Recording Symbolic Execution

Bachelor Thesis Report

By: Andreas Buob

Supervised By: Malte Schwerhoff
Prof. Dr. Peter Müller

Date: September 4, 2015
## Contents

1 Introduction 2  
1.1 Silicon 2  
1.2 Motivation 2  

2 Recording Data Structure 3  
2.1 Requirements 3  
2.2 Defining the Data Structure 4  

3 Recording the Data 7  
3.1 The Continuation Problem 7  
3.2 Achieving Logical Nesting 8  
3.3 Example: Symbolic Execution with Recording 11  
3.4 Recording of Branching 15  

4 Implementation 17  
4.1 Actual Implementation 17  
4.2 Available Visualisations 17  
4.3 Unit Testing 20  

5 Conclusions 21  
5.1 Current Limitations 21  
5.2 Future Work 21  
5.3 Concluding Words 21
1 Introduction

1.1 Silicon

Silicon is a verifier for the intermediate language Silver, an intermediate verification language for permission-based, automated program verification [JKM+14]. Verification is done using symbolic execution, meaning that the program isn’t executed with actual input, but with symbolic values. A statement such as \( x := s; \ x := x * 2 \), where \( s \) is a local variable with symbolic value \( s' \), does not result in a state where \( x \) has a concrete value, but rather a state where \( x \) is constrained by \( x == s'*2 \). Furthermore, the symbolic execution can branch into different execution paths. Symbolically executing a statement such as \( s1; \text{if (b)} \text{then s2 else s3; s4}; \) will branch execution after the statement \( s1 \): Down one path, \( s2; s4 \) is executed under the assumption that \( b \) holds, down the other path, \( s3; s4 \) is executed under the assumption \( \neg b \). Paths can also be joined again, which leads to merged states.

Symbolic execution in Silicon is performed by four symbolic execution primitives: \texttt{execute}, which accepts statements and executes them, \texttt{evaluate}, which evaluates expressions recursively, \texttt{consume} and \texttt{produce}, which can add and remove access permissions and path constraints, and check assertions based on available permissions and constraints with help of the theorem prover \footnote{A detailed overview on Symbolic Execution primitives can be found in [Sch11]}.\footnote{A detailed overview on Symbolic Execution primitives can be found in [Sch11]}

1.2 Motivation

If a developer or user of Silicon is interested in how the symbolic execution is performed, for example to be able to retrace certain output, the current possibilities are rather limited, and consist mostly of the order in which the aforementioned symbolic execution primitives were called, with respective arguments.

Debugging symbolic execution has been inspected by previous work [Col12], and an eclipse plugin for debugging symbolic execution in Syxc, the predecessor of Silicon, has been created. Unfortunately, the plugin cannot be directly applied for Silicon, and the implementation does not allow an easy distinction between symbolic execution, recording the execution, and visualizing the recording, which is why debugging symbolic execution in Silicon is done in parts: First, finding a way to record the individual steps of a symbolic execution and second, visualise the recorded executions. This project has its focus on the recording part, visualisations will be provided, but for proof of concept only.
2 Recording Data Structure

2.1 Requirements

Various ways of visualising symbolic execution have been discussed by previous work [Col12]. Figure 2.1 shows, for example, how the recorded data could be visualised by text: The indentations indicate how operations are, due to their recursive dependency, part of each other. This representation could also be used in interactive visualisations: If a user is not interested in the details of certain operations (for example, the evaluation of the righthandside in an assignment operation), it could be collapsed away. A downside of this representation is branched execution: Different execution branches due to conditional statements are somewhat hard to present without losing the information of being in different paths with different path constraints. Possible solutions for this problem have been presented in [Col12].

```
method m(){
    var a: Int
    a := 1 + 2
    assert(a != 0)
}
```

Figure 2.1: A potential textual visualisation using indentation

Another possible visualisation is the use of a graph as shown in Figure 2.2. The graph primarily shows how the execution is branched due to conditional statements, the rest of the execution is represented linearly. The information on how the operations are nested is lost in this representation. The datastructure used for recording should provide enough information to allow both types of visualisation and potentially many others, see [Col12]. Since Silicon does not just branch symbolic execution but also joins branched paths if possible, the data structure should also be able to record such joins. Furthermore, it should be able to provide execution information such as the current state, and be easily extendible to record further operations, and the data that is operated on.
method m(b: Bool){
    var a: Int
    if(b){
        a := 1
    }
    else {
        a := 2
    }
    assert(a != 3)
}

Figure 2.2: A potential visualisation of a method with branching using a graph

2.2 Defining the Data Structure

The symbolic execution of a program is logged in a tree-structure, each node represents a record. A record is a logging structure that contains information about the type of operation it represents, the state in which the operation started and, if they exist, suboperations which are called by the operation.

