding.

Root step: The execution of a verifiable element is started with an empty trace which is represented by a root step. The top level steps of the execution are listed as children of the root step.

[1]The semantics of the parent-child relationship of steps is the following: The execution of sub-steps contributes to the completion of their parent's task.

4.1.2 Integration with the verification algorithm

4.1.2.1 Branches

The continuation-passing style of Syxc's implementation causes the control flow of the program to follow execution paths. Backtracking is only used to return to where the control flow should split. This is suitable to keep track of the *active branch*. This is the task of the *branch keeper*. Its sole responsibility is to hold a reference to the active branch. In case the execution is split up, the branch keeper as well is split into two copies, now referencing the two children of the prior branch, respectively.

Since there is an existing *context* object (with different other implementation-specific tasks not relevant for this discussion), which is passed along the same paths as the branch keeper needs to be passed, it is chosen as the implementer of the branch keeper's interface[2].

4.1.2.2 Step trace

For the construction of the step trace, the basic idea is to keep a reference to the *current step* and pass it as an argument to methods that are expected to construct its sub-steps. However, the control flow of the continuation-passing style is not suitable, since sub-steps of a parent can be constructed within arbitrarily nested continuations and not just one level down from where the parent step was created. Still, Syxc's current implementation easily allows the construction of the step trace because all continuations that need to add sub-steps to a parent step are defined as closures within the same method, giving the possibility to access the reference to the parent step from a local variable. The following example demonstrates this technique.

Example Let's take as an example the production of an *And* expression (listing 4.1). When `produce` is called, a trace view `tv` is passed as an argument. A *trace view* represents the reference to a certain step within the trace, that is, the *current step*, and includes other logic that facilitates the construction of the step trace[3]. To add a `Producing` step as a child, `tv.stepInto` is called (line 2), resulting in a new trace view which establishes the newly added step as the current step. In the actual production of the *And* expression, its left-hand side is produced first (line 5) with a recursive call to `produce`. The resulting state `s1` is used to produce the right-hand side (line 6) into state `s2` which is passed to the continuation function `Q`. Note how the context is passed along the continuation functions while the newly created trace view `tv1` is passed to both `produce` method calls, indicating that

[2]Technically, a trait `BranchKeeper` containing all the functionality of the branch keeper, is mixed into the context object's implementation.

[3]The trace view's relationship to the current step is similar to the relationship of the branch keeper to the current branch.

```
1  def produce(e: Expression, s: State, c: Context,  tv: TraceView)(Q:
       (State, Context) => Unit) {
2    val tv1 = tv.stepInto(c, Producing(e, s))
3    e match {
4      case And(lhs, rhs) =>
5        produce(lhs, s, c, tv1)((s1, c1) => {
6          produce(rhs, s1, c1, tv1)((s2, c2) => {
7            Q(s2, c2)
8          })
9        })
10     case ...
11   }
12 }
```

Listing 4.1: Producing a conjunction (simplified)

within the production of `lhs` and `rhs` sub-steps are to be added as child steps of the step created on line 2 and represented by `tv1`.

Let's now assume an expression `(acc(x) && (x != null ==> x.valid)) && func(x)` where `x` is of some reference type `X` with a predicate `valid` and `func` is a function that takes an `X`. The left-hand side of the (outer) conjunction, `(acc(x) && (x != null ==> x.valid))`, contains an impure implication, thus its production will split the execution over `x != null`. Looking at the code of the `produce` method again, this means that the anonymous continuation function containing the second recursive call to produce (line 6) is executed twice with two different contexts `c1`, but with the same trace view `tv1`. That way it is ensured that sub-steps are added to the same parent step but within different branches.

4.1.2.3 History

Every verifiable element is verified in isolation, therefore there is a separate history for every one of them. All histories of a verifier run are collected and made available to the external interface.

4.1.3 External interface

Syxc's external interface is extended with a method that sets up and runs the verifier in a special debug mode and returns the collection of histories. Currently, there is no further interaction between the debugger and Syxc. See section 5.2 for an outlook of how this could be extended in the future.

