Incremental Symbolic Execution
- Bachelor Thesis -

Roger Koradi
Supervisors: Malte Schwerhoff, Valentin Wüstholz, Peter Müller
ETH Zürich, Switzerland

August 20, 2015
Abstract

Incrementally changing programs should be incrementally verifiable, each verification iteration re-using verification results of an earlier iteration for parts that are unchanged. The document describes how a caching technique developed for verification-condition-generation (VCG) based Dafny verifier can be adopted into a symbolic-execution (SE) based environment, in both a coarse- (method-level) and fine-grained (statement-level) way and discusses some issues with an initial implementation for one such verifier, Silicon.

Introduction

We observe that many programs under verification change incrementally - i.e. they gradually evolve to implement a desired functionality. Therefore, most parts of such programs are unaffected by some code changes. It would be wasteful, for example, to re-verify the entire program just because the programmer appended one more statement. Typically, such changes happen in an IDE, and indeed, many IDEs already provide support for incremental versions of other components, such as incremental type checkers, incremental compilers, etc.

The Silicon verifier - as part of the Viper [1] [2] infrastructure - is eventually supposed to be constantly running in the background during IDE sessions, much like the Dafny [4] verifier for its Boogie backend, providing periodic feedback to the programmers as they are editing the code. Therefore, response time is crucial to IDE integration. And as this is directly affected by the amount of code that needs to be re-verified during a verifier run, caching and re-using results for parts that have not changed is a crucial optimisation.

Silicon is operating on the Silver language.
To implement caching, we must assign cache lookup keys to AST nodes whose verification results we want to cache individually, namely methods, functions, predicates and statements. We thus require a way to attach information to AST nodes. To do so, we introduce attributes in [subsection 1.1]

One special attribute, the Verified-If attribute outlined in [subsection 1.2] is meant to simplify proofs we cannot completely skip by replacing a potentially expensive assertion with a simpler one that implies the more expensive.

The Assertable statement is meant to work in conjunction with the Verified-If attribute by assigning a boolean value to the conjectured outcome of an

[http://www.pm.inf.ethz.ch/research/viper]
assertion with side-effects. We explain the idea behind it in subsection 1.3.

We will then focus on our approaches for caching that largely adopt the ideas from [1], both method- and statement-level (sections 2 and 3 respectively), including our use of checksums (2.4 and 3.3) to track changes and the transformations (3.6) applied to a changed program in order to simplify its re-verification.

We will then briefly discuss shortcomings of our initial implementation (5) and possible approaches to overcoming them (6).

1 Language Extensions

1.1 Attributes

To implement our caching, we require a way to attach information to an AST node. Attributes are information-wrappers. They can be appended to any statement or contract and prepended to any method, function or predicate, using the syntax

```plaintext
@key(values)
```

Where `key` is an alphanumeric identifier specifying the purpose of the information attached and `values` is a comma-separated list of either string- or expression-values.

If multiple attribute annotations with the same key are attached to the same AST node, their values are concatenated.

We decided against supporting attributes on expressions. Doing so would introduce the need for extensive use of parentheses to make sure the attributes are attached to the right expression or statement, which is error prone and may result in confusing code.

E.g. consider a statement

```plaintext
var a:Bool := foo(1) && i < 3 @myAttribute()
```

In our implementation, `myAttribute` is attached to the assign statement. If we allowed statement on expressions, the attribute would instead be attached to the expression `3` and to attach it to the assignment, we would need to wrap the whole line into parentheses:

```plaintext
(var a:Bool := foo(1) && i < 3)@myAttribute()
```
Note that writing

\[
\text{var a:Bool := (foo(1) \&\& i < 3)@myAttribute()}
\]

would have attached the attribute to the right hand side of the assignment, instead.

Now consider the situation where we wanted to attach an attribute to each sub-expression. To give such a statement a readable form, we would need to write it like this:

\[
\text{(var a:Bool := (}
\text{foo(}
\text{1 @myArgAttribute()}
\text{)@myMcAttribute()}
\text{\&\&}
\text{(}
\text{i@myLitAttribute()}
\text{<}
\text{3@myLitAttribute()}
\text{)@myCompAttribute()}
\text{))@myConjAttribute()}
\text{))@myAttribute()}
\]

Which makes both reading and writing the statement much more arduous (note that all parentheses are mandatory), and obscures the meaning of the statement, even more so when multiple attributes are attached to a sub-expression, the attribute keys do not imply the intended location and the attributes have values or worse that if they are expressions, may themselves have attributes attached to them.

### 1.2 The Verified-If Attribute

One special attribute we defined is the verified-if attribute.

\[
@\text{verified-if(exp)}
\]

Where \textit{exp} is a pure expression (i.e. one that does not involve permissions) evaluating to a boolean value.

It is meant to be used as a simplification of assertions. During inspection of a program, we may know a statement is verified if a certain condition holds, yet may be unable to check that condition until a certain time in our execution. By attaching a verified-if attribute to a statement - in particular, to assertions - we let the verifier know that if \textit{exp} holds, anything that would otherwise require proof during the execution of that statement can
be blindly assumed.