There are different types of records, one for each symbolic execution primitive (execute, evaluate, produce and consume), and separate records for code structures of special interest such as if-then-else statements, conditional expressions and methods as a whole.

SymbolicRec v = ExecRec v Seq[SymbolicRec v]
    | EvalRec v Seq[SymbolicRec v]
    | ConsRec v Seq[SymbolicRec v]
    | ProdRec v Seq[SymbolicRec v]
    | GlobalBranchRec
        EvalRec v Seq[EvalRec v]
        Seq[SymbolicRec v]
        Seq[SymbolicRec v]
    | DiamondRec
        EvalRec v Seq[EvalRec v]
        EvalRec v Seq[EvalRec v]
        EvalRec v Seq[EvalRec v]
        Seq[SymbolicRec v]
    | MemberRec Seq[SymbolicRec v]

Figure 2.3: Definition of the records. v is the data that is stored in each record, e.g. current state, and the statement that is executed or the expression that is evaluated
The four record types for the four symbolic execution primitives `execute`, `evaluate`, `consume`, `produce` are rather straightforward: They contain information about the statement or expression they are about to process as well as the information that is needed to do so such as the current state, and a sequence of records of operations that are invoked by the operation.

The `MemberRec` is basically the 'root'-record for every method\(^2\) that is present in the currently verified Silver file.

The `GlobalBranchRec` records a branching point in the symbolic execution, e.g., a if-then-else-statement. It contains a condition and two outgoing branches. The condition and its evaluation is stored in the `EvalRec Seq[EvalRec]`, the two branches in the two `Seq[SymbolicRec]`.

The `DiamondRec` also records a branching point. The execution, however, branches only locally into two branches which are then joined again. This happens for example when conditional expressions such as `(b ? e1 : e2)` are evaluated. Branching is needed since the evaluation of `e1`, `e2` may depend on the assumed value of `b`. After the evaluation of the expressions in their respective branches, the execution is joined again, continuing with what is recorded in the `Seq[SymbolicRec]`.

\(^2\)Predicates and Functions also exist in Silver \(^{JKM14}\) and are recorded in a `MemberRecord` as well.
method m(b: Bool)
{
    var a: Int
    a := 1
    if(b) {
        a := 2
    }
    a := a + 1
}

Figure 2.4: A silver method and its symbolic execution record tree (informal representation)

Figure 2.5: Local Branching due to a conditional expression: (b ? e1 : e2)
3 Recording the Data

3.1 The Continuation Problem

Silicon makes heavy use of continuation-passing style. The four symbolic execution methods exec, eval, produce, consume take a function as argument, the continuation function. Informally, that function contains the ‘remainder’ of the execution. In Figure 3.1 this is demonstrated by the exec method. If the given statement that needs to be executed is a concatenation of two statements, exec calls itself with the first statement as parameter and a continuation function that executes another call of exec with the second statement as parameter and the initially given continuation function.

A method call is processed similarly: First, the parameters are evaluated, then, as part of the continuation of the evaluation, the precondition is consumed with the production of the postcondition in the continuation method.

Throughout this report, σ will denote states, and Q will denote the continuation function.

\[
\begin{align*}
\text{exec}(\sigma, s1; s2, Q) &= \text{exec}(\sigma, \text{methodCall}, Q) = \\
&= \text{eval}(\sigma, \text{parameters}, \lambda(\sigma_1) \cdot \text{consume}(\sigma_1, \text{precondition}, \lambda(\sigma_2) \cdot \text{produce}(\sigma_2, \text{postcondition}, Q)))
\end{align*}
\]

Figure 3.1: Examples of continuation-passing style usage in Silicon (simplified): Execution of concatenation of statements, and of method calls.

We distinguish between two sort of nesting; the operational nesting and the logical nesting. The operational nesting is given by the order of the symbolic execution method calls in the continuations; it directly depends on the way the code is executed. The logical nesting of the operations shows which operation is part of another, how operations consist of other operations.

```
method m(){
  var a: Int
  a := 1 + 2
  foo()
}
```

```
exec a := 1 + 2
|- eval 1 + 2
|- eval 1
|- eval 2
|- exec foo()
|- eval params
|- consume pre
|- produce post
```

```
exec a := 1 + 2
|- eval 1 + 2
|- eval 1
|- eval 2
exec foo()
|- eval params
|- consume pre
|- produce post
```

Figure 3.2: A method m and its operational (middle) and logical (right) nesting
Figure 3.2 shows an example of operational and logical nesting of the same method. To achieve operational nesting, one simply needs to know where the execution of symbolic execution primitives start and record them in their respective order.