The debug mode was introduced because recording the execution histories of a program has a negative impact on the performance of the verifier. Therefore, two trace view implementations are provided. One of these suppresses any recording.

It is automatically used if Syxc is run from the command line. The other implementation records the histories as expected and is only used if Syxc is started in this debug mode.

4.1.4 Other contributions to Syxc

During our work with Syxc, we have uncovered the following problems:

1. Global branches within local branches are not handled properly. The joining of local branches is done with the assumption that there are no global branches within the execution of the local branches. Listing 4.2 gives an example program that reproduces the bug. Syxc verifies the method `test` successfully, even if the assertion obviously does not hold. In this example, the local branch contains a global branch, which results in two local paths. The program can be verified with the local *false* path taken, but it should fail with the *true* branch taken. However, the true branch is executed first and all of its effects are overwritten by the *false* branch being executed after it. Therefore the program is verified successfully. A working fix for this bug has not yet been found.

2. Recursive predicates caused the execution to run infinitely. Listing 4.3 gives an example of a predicate, whose execution caused an infinite recursion. Although Chalice supports this kind of predicates, Syxc is not able to handle it. The reason for it is, that there is currently no way to stop such a recursion, similar to how recursions in functions are stopped. The newest version of Syxc forbids such predicates and exits with a corresponding error message.

4.2 Syxc Debug Interface

The Syxc Debug Interface (SyxcDI) is the interface between the Syxc verifier and the debugger user interface. Its main tasks are to provide simple access to and navigation through execution histories recorded by Syxc. It also hides certain implementation concepts of histories, such as the branch tree. The design of SyxcDI follows the Java Debug Interface (JDI)[4] to some extent, though being much simpler.

The `VerificationManager` is the entry point of SyxcDI and is basically a wrapper of the Syxc verifier. It can be seen as the counterpart of `VirtualMachineManager`[5] in JDI. Listing 4.4 shows its public interface[6]. Calling `runVerifier` starts Syxc and verifies the supplied program. The result is an instance of

[4] http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html

[5] http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachineManager.html

[6] All code samples in this section are slightly simplified for the sake of brevity. The simplifications mainly affect generic parameters of classes being necessary due to some of Syxc's implementation details.

```
class OrderedLinkedList
{
    var head: OrderedLinkedListNode

    predicate valid {
        acc(head) && (head != null ==> head.valid)
    }

    function size () : int
        requires rd(valid)
    {
        unfolding rd(valid) in head == null ? 0 : head.size()
    }

    method test ()
        requires valid && 0 < size()
        ensures valid
    {
        assert size() == 1 // holds in Syxc, but shouldn't
    }

}

class OrderedLinkedListNode {

    var next: OrderedLinkedListNode

    predicate valid {
        acc(next) && (next != null ==> next.valid)
    }

    function size () : int
        requires rd(valid)
    {
        1 + unfolding rd(valid) in next != null ? next.size() : 0
    }

}
```

Listing 4.2: Example demonstrating a bug in Syxc with nested branching

```
predicate valid {
    acc(value) && acc(next)
    && (next != null ==> (next.valid && unfolding next.valid in
        value > next.value))
}
```

Listing 4.3: Infinite recursion with predicates

```
trait VerificationManager {
  def runVerifier (programText: String, includePattern: String) :
      ExecutionSimulator
}
```

Listing 4.4: Public interface of VerificationManager

`ExecutionSimulator` (listing 4.5), the counterpart of JDI's `VirtualMachine`[7]. The `ExecutionSimulator` provides means to select a history, an execution path and a specific step to debug (using the `change* methods`) and to navigate step-wise within the current step trace (using the `step*` methods). Additionally, for each of these methods there is a `can*` method, which checks the conditions that have to be met in order for the corresponding action to be valid. For example, `stepInto()` is only valid if the current step contains sub-steps. Accordingly, `canStepInto()` returns `true` if this assertion holds, or else `false`. The `can*` methods are supposed to be used by clients to decide whether a UI option should be displayed to the user.