Note that in our own implementation, we cannot “blindly assume” the existence of a certain amount of permissions. This is due to the fact that permissions are modelled as so-called “chunks” on a “heap” and “asserting” permissions (“exhale”) effectively consumes the chunks whereas “assuming” permissions (“inhale”) produces them. Thus, “blindly assuming” permissions in Silicon would falsify the verification result.

I.e. if “blindly assuming” permissions did not check and modify the heap, any permissions that would have been consumed had the statement been executed normally, would remain on the heap and could (erroneously) be consumed a second time.

As a result, our implementation only assumes the pure parts of assertions during the execution of a statement marked with a verified-if attribute whose \( exp \) holds.

### 1.3 The Assertable Stmt

Silver’s \textit{exhale/inhale} statements are impure alternatives to the pure \textit{assert/assume} statements known in first-order verification languages.\cite{silver2002} We say an assertion is \textit{pure}, if it does not modify the heap - i.e. does not contain permissions.

In Silicon,

\[
\text{inhale } A
\]

assumes all pure assertions in \( A \) and produces appropriate heap chunks for the permissions specified in \( A \) and

\[
\text{exhale } A
\]

acts as the inverse of \textit{inhale} in that it \textit{asserts} all pure assertions in \( A \) and consumes the chunks corresponding to the permissions specified in \( A \) from the heap.

In our caching implementation, we require a way to check if an \( \text{exp} \) that held at a particular place in the cached version still holds in the version under re-verification. Because simply exhaling the \( \text{exp} \) would cause a loss of permissions, we introduce the assertable stmt.

\[
\text{var } b : \text{Bool} \\
\text{b := assertable(} \text{exp} \text{)}
\]

This assigns to \( b \) a boolean expression that evaluates to true if and only if \( \text{exp} \) could be exhaled. This allows us - in particular - the use of \( b \) as the
condition of a verified-if attribute.
In our own implementation, we settled for a downgraded version of the
statement that assigns either “true” if the expression could be exhaled at the
point of time the assertable statement was executed and unknown otherwise.
More information about this is given in [5.1]

2 Coarse-Grained Caching

We will refer to a method/function/predicate as a member, if the distinction
between the three cases does not matter.

A statement or expression invokes a member \( m \) during its execution if

\[
m \text{ is a function that is applied}
\]

or \( m \) is a predicate that is folded/unfolded

or \( m \) is a method that is called

2.1 Approach

Coarse grained caching is caching at the member level. If the member is
in the cache and “has not changed”, we use the stored information to com-
pletely skip verification of the member. If the member is not in cache or “has
changed”, we throw away the cached information - if any - and completely
re-verify the member.

We use a pair of checksums to keep track of changes to a member. We
call these the contractEntity-checksum and the bodyEntity-checksum and
collectively refer to them as the entity checksums of a member.
A change of either checksum implies that the member itself has changed to
the point where a re-verification is necessary. A change in the contractEn-
tity checksum marks a change in the member’s contracts and a change in
the bodyEntity checksum marks a change in the member’s body. We call
the combination (see subsection 2.4) of the two entity checksums the entity
checksum of the member’s.

A change to a member’s (transitive) dependency (2.2) (contract or body)
also leads to the re-verification of the member.
To keep track of changes to dependencies, we compute contractDependency-
and bodyDependency- checksums, the combination of which we dub the
dependency checksum of the member’s.
Thus, a member is said to have changed if since its last verification, either
its entity or dependency (or both) have changed.
Our cache is a map from members to some cache entry. We thus further assign a unique identifier (UID) to each member in order to act as cache lookup key.

In our implementation, the uid is string-valued and manually attached to a member via a string-attribute. The entity checksums are computed from values of string-attributes manually attached to the member. Because they are manually attached, nothing stops the user from assigning the same “initial string” to each entity checksum. To make the entity checksums unique for each method, we prefix the user-specified string with the uid and a string constant “contract” or “body”, respectively, before we hash the resulting string.

2.2 Member Dependencies

Local Dependencies

We say a member \( a \) locally depends on member \( b \) if \( b \) is invoked in either the contracts or the body of \( a \)'s. If the invocation of a local dependency happens in \( a \)'s contracts, we say \( b \) is a “contract dependency” of \( a \). Otherwise, we call \( b \) a “body dependency”.

The differentiation between contract and body dependencies is necessary, as we will see once we have introduced transitive dependencies.

Transitive Dependencies

We say a member \( a \) transitively depends on \( b \) if

\[ b \text{ is a local dependency of } a \]

or \( a \) locally depends on a member \( m \) and \( b \) is a transitive dependency of \( m \)'s.

If a method \( m \) is a local dependency of member \( a \), we do not care about \( m \)'s local body dependencies when computing \( a \)'s transitive dependencies because from \( a \)'s point of view, we can reason about \( m \)'s impact on the state solely in terms of its pre- and postconditions (which we cannot do for functions or predicates).