Logical nesting, however, cannot directly be obtained from operational nesting; only knowing in which order the symbolic execution primitives are executed is not enough. If one would know the start and the end of the execution of primitives, one would still not have logical nesting: As shown in Figure 3.1, the execution of a concatenation of statements is defined through continuation passing, the execution of the second statement is defined in the continuation function of the execution of the first statement, and thus part of the execution of the first statement. The execution of the very first statement would thus start first and end last, since the execution of all further statements is defined in the continuation function, which is executed before the execution of the first statement ends.

### 3.2 Achieving Logical Nesting

The main concept is that it is necessary to know where the execution of a symbolic execution primitive starts and where the continuation of that primitive starts. To achieve this, we define wrapper-methods that wrap the symbolic execution methods as described in Figure 3.3. To mark the start of a symbolic execution primitive, we use the `insert` method, to mark the start of the continuation function, we use the `collapse` method.

![Figure 3.3](image.png)

Figure 3.3: The original `exec` method gets renamed to `exec'`, a wrapper method `exec` is defined which calls `exec'`. Analogue for the other three symbolic execution methods `eval`, `produce`, `consume`.

The two methods `insert` and `collapse`, as defined in Figure 3.3, define the recording data structure; the order in which they are called defines the nesting hierarchy. The recording data structure is built up in immutable logs, which contain the records.
1. log = (top, currentRecord, recordStack)

2. insert(record, log){
   log’ = log
   log’.currentRecord.children = record::log’.currentRecord.children
   return (log’.top, record, record::log’.recordStack)
}

3. collapse(log){
   return (log.top, log.recordStack.tail.head, log.recordStack.tail)
}

Figure 3.4: Definition of a log, and the two methods that operate on logs, insert and collapse.

A log is a tuple. The first element is the top element (‘root’) of the recording tree, which is usually, but not necessary, a MemberRec (see Figure 2.3). The second element is the currentRecord, insert adds new records as children of the currentRecord. The third element is the recordStack: a stack containing all records of primitives which have not called their continuation function yet.

Applications of insert and collapse are visualised in Figure 3.5. Given a record and a log, insert returns a new log, where the record has been added to the children of the old currentRecord, and the new currentRecord is the given record. record was also added to the recordStack. collapse returns a new log, where the first element in the recordStack has been removed, and the head element of the new recordStack is the new currentRecord.
Figure 3.5: Example on how insert and collapse create a tree structure. 'CR' is the currentRecord, the big arrow also points to the currentRecord.
3.3 Example: Symbolic Execution with Recording

In the following, two Silver statements are executed to demonstrate how the record tree develops during execution:

\[ a := 1 + 2 \]
\[ m() \]

This two statements are given to the \texttt{exec} method. Execution of a concatenation of statements has already been defined in Figure 3.1.

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation \texttt{Q}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{exec(σ, a:=1+2;m(), log_1, Q)} = \texttt{exec(σ, a:=1+2, log_1, \lambda(σ', log').exec(m(), Q(σ', log')))}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the left table column, the operations that were performed on the logs are noted, in the column in the middle, the code that gets executed, and the content of the current continuation function in the column to the right. There hasn’t been any operation on the logs so far, and the continuation function \texttt{Q} is assumed to be empty at start. In the next step, the continuation contains now the execution of the second statement, the method call of \texttt{m}. The first statement, the assignment, is executed; note that the code is now slightly reduced for better readability, states are omitted since they are always only threaded through, the records are implicitly given to \texttt{insert}, we write \texttt{insert(a:=1+2)} instead of \texttt{insert(ExecRec(a:=1+2))}:  

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation \texttt{Q}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{insert(a:=1+2, log_1)}</td>
<td>\texttt{exec(a:=1+2, log_1, Q) = let log_2 = insert(a:=1+2, log_1) in exec'(a:=1+2, log_2, \lambda(\text{log}') \cdot collapse() Q())}</td>
<td>\texttt{exec(m())}</td>
</tr>
</tbody>
</table>

With \texttt{insert(a:=1+2, log\_1)}, an operation on the logs has been performed. The continuation now contains a collapse, as it has been defined in the continuation of the \texttt{exec'} method. In the following steps, we will also ignore the \texttt{log} variables, since they are always just handed through anyways:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation \texttt{Q}:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{insert(a:=1+2)}</td>
<td>\texttt{exec'(a:=1+2, Q) = eval(1+2, Q)}</td>
<td>\texttt{collapse() exec(m())}</td>
</tr>
</tbody>
</table>
The execution of the assignment just invokes the evaluation of the righthand-side, no changes to the continuation or operations on the logs.