Finally, the three events `treeEvent`, `pathEvent` and `stepEvent` can be subscribed to by clients to get notified about changes to the selection of the current execution tree, path or step, respectively.

4.2.1 Step trace

As state above, the `ExecutionSimulator` uses the recorded history as a basis for its operations. We remember that the history only contains a multi-path execution trace. However, as we want to show only single-path traces to the user, SyxcDI has the task to transform multi-path into single-path execution traces. In general, this works as follows: An execution path in the form of a leaf branch is selected either by the user or initialized by SyxcDI. SyxcDI then filters the steps of the multi-path trace, so that the resulting trace contains only steps whose assigned branches are contained in the selected path. A branch is contained in a path, if it is the leaf branch representing the path or one of its antecedents.

Global branches within local branches introduce more complexity to the trans-

[7] http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachine.html

```
trait ExecutionSimulator {

  def results: Set[VerificationResult]
  def elementStatus(name: String) : VerificationStatus

  def currentElement: String
  def currentTrace: TraceStep
  def currentStep: TraceStep

  def treeEvent: Events[TreeChangedEvent]
  def pathEvent: Events[PathChangedEvent]
  def stepEvent: Events[StepChangedEvent]

  def canChangeElement(name: String) : Boolean
  def canSwitchBranches(b: TwinBranch) : Boolean
  def canChangePath(b: Branch) : Boolean
  def canChangeStep(s: TraceStep) : Boolean
  def canStepInto: Boolean
  def canStepOver: Boolean
  def canStepOut: Boolean
  def canStepBack: Boolean

  def changeElement(name: String)
  def switchBranches(b: TwinBranch)
  def changePath(b: Branch)
  def changeStep(s: TraceStep)
  def stepInto()
  def stepOut()
  def stepOver()
  def stepBack()
}
```

Listing 4.5: Public interface of ExecutionSimulator

formation of multi-path into single-path traces. Figure 4.3 shows an execution tree with such nested branches. It consists of two paths $p_1$and p_2. p_1 contains local branches, depicted by the blue edges, that are joined again (dotted lines). We see that the local branch at the right hand side contains a global branch. This means that within that local branch, there are again two paths. We can say that this local branch itself is the root branch of a *local branch tree*, short *local root*, with two *local paths* p_{1a} and p_{1b}. Thus, when it comes to constructing a single-path trace, it is not sufficient to choose between the (global) paths $p_1$and p_2, but in case $p_1$is chosen, the has to be made a choice between p_{1a} and p_{1b}, as well. In general, for every (local) root branch, a (local) path must be chosen for the construction of the single-path trace.

The criterion for a branch to be in the single-path trace is now extended as follows: If the branch is located within a local tree, it must be contained in the selected local path and the local root's containing branch must be contained in its tree's selected path. Note that this criterion is recursive, as there can also be local trees within other local trees, and so forth.

The transformation of multi-path into single-path traces is implemented within the class `TraceStep`, which is a wrapper for the `Step` instances that are provided by the recording facilities of Syxc. Listing 4.6 shows its public interface. It provides, among other things, basic operations for the traversal of the single-path trace. Internally, the single-path trace is constructed lazily on invocations of the method `children` (or other methods that depend on `children`). Listing 4.7 shows the implementation of `children` (as a `lazy val`) in the implementation class `AbstractTraceStep`. It uses the wrapped instance of `Step`, which is provided by Syxc, as well as an instance of `PathSelection`, which represents the current selection of the global and local paths. The implementation is very straight-forward: filter those steps out of the step's children, whose branch is contained in the current path selection and create new `TraceStep` instances using a factory. `PathSelection` (shown in listing 4.8) is basically a map of root branches (including local branches) to leaf branches (representing paths) with two operations: `contains(b: Branch)`, which was used above by the implementation of `children`, checks whether a given branch is part of the current path selection. Our previously defined recursive criterion is applied here. `getPath(r: RootBranch)` returns the currently selected path (in the form of a leaf branch) of a root branch. Note that the path selection is done lazily as well: As soon as the current path of a (local) execution tree is requested, that has not already been decided on, the method `initCurrentBranch` (not shown here) is used to automatically make a reasonable choice. The simple heuristics for this choice is: If there is a path containing an error, choose it. Otherwise choose the *true-most path*, that is the path that contains only *true*-branches.