As with local dependencies, we differentiate between transitive contract and body dependencies for that very reason.

2.3 Requirements

All members must have a UID and entity checksums attached to them.
Because the UID is used as a key for cache lookup, it must never change so long as the cache is not deleted. Changing the UID risks looking up false results.

If a change occurs in the contract or body of a member, the corresponding entity checksum must change. If either entity checksum changes when no change had occurred, the member and all members transitively depending on it are re-verified. If an entity checksum does not change, even though it should have because a change in the member’s contract/body occurred, the caching infrastructure wrongly assumes there have been no changes to the member and skips verification, which may mask verification errors the changes newly introduced or report an error the change fixed.

### 2.4 Checksum Computation

For our computation, we require two binary checksum operations:

- a commutative “addition”
- a non-commutative “combination”

For two checksums $a, b$:

let $a + b$ denote their addition

and $\text{combine}(a, b)$ their combination.

Both operations aim to merge two checksums into one such that a change to either will change the result. We want to use them to construct a single checksum from a set of checksums that will change if any one of the checksums used in the construction changes.

The *addition* of checksums reflects the case where the order in which the checksums are merged does not matter. E.g. for the computation of a body dependency checksum, it should not matter in what order the dependencies occur in the body.

The *combination*, on the other hand, is used where the order does matter. E.g. it does matter in what order statements are executed, which is why we will make use of the combination in computing statement checksums (3.3) for our fine-grained caching outlined in section 3.

Note that in our implementation, checksums are byte arrays. Adding two checksums equals xor-ing them, combining them means hashing their concatenation.

I.e:

$$a + b := a \ xor \ b$$

$$\text{combine}(a, b) := \text{hash}(a ++ b)$$
This is sufficient because \( a \) and \( b \) are supposed to be different from each other.

By traversing the ASTs formed by the members’ contracts/bodies, we find all local contract dependencies and all local body dependencies for each member.

We can then compute the transitive closure of these sets to get for each member a set \( tcd \) of transitive contract dependencies and a set \( tbd \) of transitive body dependencies. (Note that \( tcd/tbd \) do not contain the member they belong to because we already track changes to these by means of their entity checksums.)

Let \( m \) be a member. We define the following abbreviations:

\[
\begin{align*}
    m.cE &= m’s \text{ contractEntity checksum} \\
    m.bE &= m’s \text{ bodyEntity checksum} \\
    m.entity &= \text{combine}(m.cE, m.bE) = m’s \text{ entity checksum} \\
    m.cD &= m’s \text{ contractDependency checksum} \\
    m.bD &= m’s \text{ bodyDependency checksum} \\
    m.dependency &= \text{combine}(m.cD, m.bD) = m’s \text{ dependency checksum} \\
    m \neq \text{method} &= \text{“m is not a method”}
\end{align*}
\]

We compute:

\[
\begin{align*}
    m.cD &= m.cE + \text{depSum}(tcd) \\
    m.bD &= m.bE + \text{depSum}(tbd)
\end{align*}
\]

where

\[
\text{depSum}(D) := \sum_{d \in D} d.cE + \sum_{\{d \in D | d \neq \text{method}\}} d.bE
\]

### 2.5 The Cache

The cache is a map

\[
uid : \text{String} \mapsto (\text{ent} : \text{Checksum}, \text{dep} : \text{Checksum}, \text{CacheEntry})
\]

The cached entity and dependency checksums \( \text{ent} \) and \( \text{dep} \) are compared to the current in order to assign a priority \([1]\) to the result of the lookup and return a \( \text{CacheResult} \) to the verifier.

A \( \text{CacheResult} \) is a tuple \((\text{Priority}, \text{CacheEntry})\), where \( \text{CacheEntry} \) is either:

- a wrapper for a member to be re-verified in case of a cache miss or if \( \text{dep} \neq \text{dependency} \)
- the \( \text{CacheEntry} \) from the cache that contains the necessary information to skip re-verification, otherwise
In the coarse-grained caching, the member returned in case of a cache miss or a dependency mismatch is quite simply the member passed to the caching interface for lookup. (In the fine-grained caching outlined in section 3, we may transform the member before we return it to the verifier.)

Before verification, the verifier performs a lookup for each member and sorts the received CacheResults by priority before re-verifying them in order. If possible, it uses the CacheEntry to skip verification and report the cached results, otherwise it re-verifies the member it received.

Note that in our implementation, we only sort members other than functions because functions are axiomatised in Silicon and that axiomatisation requires two things:

one: the axiomatisation of callees
two: snapshots of the heap recorded during execution of the function

Therefore, functions are already sorted in an order consistent with the static callgraph of the program and our implementation maintains that order. We do sort the non-function members, though. An alternative implementation could additionally re-order independent functions according to their priorities.

Also, the CacheEntry for a function must in our case also provide the snapshots recorded.

3 Fine-Grained Caching

3.1 Approach

Fine-grained caching improves on the coarse-grained caching in that it also caches each method’s statements and remolds a method to skip re-verification of parts that have not changed.