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(a:=1+2)</td>
<td>eval(1+2, Q) =</td>
<td>collapse()</td>
</tr>
<tr>
<td>insert(1+2)</td>
<td>insert(1+2)</td>
<td>exec(m())</td>
</tr>
<tr>
<td>eval'(1+2, λ) ·</td>
<td>collapse()</td>
<td></td>
</tr>
<tr>
<td>Q()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another insertion has been performed, the continuation now contains another collapse:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(a:=1+2)</td>
<td>eval(1+2, Q) =</td>
<td>collapse()</td>
</tr>
<tr>
<td>insert(1+2)</td>
<td>insert(1+2)</td>
<td>exec(m())</td>
</tr>
<tr>
<td>eval'(1+2)</td>
<td>collapse()</td>
<td></td>
</tr>
<tr>
<td>Q()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another operation on the logs has been performed with insert(1+2). The continuation grows with another collapse:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(a:=1+2)</td>
<td>eval'(1+2, Q) =</td>
<td>collapse()</td>
</tr>
<tr>
<td>insert(1+2)</td>
<td>eval(1, λ) ·</td>
<td>collapse()</td>
</tr>
<tr>
<td>eval(2, Q)</td>
<td>collapse()</td>
<td>exec(m())</td>
</tr>
</tbody>
</table>

The evaluation of the righthand-side of the assignment evaluates the lefthandside of the binary operation and adds the evaluation of the righthand-side to the continuation:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(a:=1+2)</td>
<td>eval(1, Q) =</td>
<td>eval(2)</td>
</tr>
<tr>
<td>insert(1+2)</td>
<td>insert(1)</td>
<td>collapse()</td>
</tr>
<tr>
<td>eval'(1, λ) ·</td>
<td>collapse()</td>
<td>exec(m())</td>
</tr>
<tr>
<td>Q()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The lefthandside of the addition is inserted. Since \( \text{eval}'(1) \) does not invoke any other method, the execution continues with its continuation:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation ( Q: )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{insert}(a:=1+2) )</td>
<td>collapse()</td>
<td>( \text{eval}(2) )</td>
</tr>
<tr>
<td>( \text{insert}(1+2) )</td>
<td>( Q() )</td>
<td>collapse()</td>
</tr>
<tr>
<td>( \text{insert}(1) )</td>
<td>collapse()</td>
<td>exec(( m() ))</td>
</tr>
<tr>
<td>( \text{collapse}() )</td>
<td>( \text{eval}'(2) )</td>
<td>collapse()</td>
</tr>
</tbody>
</table>

A \text{collapse} gets executed, then the continuation function is processed:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation ( Q: )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{insert}(a:=1+2) )</td>
<td>( \text{eval}(2, Q) = )</td>
<td>collapse()</td>
</tr>
<tr>
<td>( \text{insert}(1+2) )</td>
<td>( \text{insert}(2) )</td>
<td>collapse()</td>
</tr>
<tr>
<td>( \text{insert}(1) )</td>
<td>( \text{eval}'(2, \lambda() \cdot )</td>
<td>exec(( m() ))</td>
</tr>
<tr>
<td>( \text{collapse}() )</td>
<td>( \text{collapse}() )</td>
<td>( Q() )</td>
</tr>
<tr>
<td>( \text{insert}(2) )</td>
<td>collapse()</td>
<td>( \text{eval}'(2, \lambda() \cdot )</td>
</tr>
</tbody>
</table>

The righthandside of the addition is inserted. Analogue to the evaluation of the lefthandside, \( \text{eval}'(2) \) invokes no further methods, the continuation is executed:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation ( Q: )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{insert}(a:=1+2) )</td>
<td>collapse()</td>
<td>collapse()</td>
</tr>
<tr>
<td>( \text{insert}(1+2) )</td>
<td>( Q() )</td>
<td>collapse()</td>
</tr>
<tr>
<td>( \text{insert}(1) )</td>
<td>collapse()</td>
<td>exec(( m() ))</td>
</tr>
<tr>
<td>( \text{collapse}() )</td>
<td>collapse()</td>
<td>( \text{eval}(2) )</td>
</tr>
<tr>
<td>( \text{insert}(2) )</td>
<td>collapse()</td>
<td>collapse()</td>
</tr>
</tbody>
</table>
Another collapse is performed, the continuation is now processed:

<table>
<thead>
<tr>
<th>Operations on logs:</th>
<th>Executed Code:</th>
<th>Continuation Q:</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(a:=1+2)</td>
<td>collapse()</td>
<td></td>
</tr>
<tr>
<td>insert(1+2)</td>
<td>collapse()</td>
<td>exec(m(), Q())</td>
</tr>
<tr>
<td>insert(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>collapse()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>collapse()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>collapse()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>collapse()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two more collapses are performed, the remaining code to be executed is `exec(m(), Q)`, where the continuation function `Q` is currently ‘empty’. The remaining execution is omitted; the method call would be executed in a similar way to the assignment: First, the parameters of the method call (here: none) would be evaluated, then the precondition would be consumed, and finally the postcondition produced.

<table>
<thead>
<tr>
<th>Operations on logs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \log_2 = \text{insert}(a:=1+2, \log_1) )</td>
</tr>
<tr>
<td>( \log_3 = \text{insert}(1+2, \log_2) )</td>
</tr>
<tr>
<td>( \log_4 = \text{insert}(1, \log_3) )</td>
</tr>
<tr>
<td>( \log_5 = \text{collapse}(\log_4) )</td>
</tr>
<tr>
<td>( \log_6 = \text{insert}(2, \log_5) )</td>
</tr>
<tr>
<td>( \log_7 = \text{collapse}(\log_6) )</td>
</tr>
<tr>
<td>( \log_8 = \text{collapse}(\log_7) )</td>
</tr>
<tr>
<td>( \log_{res} = \text{collapse}(\log_8) )</td>
</tr>
</tbody>
</table>

Figure 3.6: Resulting operations on the logs when executing `exec(a:=1+2)` and the representation of the constructed records, stored in the last log, \( \log_{res} \)
3.4 Recording of Branching

As defined in Figure 2.3, there are special records for branched execution. Since `insert` takes only one `record` as argument, `GlobalBranchRec` and `DiamondRec` need special methods to fill the outgoing branches: `finishGlobalBranchRec` and `finishDiamondRec`, as defined in Figure 3.7.

1. `finishGlobalBranchRec(log_cond, log_thn, log_els, log){
   log' = insert(GlobalBranchRec(log_cond.top, log_thn.top, log_els.top), log)
   return log'
}

2. `finishDiamondRec(log_cond, log_thn, log_els, log_join, log){
   log' = insert(DiamondRec(log_cond.top, log_thn.top, log_els.top, log_join.top), log)
   return log'
}

Figure 3.7: Definition of the two methods `finishGlobalBranchRec` and `finishDiamondRec`.

A definition of a evaluation of a conditional expression is given in Figure 3.8. First, the expression `e0` that serves as condition is evaluated, then, the symbolic execution is branched with Silicon’s `branchAndJoin` method, which takes three continuation methods as argument. The first two contain the execution in the two different branches, the third contains the execution after the branches are joined again (see Figure 2.5 for visualisation).

For recording, the evaluation of the condition and the evaluations in the branches receive empty logs. The resulting logs of those evaluations contain only recording about the respective evaluations and nothing else. These logs are given to `finishDiamondRec`, which returns a log that attached the records of the aforementioned logs to a `DiamondRec`. The definition for other branching constructs such as if-then-else-blocks is analogous, using `finishGlobalBranchRec` instead.
eval(σ, (e0 ? e1 : e2), Q) =
  eval(σ, e0, λ(σ′)) ·
  branchAndJoin(σ′, e0, λ(σ_{e=0}) · eval(σ_{e=0}, e1, Q_{Nop}),
  λ(σ_{e=0}) · eval(σ_{e=0}, e1, Q_{Nop}),
  λ(σ_{join}) · Q(σ_{join})))

⇓

eval(σ, (e0 ? e1 : e2), log, Q) =
  eval(σ, e0, log_{∅}, λ(σ′, log_{cond}) ·
  branchAnd Join(σ′, e0, λ(σ_{e=0}, log_{∅}) · eval(σ_{e=0}, e1, log_{∅}, λ(σ_{e=1}, log_{e=1}) ·
  Q_{Nop}),
  λ(σ_{e=0}, log_{∅}) · eval(σ_{e=0}, e1, log_{∅}, λ(σ_{e=2}, log_{e=2}) ·
  Q_{Nop}),
  λ(σ_{join}, log_{join}) ·
  let log_{res} = finishDiamondRec(log_{cond}, log_{e=1}, log_{e=2}, log_{join}, log) in
  Q(σ_{join}, log_{res})))