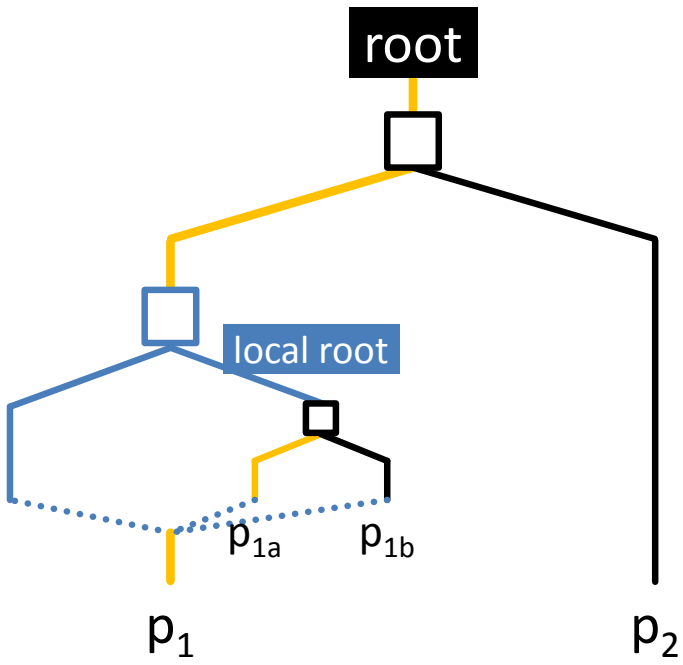


Figure 4.3: Path selection in case of nested branches

```
trait TraceStep {
  def hasParent: Boolean
  def parent: TraceStep
  def ancestors: List[TraceStep]

  def hasChildren: Boolean
  def children: List[TraceStep]

  def isLastSibling: Boolean
  def isFirstSibling: Boolean
  def previousSibling: TraceStep
  def nextSibling: TraceStep

  def preState: SymbolicState
  def postState: SymbolicState

  def status: VerificationStatus
}
```

Listing 4.6: Public interface of TraceStep

```
abstract class AbstractTraceStep
    (val step: Step, val path: PathSelection)
    extends TraceStep
{
  ...

  lazy val children : List[AbstractTraceStep] =
    step.children
        .filter(s => path.contains(s.branch))
        .map(s => TraceStepFactory(s, path))

  ...
}
```

Listing 4.7: Lazy construction of single-path trace

```
case class PathSelection(private var currentBranches: Map[
    RootBranch, Branch])
{
  def getPath(r: RootBranch): Branch = {
    if (!currentBranches.contains(r)) {
      currentBranches = currentBranches + (r -> initCurrentBranch(r
          ))
    }
    currentBranches(r)
  }

  def contains(b: Branch): Boolean = b.rootBranch match {
    case lb: LocalBranch =>
        contains(lb.parent) && getPath(lb).ancestorsAndSelf.
            contains(b)
    case r => getPath(r).ancestorsAndSelf.contains(b)
  }

  ...
}
```

Listing 4.8: Path selection class

4.2.2 State API

SyxcDI also offers an API for the symbolic state and, in particular, heaps. The goal of the API is to hide Syxc's internal representation of state, which is designed to meet the requirements of symbolic execution and verification, but not those of a presentation to the user. In particular, heap chunks are a concept that has this property.

In section 3.4.6, when we discussed visualizations of the heap, we already introduced concepts like heap objects, the heap graph and references. Also, we have seen that the textual presentation (the *variables view*) needs a slightly different representation than the presentation as a graph, namely a tree structure.