To that end, we assign a statement checksum to each statement in a method and store the necessary information to skip the statement in a statement cache map that we include in the cache entry stored after verification of a method. (Note that a failing statement must invalidate its cache entry, see 5.2.)

Upon lookup, we can then use the retrieved statement cache to replace stmts that do not produce a cache miss - i.e. that are verified - with a state. When we encounter a statement that was not in the statement cache, we use the state information provided by previous statements to inject a statement that
restores the verifier state to what it used to be just before the statement producing the cache miss would have been executed in the cached version of the program and pass the modified method to the verifier for re-verification. There are several statement that we handle specially:
The if-then-else (ite) statement because the branch itself only directs the flow of the execution, which is why we can content ourselves with caching only the stmts in the then- and else-branches.
(Note that our evaluation [4] proved this assumption to be overly optimistic in our initial implementation, at least. We will discuss the issue and a possible solution in 6.2.)
The while-statement because the statements after the while will reason about the while solely in terms of its condition and invariants.
The constraining statement because it has a body.
And the method call statement because we would want to exploit the fact that weaker preconditions and stronger postconditions are “acceptable” changes - that is, they do not actually influence the next statement.
We will cover these cases in more detail in subsection 3.6

3.2 Requirements
All the requirements from subsection 2.3 apply.

In our implementation and due to the way we handle branching stmts there, we must also be able to invalidate a stmt cache entry if a stmt fails (5.2).

3.3 Checksum Computation
The checksum computation operates on the CFG - as opposed to the AST and assigns statement checksums to each statement. Working on the CFG has the benefit that the statements after an ite-statement are visited once for each branch and thereby replicated, effectively separating the two branches.
This yields the benefit that should a statement only fail in one branch but not in the other, re-verification needs not re-consider the succeeding branch.

E.g, consider the program in Figure 1, where stmt3 is guaranteed to succeed if executed after stmt1 and guaranteed to fail if executed after stmt2.
Our checksum computation separates the branches (Figure 2), s.th. if we were to change stmt2, the formerly succeeding branch would not be affected.
In particular, stmt3 and stmt3’ are assigned different statement checksums.

With exception of the special cases mentioned below, a statement’s checksum is computed by combining its predecessor’s checksum with the hash of the statement’s pretty printed representation and then adding the dependency checksums of all members invoked by the statement. (Note that only method
call statements will directly invoke a method. Thus, as method calls are handled specially, we will never actually add the dependency checksum of a method to a statement checksum of ordinary statements.)

We define the following abbreviations:
- \( c_{Pre} \) = statement checksum of preceding statement
- \( \text{hash}(x) \) = hash of \( x \)'s pretty printed representation
- \( x.D \) = members invoked in \( x \)
- \( \text{hashSum}(D) = \sum_{d \in D} d.\text{dependency} \)
- \( x.\text{cond} \) = condition of \( x \) (if any)
- \( x.\text{invs} \) = invariants of \( x \) (if any)

When we say a statement \emph{receives} a checksum, we mean that the receiving statement assumes that checksum to be its \( c_{Pre} \).

The very first statement in a method receives a combination of the hash of the method’s precondition’s pretty-printed representation, to which the dependency checksums of all members invoked in the precondition are added.

**If-then-else**

An ite statement in its translated form is basically just a collection of two conditional edges in the CFG. However, we do compute a condition checksum \( c_{Cond} \), which is a combination of the preceding stmt’s checksum and the hash of the branch condition’s pretty printed representation to which we add the dependencies of all
members invoked in the condition. (In light of the issues revealed by our evaluation, we may assign $cCond$ to the branch as a statement checksum. We will further discuss this in §6.2.)

\[
cCond := \text{combine}(cPre, \text{hash}(\text{ite.cond})) + \text{hashSum}(\text{ite.cond}.D)
\]

To make sure both branches get assigned different checksums, we continue in the then-branch with

\[
cThn := \text{combine}(\text{hash(“then”)}, cCond)
\]

Whereas in the else-branch, we continue with

\[
cEls := \text{combine}(\text{hash(“else”)}, cCond)
\]

The “next statement” after the ite stmt (e.g. if the branch occurred within the body of a constraining stmt) would then receive

\[
cAfter := \text{combine}(tAfter, eAfter)
\]

Where $tAfter/eAfter$ are the checksums the last statement in the then/else-branch would pass to its successor.

**While**

The statements of a while body are not cached - and hence not assigned a statement checksum - because the statements following a while-loop will reason about the loop’s impact on the state solely in terms of its condition and invariants. We compute

\[
\text{loopEntity} := \text{hash}(\text{while.cond}) + \sum_{i \in \text{while.invs}} \text{hash}(i)
\]

\[
\text{stmtChecksum} :=
\begin{align*}
\text{combine}(cPre, \text{loopEntity}) + \\
\text{hashSum}(\text{while.cond}.D \cup \bigcup_{i \in \text{while.invs}} i.D)
\end{align*}
\]

Other than the $\text{stmtChecksum}$, we also attach a $\text{bodyChecksum}$ to the while statement that is the checksum the while’s body would pass to its successor when given $\text{stmtChecksum}$ as the preceding checksum, to which we added the contractDependency checksums of all methods invoked in the body of the while.