Figure 3.8: Definition of Evaluation of conditional expressions, with and without recording. log_{∅} is an empty log, Q_{Nop} a continuation function that does nothing.
4 Implementation

4.1 Actual Implementation

The actual implementation differs in a few points from the description above, for various reasons. One of the biggest differences is that \texttt{insert} and \texttt{collapse} do not pass immutable log variables around, but rather work on a globally defined (mutable) logging object.

Since we use one mutable objects instead of several immutable ones, the \texttt{finish}-methods defined in Figure 3.7 had to be implemented differently. Instead of passing resulting logs from their respective branches to a \texttt{finish} method which merges them together, \texttt{finish}-methods were implemented as 'markers', that tell the mutable logging object when the condition is evaluated, or when a certain branch has finished its execution.

Also, branches might not even be executed since the condition might not hold. The current solution is that there is a \texttt{Unreachable}-node at the start of the branches of branching records by default; if the branch is executed, the \texttt{Unreachable}-node is overwritten.

4.2 Available Visualisations

Three types of visualisations have been implemented, mainly as proof of concept, to show that the recorder does what it’s supposed to do:

\textbf{SimpleTree}: Renders the record structure to a string output. Similar to the example shown in Figure 2.1, SimpleTree makes use of indentation to show the record hierarchy.

\textbf{Graph}: The record structure gets rendered to DOT, which can then be interpreted by GraphViz \cite{gra}. A demonstration can be found in Figure 4.2.

\textbf{HTML}: The record structure is rendered to a JavaScript-datatype, which is then used by a HTML-template. Is presented similar to SimpleTree, but allows single nodes to be 'collapsed', meaning that their children get hidden. Also displays the states.

For implementation of further visualisations, there are guidelines and tutorials on how to use (and extend, if necessary) the recording structure.
method m(b1: Bool, b2: Bool)
{
    var a: Int
    a := 1
    if(b1){
        a := (b2 ? ((a==1) ? 2 : 3) : ((a!=1) ? 4 : 5))
    }
    else{
        if(b2){
            a:= ((b1 && b2) ? (b1 ? 1 : 2) : (b2 ? 3 : 4))
        }
        else{
            // Do nothing: Unreachable branch by default.
        }
    }
}
Figure 4.2: Example output of the Graph visualisation, using GraphViz. Code of method m can be found in Figure 4.1.
4.3 Unit Testing

There are unit tests available. The main focus of the unit tests is to make it possible to detect changes on how symbolic execution is recorded; e.g., if there is a change to Silicon that should not have impact on recording but actually has. If there are changes made to the recording itself, one gets a direct comparison between the new recording and the old one.

For unit testing, the record structure gets rendered to a string representation that only contains information about the types of records in use and the hierarchy among them. The actual output is then compared to an already available expected output, differences are reported.
5 Conclusions

5.1 Current Limitations

Currently, if Silicon aborts symbolic execution (e.g., due to a failed assertion), the output of recording is not guaranteed to be sound. In ‘most cases’, recording should provide information as expected, or in a way that it ‘makes sense’, but there is no guarantee for that. For example, if an assertion fails, Silicon might retry the assertion after rearranging states - if this happens, the execution of the assertion gets recorded twice, but there is currently no information in the records on why this happens.

5.2 Future Work

As mentioned above, the behaviour of the recording in case of an abortion of Silicon is not well-defined. Recording could provide more information on why Silicon aborts, it could for example show input for which an assertion fails.

More type of records could be made available to give even more fine granular information on how symbolic execution is performed. Tutorials on how to use and extend records are available in the source code.

It is currently not possible to disable recording completely. There is a flag which enables/disables writing to files for visualisation purposes, in order to avoid I/O-overhead when verifying several Silver files at once, but the recording data structure is still built.

5.3 Concluding Words

In this project, a way of recording symbolic execution in Silicon has been implemented, and prototype visualisations for proof of concept have been provided. To avoid unwanted changes on how symbolic execution gets recorded, developers of Silicon will be notified when such changes occur due to available unit tests. The recording object in Silicon should be clearly distinguishable from the original source code, and documentation on how to use, extend and modify the recording object were provided.
References