The design of the state API that SyxcDI offers to its clients provides an interface for both representations. Listing 4.9 shows a part of the interface, focusing on the type hierarchy. The tree representation uses the interface `HeapTreeNode`, that provides its children as a list of other `HeapTreeNodes`. The graph representation uses `HeapGraphNode`, which provides the list of edges to a given target object, and `HeapGraphEdge`, which provides its source and target node[8].

There are now different implementations of these interfaces. Some of these implement even both `HeapTreeNode` and `HeapGraphNode`. For example, a `LocalVariable` represents both a node in the heap tree and a node in the heap graph. Similarly, a `RefFieldMember` (an object field of a reference type) represents both a tree node and a graph edge.

The transformation of Syxc's internal state representation into our mixed graph/tree representation is done lazily whenever an intermediate state is requested by a client of SyxcDI.

4.3 Chalice language plugin for Eclipse

This plugin adds support for the Chalice language to the Eclipse platform. It is built with Xtext[9], a framework for the development of programming languages and domain specific languages. Xtext generates a full-fledged code editor for a language for which an LL(k) grammar is provided. The main features of our plugin are:

Syntax checking and colouring: The parser generated by Xtext is performing syntax checking an colouring simultaneously with the user's typing. Errors are reported directly within the code editor.

Type checking: The existing Chalice parser is used to perform type checks on the program. It is invoked as the user types and errors are reported immediately.

[8] The target node is never a `LocalVariable`, since it is not possible to have a reference from a heap object to a local variable. Therefore, the type of target is `HeapObject` and not `HeapGraphNode`.

[9] http://www.eclipse.org/Xtext

```
trait HeapTreeNode {
  def children: Iterable[HeapTreeNode]
  ...
}

trait HeapGraphNode {
  def edges(target: HeapObject): Iterable[HeapGraphEdge]
  ...
}

trait HeapGraphEdge {
  def source: HeapGraphNode
  def target: HeapObject
}


trait LocalVariable extends HeapGraphNode with HeapTreeNode
class LocalValueVar extends LocalVariable
class LocalRefVar extends LocalVariable

class HeapObject(members: Iterable[ObjectMember]) extends
    HeapGraphNode

trait ObjectMember extends HeapTreeNode
class ValueFieldMember extends ObjectMember
class RefFieldMember extends ObjectMember with HeapRef
class PredicateMember extends ObjectMember

class LocalVarRef extends HeapGraphEdge
trait HeapRef extends HeapGraphEdge
```

Listing 4.9: Symbolic heap abstraction

Cross-linking: Within a program, the user can navigate between different declarations by pressing Ctrl and clicking on a reference to a declaration. For example, given a method call, Ctrl+Click on the method name will jump to the called method's declaration. Also, by hovering over a reference, the relevant information is displayed in a tooltip. For example, the hovering over the use of a predicate displays the predicate's body in a tooltip.

Program outline: Xtext automatically generates a full-fledged outline view which is synchronized constantly with the active editor's code. The view is the direct equivalent of the concept described in section 3.4.1.

The Chalice language plugin is completely independent from Syxc or any other of our plugins. It only uses the Chalice parser for type checking.

In order to be able to display verification specific information in the outline view, an extension point for this plugin was developed. Other plugins can write extensions for this extension point and provide custom labels and icons for the elements of the outline view.

4.4 Syxc debugger plugin for Eclipse

The Syxc debugger plugin is the second UI component of our SyxcIDE architecture. While the Chalice language plugin only covers Chalice specific issues, the Syxc debugger plugin includes the possibility to symbolically execute and verify programs into the Eclipse platform. Its main features are:

Program verification: The plugin uses the Syxc Debug Interface described in section 4.2 to verify programs. The user can choose to verify a whole program using a button in Eclipse's toolbar, or to just verify a class or a class member using the corresponding context menu entry of the program outline view described in section 4.3. After the verification is completed, the verification status of verified elements is displayed in the outline view (using an extension to the extension point described above) and verification errors, if any, are displayed in the code editor using markers.

Simulation of the symbolic execution: Once a program was verified using one of the above options, the user can choose to simulate the execution of a verifiable member using a corresponding context menu item of the program outline. This activates the trace view and the different state views described below. The user can now use buttons in Eclipse's toolbar or the arrow keys to simulate stepping. In the code editor, the code corresponding to the current step is highlighted.