We will later use the bodyChecksum to determine if a while-loop that was in cache can be skipped entirely or whether its body must first be re-verified to still maintain the loop’s invariants. In either case, the next statement is not affected because it only receives $\text{stmtChecksum}$.

A future version might introduce nested statement caches and the caching of the while loop’s statements. However, this incurs a potentially expensive
check in our transformation, which we will further discuss in §3.6.

**Constraining**

The constraining statement is very similar to the `ite` stmt, different in that we actually assign statement checksums to the statements in the constraining’s body and the statement following the constraining receives the checksum the last statement in its body would pass to its successor.

As with the while statement, we assign a body checksum to the constraining statement, which is the checksum passed to the next stmt but to which the `contractDependency` checksums of all methods invoked in the constraining’s body are added.

**Method Call**

A method call’s statement checksum is the combination of `cPre` and the pretty printed representation of the method call, to which we add the dependency checksums of all members invoked in one of the call’s arguments.

Note that we do not add the callee’s dependency checksum to the call’s statement checksum. Instead, we attach the callee’s `contractDependency` checksum with a different attribute.

This, because we do not want the statements following the method call to miss in cache if the callee’s contracts have changed but the changes are limited to weaker pre- or stronger postconditions. We will discuss this idea in more detail in subsection 3.6.

**3.4 The Cache**

We now additionally cache a statement cache. Therefore, our new cache is a map

\[ \text{uid} : \text{String} \mapsto (\text{ent} : \text{Checksum}, \text{dep} : \text{Checksum}, \text{CacheEntry}, \text{stmtCache}) \]

In our implementation, a cache lookup remains unchanged from the verifier’s perspective. The verifier still queries with a member and receives a `CacheResult` that holds either a `CacheEntry` to skip verification of the member with or a `CacheEntry` wrapping a member to be verified.

The only difference is that now, if the verifier receives a `CacheResult` with a member, it may not be the member originally passed to the cache infrastructure for lookup, as a method that did not produce a cache miss but whose
cached dependency does not match its current will now trigger a transformation (see subsection 3.6).

The caching infrastructure also keeps track of the $uid \mapsto stmtCache$ mapping for later insertion of reverified stmt’s cache entries and for later including the stmtCache in the member’s cache entry.

The statement cache is a map

$$stmtChecksum \mapsto (VerifierState, StmtInfo)$$

Where the VerifierState is an object that should wrap enough information for the corresponding verifier to restore its state to what it used to be after the execution of the statement the statement cache entry belongs to. Our own implementation’s “verifier state” is further explained in subsections 3.5 and 5.3. We will make use of the verifier state object cached in our transformation as it will allow us to inject RestoreState statements (subsection 3.5) at appropriate places.

The StmtInfo holds any additional information the transformation may need if the statement does not produce a statement cache miss. I.e:

- nothing
  - if the statement is not a method call, a while loop or a constraining statement
- the body dependency checksum
  - if the statement is a while loop or a constraining statement
- the callee’s contractDependency checksum, its precondition and its postcondition
  - if the statement is a method call

The verifier creates appropriate verifier state and statement info objects after each statement that was executed successfully (i.e. without raising a verification error) and passes it to the caching infrastructure to store it in the stmtCache that are processed one after the another.

A verifier is also expected to invalidate failing statements’ entries.

### 3.5 The RestoreState Statement

The RestoreState statement - as its name implies - holds a cached verifier state into which the verifier should change before executing the next statement. In our implementation, neither VerifierState nor RestoreState statement can be parsed and both are instead produced programmatically.
Note that in our own implementation of Silicon’s RestoreState statement (outlined in subsection 5.3), the “verifier state” is actually a sequence of restorable Silicon states.

3.6 The Transformation

A member that did not produce a cache miss but whose cached dependency checksum does not match has been verified before and is transformed before re-verification.

The transformation aims to create a version of the member that produces the same results as if the member was completely re-verified but that cuts down on the amount of work that needs to be done during re-verification by removing verification steps that would only check properties already known to hold.

To that end, we use the statement cache entries (3.4) created during the initial verification. The verifier states they contain - in conjunction with the RestoreState statement (3.5) - allow us to replace a statement with the state established after its execution, thereby completely bypassing actual execution of the statement. We say the statement was compressed to a state.

When we compress successive statements, we need only restore the most recently recovered verifier state.

Furthermore, we can mark statements to be “verified under a certain set of assumptions”. That is, we can use the verified-if attribute (1.2) to skip or simplify proofs for statements that we need to re-execute because the condition under which they are verified can only be checked at runtime. Finally, there may be statements that have changed or that are new. For obvious reasons, we cannot do anything to these statements, but we must ascertain that they are verified under the same conditions they would have been verified had the untransformed member been passed to the verifier.

Our transformation does nothing to functions or predicates.