Trace view: The trace view is a simple implementation of the concept described in section 3.4.4. The current and, if any, the erroneous step are marked

appropriately. The user can choose a current step randomly by double-clicking it in the trace view. Branching steps are marked with a special icon. If the twin branch is available (i.e. it has been executed and it is not unreachable), the user can switch branches using the context menu item of the branching step.

Variables view: This view is the implementation of the concept described in section3.4.6.1. It uses the tree abstraction of the state provided by SyxcDI (see section 4.2.2). The underlying GUI element is a JFace multi-column tree viewer. JFace[10] viewers provide a level of abstraction from Eclipse's widget toolkit SWT[11] that allows to bind domain objects to the GUI elements and compute the structure of the content (that is the tree structure in this case) and its appearance (for example, text labels and icons) independently. For example, our view implementation binds an instance of a symbolic state to the JFace tree viewer. A so-called *content provider* determines the root-level nodes (i.e. the set of `LocalVariable` instances, see listing 4.9) and, lazily, the child nodes of each node (be it a `LocalVariable` or an `ObjectMember`). On the other hand, the *label provider*, the component responsible for the appearance of each node, computes the text label to be displayed for each column of the tree viewer.

Heap graph view: The concept described in section 3.4.6.2 is implemented using the Zest library[12] which is part of the Eclipse Graphical Editing Framework (GEF). Zest includes a JFace graph viewer, which provides similar abstractions as the tree viewer we have discussed above. Our view implementation binds an instance of a symbolic state to the graph viewer. Its content provider determines the set of graph nodes (instances of `HeapGraphNode`, which includes both local variables and heap objects) using the graph abstraction of the state provided by SyxcDI. Also, the connections between nodes are determined by our content provider implementation. Our graph viewer's label provider does not only produce text labels, but it draws custom figures for every node. These figures are created using the layout and rendering toolkit Draw2d[13], another component of GEF.

Path condition view: Our implementation of the concept described in section 3.4.6.5 is kept very simple in that it only displays the list of terms contained in a path condition using the string format provided by Syxc's term implementation. It is, however, possible to filter the terms according to some criteria, such as inequalities to *null* or equalities to *combine* terms.

Our state views do neither support state manipulation nor state diffing. However, all of them are able to switch between the pre- and the post-state of the current

[10] http://wiki.eclipse.org/JFace
[11] http://eclipse.org/swt/
[12] http://www.eclipse.org/gef/zest/
[13] http://www.eclipse.org/gef/draw2d/

step using corresponding buttons. The two heap visualization views also can be instructed to display the *old* heap.

4.5 Limitations

We identify a number of limitations of our implementation:

Language support: Although the recording facilities introduced to Syxc support the same language as Syxc does, we did not focus on a complete language support on the frontend. In particular, language features specific to concurrency are not covered, but may still be used in programs to debug with some caveats.

Syxc's external interface: At the moment, there is no possibility to interfere with the symbolic execution from outside of Syxc. Syxc is treated as a black box which takes a program and returns execution histories.

Performance: The memory usage of multiple components of our architecture is very high. In particular, Syxc's continuation-passing style causes the call stack be grow very large. This problem is even intensified by the recording. Also, Xtext requires a large amount of memory to build the syntax tree. To run the tool, it is highly advised to use a machine well-equipped with memory and to make sure, the Java virtual machine running the Eclipse instance is granted enough memory.

5 Conclusion

We have successfully developed a comprehensive conceptual design of a powerful debugging environment for symbolic execution.

Our work was guided by a small number of general principles which we defined in order to create a consistent, goal-oriented user experience. These principles address the question of how to support the user in gaining an understanding of such complex procedures as the symbolic execution of a program. We identified the structure of the data that is involved in symbolic execution and developed a general scheme of how to approach the task of debugging using this data. Finally, we sketched user interfaces to present the data and provide support reasonable user interaction.