Our transformation of methods is outlined below.

The transformation keeps track of:

- a set $B$ of new so-called “assumption variables” of boolean values (initially empty)

- the last statement’s verifier state, $stateLast$ (initially none)

Ordinary Statements

We say a statement is ordinary, if it is neither an ite statement, nor a while loop, nor a constraining statement nor a method call.
If an ordinary statement is in cache, statement cache lookup will yield a verifier state.

If the statement is in the statement cache and B is empty, we compress it by removing the statement from the method’s body and updating stateLast to be the newly retrieved state, then continue with the next statement.

If the statement is in the statement cache and B is not empty, we attach a verified-if attribute with the conjunction of all assumption variables in B as its value to the statement. We will refer to this as vi-marking the statement.

If the statement is not in the statement cache and there is a stateLast available, we inject a RestoreState that takes stateLast before the current statement and stop the transformation for the current branch.

If the statement is not in the statement cache and no stateLast is available, we simply stop the transformation for the current branch.

**If-then-else**

If all statements before an if-branch can be compressed, the transformation injects a RestoreState statement before the branch.

Either way, “stateLast” is set to none and the transformation continues in each of the branches in turn.

**Constraining**

If a constraining statement is not in the statement cache, it will be treated the same way as if it were an ordinary statement.

If a constraining statement is in the statement cache, lookup will yield a tuple

\[(\text{cachedVerifierState}, \text{cachedBodyChecksum})\]

If the cachedBodyChecksum matches the current bodyChecksum attached to the constraining statement by means of an attribute, we check if all statements in the body are in the statement cache. This check is necessary, because some statements may not have changed but produced an error that invalidated their entries.

If the check succeeds, and B is empty, we update lastState to be the cachedVerifierState and remove the constraining statement, continuing with the
next.

If the check succeeds and $B$ is not empty, we reset \textit{lastState} to none and vi-mark as many statements in the body as possible (and continue vi-marking with the next statement if we can mark them all).

If the \textit{cachedBodyChecksum} does not match the current or if the check fails, we transform as much of the body as is possible, inject a \texttt{RestoreState} statement, before the constraining statement if a \textit{lastState} is available, then either stop the transformation for the current branch if a statement in the body was not in cache or continue vi-marking at the next statement if the last statement in the body would continue vi-marking at its successor.

\textbf{While}

If a while loop is not in the statement cache, it will be treated the same way as if it were an ordinary statement. (Note that because we are not caching the statements in a loop’s body, a statement failing in the loop’s body will invalidate\cite{5.2} the statement cache entry of the loop.)

If a while loop is in the statement cache, lookup will yield a tuple $(\text{cachedVerifierState}, \text{cachedBodyChecksum})$

If the \textit{cachedBodyChecksum} matches the current \textit{bodyChecksum} attached to the while loop by means of an attribute and $B$ is empty, we update \textit{lastState} to be the \textit{cachedVerifierState}, remove the loop from the method’s body and continue transformation with the next statement. Note that if we were to later introduce nested statement caches and caching of statements in while loop bodies, we would - as with the constraining - additionally have to check that all statements in the body are actually in the statement cache, i.e. none are invalidated. We briefly discuss nested statement caches in \cite{7}

If the body checksums match but $B$ is not empty, we vi-mark the while loop and all statements in its body, then continue vi-marking with the next statement.

If the body checksums do not match and $B$ is empty, we inject a \texttt{RestoreState} statement before the loop if a \textit{lastState} is available and mark the loop with an attribute telling the verifier that it only needs to check whether the body still maintains the loop invariants. However, we still set \textit{lastState} to the \textit{cachedVerifierState} and continue transformation with the next statement. We can do this because the next statement only reasons about the while in terms of its condition and invariants.
3  FINE-GRAINED CACHING

Note that if we were to later introduce nested statement caches and caching of statements in while loop bodies, we could additionally transform the body.

If the body checksums do not match and $B$ is not empty, we continue vimarking at the next statement.

**Method Call**

Our special handling of method calls is motivated by the fact that we can reason about a method call’s impact on the verifier state solely in terms of the callee’s pre- and postconditions. E.g. we can use assertable statements \(^{1.3}\) and verified-if attributes\(^ {1.2}\) to simplify proofs when the preconditions have not changed and the change to the postcondition does not matter for the correctness of the statements following the call. If the preconditions have not changed and we could still exhale the old postcondition, then not only do the pure parts of the old postcondition still hold, we also have at least as many permissions after the new version of the call as we had in the old version.

Note that this assumes that our program does not feature assertions that would check for the existence of at most a certain amount of permissions. Otherwise, the assertable statement could lead to such assertions’ being erroneously assumed.

If a method call is not in the statement cache, it is treated as if it were an ordinary statement.

If a method call is in the statement cache, lookup will yield a tuple

$$(\text{cachedVerifierState}, \text{cachedCalleeChecksum}, \text{oldPre}, \text{oldPost})$$

Where $\text{oldPre}$ and $\text{oldPost}$ are the callee’s old pre- and postconditions, respectively and $\text{cachedCalleeChecksum}$ is the callee’s old contract dependency checksum.

If the callee’s current contract dependency checksum matches the cached and $B$ is empty, we set $\text{lastState}$ to the $\text{cachedVerifierState}$ and remove the method call before continuing the transformation with the next statement.

If the callee’s current contract dependency checksum matches the cached and $B$ is not empty, we set $\text{lastState}$ to none, vi-mark the call and continue vi-marking at the next statement.

If the callee’s current contract dependency checksum does not match the current, we require in our implementation that its preconditions have not changed. If they have changed, we treat the call as if it missed in cache. We
briefly discuss alternatives to this in §6.1.

We thus inject several statements:

First, we inject a RestoreState statement before the method call if a last-State is available.

Then for each argument $arg_i$, we create an assignment

$$\text{var } a_i : \text{arg}_i = arg_i$$

where $a_i$ are new local variables.

Finally after the method call, we inject

$$b_k := \text{assertable(oldPost') }$$

we call $b_k$ an assumption variable and add it to the set $B$.

$oldPost'$ is a transformed version of $oldPost$, where we replaced each occurrence of a formal argument with the corresponding $a_i$ and every occurrence of a formal return with the corresponding target variable. E.g. a call

$$i := \text{foo}(i)$$

where $i$ is of integer value and $\text{foo}(i: \text{Int})$ returns an integer $r$ and has post-condition “$r = i + 1$” will be replaced by

$$\text{val } a_0 : \text{Int} := i$$
$$i := \text{foo}(i)$$
$$\text{var } b_k : \text{Bool}$$
$$b_k := \text{assertable}(i == a_0 + 1)$$

We then continue transformation at the next statement.

## 4 Evaluation

For our evaluation, we chose the longest-running testcase we had at hand and the longest-running method it featured and observed the times they took to verify as we changed the environment in which they were verified. Figure 3 summarises our findings.

There are no surprises in (a)-(c). If everything is cached (b), then re-verification takes virtually no time. And if we only need to re-verify the single method we changed, then of course we
Fig. 3:
blue: time to verify the entire testcase
red: time to verify the test method
(a) if nothing is cached
(b) if everything is cached
(c) if everything is cached and we add a statement at the very end of the
method’s body and re-verify the testcase using coarse-grained caching only
(d) if everything is cached and we add a statement at the very end of the
method’s body and re-verify the testcase using fine-grained caching

do save the time needed to verify the other methods. In our coarse-grained
 caching, that method is not changed prior to being re-verified, which is why
the time taken to verify it in (c) is virtually the same as in (a).

(d), though, did not meet our expectations.
Inspection of the transformed method shows that our handling of ite state-
ments during transformation is punished by an increase in the time taken
to re-verify the method. Because we compress the statements before the
branch and the statements after the branch separately, we may have mul-
tiple RestoreState statements for an evaluation path, some of which are
obsolete. When we manually remove the obsolete RestoreState statements,
re-verification time in fine-grained caching drops even below (c), as we would
want.
We refer to 5.3 and 6.2 for a more indepth discussion of the issue and a
possible solution.

On a side note, we also tried to evaluate the merit of marking an assertion
with a verified-if attribute, but failed to construct a test case with an assertion that would take the prover long enough to prove or disprove for us to draw meaningful conclusions.

5 Implementation

5.1 Assertable Statement

Our implementation provides an “assertable at this point in time”. It will try to exhale the expression passed to it and assign “true” to the target variable, if it succeeds and unknown otherwise.

To see the difference to the ideal semantics, consider a program

\[
b := \text{assertable}(x==0) \\
\text{assume}(x==0) \\
\text{assert}(b)
\]

Ideally, we would want the assertion to hold. However, in Silicon, the assertion will fail because at the time the assertable was executed, no information about \(x\) was available.

The reason we chose to do that is that Silicon’s not merging branches makes implementing the semantics more difficult. Consider an assertable statement

\[
b := \text{assertable}(a \Rightarrow \text{acc}(x.f))
\]

Assuming no further knowledge of \(a\), this will branch into a state where \(a\) is true and where we try to exhale \(\text{acc}(x.f)\) and a state where \(a\) is not true and we do not try to exhale anything. Each branch now runs its course independently of each other. To implement “\(b\) evaluates to true if and only if the expression can be exhaled“, we would need to artificially join the branches after the evaluation of the assertable. Our initial implementation does not do this.

5.2 Stmt Cache Entry Invalidation

In Silicon, one complication are statements that branch - i.e. that may be executed in more than one way - because such branches are not merged before continuing with the next statement but instead executed and verified individually.

\[
\text{e.g.} \\
\text{inhale}(b \Rightarrow \text{acc}(x.f))
\]
Our implementation supports this by storing a sequence of state informations in a statement’s cache entry. Instead of restoring a single verifier state per RestoreState statement (5.3) and then continuing from there, we restore and continue for each stored state in turn.