The findings of this work have been incorporated directly into the design and implementation of a tool that provides a structured, visual and interactive presentation of the symbolic execution of Chalice programs.

For both our conceptual and implementational work we used the verifier Syxc. The thorough analysis of the inner workings of Syxc have also revealed possibilities for improvements. Our tool will not only be a help for users wanting to verify of programs, but it will also support the development of Syxc.

5.1 Related work

There are not many examples for debugging tools in the field of software verification. VeriFast [JSP10] is program verification tool for C and Java programs, using a symbolic execution approach similar to Syxc[1]. It comes with a simple graphical user interface which, similar to our tool, allows stepping through the execution of a program and inspecting the intermediate states.

The Boogie Verification Debugger (BVD) [LGLM11] is a small tool for programs verified with the verification engine Boogie [Lei08]. Boogie implements the verification condition generation approach, thus there are no intermediate states after every step that could be displayed to the user. However, BVD uses the counterexample model of the theorem solver, which contains a set of states that lead to a failure, and translates these states for the user to be understandable. A similar approach is used by [MR11].

[1] However, VeriFast bases on separation logic whereas Syxc/Chalice use the implicit dynamic frames approach.

5.2 Future work

We briefly list areas of future work:

- Syxc should be extended to support the interference into an execution from outside. This would allow additional features, like queries to the theorem prover.
- Several features described in the conceptional part of this report have not yet been implemented: for example, execution tree view, state manipulation, code bubbles
- Counter-examples from the theorem solver could be presented to the user for a better understanding of a failure.

Bibliography

[BRZ+10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 455–464, New York, NY, USA, 2010. ACM. (cited on pages 50 and 53)

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. (cited on page 11)

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *TACAS*, 4963. Lecture Notes in Computer Science:337–340, 2008. (cited on page 16)

[FW08] Daniel P. Friedman and Mitchell Wand. *Essentials of programming languages (3. ed.)*. MIT Press, 2008. (cited on page 22)

[GC10] Mike Gordon and Hélène Collavizza. Forward with hoare. In A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer London, 2010. (cited on page 11)

[HLMS11] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Fractional permissions without the fractions. In *Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs*, FTfJP '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM. (cited on page 16)

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. (cited on page 10)

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Programming Languages and Systems (APLAS 2010)*, pages 304–311. Springer-Verlag, November 2010. (cited on pages 9 and 83)

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. (cited on page 12)

[KMS12] IoannisT. Kassios, Peter Müller, and Malte Schwerhoff. Comparing verification condition generation with symbolic execution: An experience report. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 196–208. Springer Berlin Heidelberg, 2012. (cited on pages 12 and 27)

[Lei08] K. Rustan M. Leino. This is boogie 2, 2008. (cited on page 83)

[LGLM11] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The boogie verification debugger. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 407–414, Berlin, Heidelberg, 2011. Springer-Verlag. (cited on page 83)

[LM09] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag. (cited on pages 9 and 13)

[LMS10] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *Proceedings of the 19th European conference on Programming Languages and Systems*, ESOP'10, pages 407–426, Berlin, Heidelberg, 2010. Springer-Verlag. (cited on page 16)

[MR11] P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In M. Butler and W. Schulte, editors, *Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 73–87. Springer-Verlag, 2011. (cited on page 83)

[Ode11] Martin Odersky. The scala language specification, 2011. (cited on page 65)

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002. (cited on page 11)

[Sch11] Malte Schwerhoff. Symbolic execution for chalice. Technical report, 2011. (cited on pages 9, 16, and 19)

[SJP09a] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer Berlin / Heidelberg, 2009. (cited on pages 11 and 13)

[SJP09b] Jan Smans, Bart Jacobs, and Frank Piessens. Symbolic execution for implicit dynamic frames. Technical report, Tech. rep. Katholieke Universiteit Leuven, Belgium, 2009. (cited on pages 12, 13, and 16)

[Zel07] Andreas Zeller. *Beautiful Debugging*, chapter 28. O'Reilly Media, Inc., June 2007. (cited on page 16)