Therefore, a failing statement must invalidate any entry it may have cached prior to its failure. This because by the time a statement fails, it may already have been cached with some verifier states where it did not. If we did not invalidate the entry, these states would be treated as if they were the only ones that could be reached, which upon re-verification skips the branch that produced the error and all branches that would have followed it, had it succeeded.

5.3 RestoreState Statement

A restorable Silicon state consists of

- a store,
  a mapping from variables to terms representing their values

- two heaps g.h,
  holding heap chunks with terms describing permissions and heap values

- a set of terms A
  of prover assumptions

- a set of terms bc
  of branch conditions

The RestoreState statement takes as an argument a set of restorable Silicon states and for each state restores the verifier’s state and continues execution of the rest of the program following the RestoreState statement.

In any term, a variable $v$ may be represented as a subterm

$v@i$

Where $v$ is the name of the variable and $i$ is an integer counter saying that this variable was the $i^{th}$ to be created during the current verification run. Because this number will change depending on what and in what order is verified, the RestoreState statement cannot safely correlate the terms in the current state with the terms in the state to be restored. Therefore, the execution of a RestoreState statement will re-write all terms in the state to be restored with fresh prover variables. It then tells the prover to assume all
terms it so created from the set $A$, sets the store, heaps and branch conditions and continues execution.

A possible future optimisation could be the computation of the terms common to all states to be restored and rewriting these once in advance, then for each state only additionally rewriting the terms exclusive to it.

Note, though, that even if we applied the solution to our issue of obsoletely cascading RestoreState statements [6.2], we may still end up with multiple RestoreState statements in an execution path due to our handling of while loops. If a while loop is in cache and only its body changed, we must inject a RestoreState statement before the loop in order to check that the new body still maintains the invariants, but because the statements following the loop do only rely on its condition and invariants, we can continue compression after the while loop.

6 Implementation Shortcomings

6.1 Method Calls and Preconditions

Our implementation only supports stronger postconditions but not weaker preconditions. We briefly mention two ideas on how to support changing preconditions and why neither really convinces us.

Extended Assertable

This idea is based on our assertable statement [1.3]. We implement a second version of the statement that takes two expressions as opposed to one:

$$b := \text{assertable}(\text{exp1}, \text{exp2})$$

Which assigns to $b$ a pure boolean expression evaluating to true if and only if $\text{exp2}$ can be exhaled after inhaling $\text{exp2}$ in a fresh verifier state (i.e. without any prior assumptions).

The idea is that if we pass the current precondition as $\text{exp1}$ and the old precondition as $\text{exp2}$, then $b$ would evaluate to true if the new precondition is not stronger than the old one.

When using this method, the assertable for the postcondition would remain unchanged from our current implementation.
Downside of this method is that it fails to capture changes in the precondition that are compensated by corresponding changes in the postcondition.

E.g. assume there is a method

```java
method foo(x:Ref)
    requires acc(x.f,write)
    ensures acc(x.f,write)
```

A call to this method will exhale the write access permissions to `x.f` in the precondition and re-inhale the write access permissions in the postcondition. If we changed these contracts in a later version of the program to

```java
method foo(x:Ref)
    requires acc(x.f,read)
    ensures acc(x.f,read)
```

Then for the handling of the precondition and starting from a fresh state, we would inhale `acc(x.f,read)` and try to exhale the old precondition, `acc(x.f,write)`, in the resulting state, which would fail. However, this change in contracts doesn’t matter to the statements following the call because the overall amount of permission to `x.f` remains unchanged.

**Permission Counting**

This idea tries to overcome the limitation of the extended assertable one by instead requiring that the total amount of permissions before and after the method call remains unchanged.

We would remove impure expressions in the pre and postconditions and only check whether the pure parts from the current precondition imply the pure parts of the cached one and the pure parts of the cached postcondition imply the pure parts of the current. We then count the total amount of permissions for each available heap location and compare the amount of permissions available (i.e. not exhaled) after the call to the permissions that were available after the cached version of the call.

Downside of this method is that we would need to correlate cached permission terms (see 5.3) with current ones, which may be a risky thing to do unless we can somehow establish a condition that will allow us to check if the variables in the terms that we consider to be correlated really are meaning the same variable, which is further complicated if we take into account that the “current” state variables may in fact be a product of an earlier RestoreState statement that might have rewritten terms from a different version of the program. Other issues includes renaming or splitting
permissions across several new chunks.

6.2 Eliminating Intermediate RestoreState Statements

Consider a program whose CFG looks like in Figure 4 and assume all statements are in cache. Our current transformation (3.6) will result in the program in Figure 5 where the RestoreState statements in the marked boxes are actually obsolete as they are only needed for the branching. However, due to the fresh prover variables they create and the additional assumptions they pass to the prover, they significantly increase the effort the prover has to proof an assertion. Also, RestoreState statements following another will be executed once for every branch created by the preceding RestoreState statement.

A naive implementation might check whether the first statement in each of the branches is in the statement cache and desist from creating a RestoreState statement if that is the case. Instead, we create a new local boolean variable to replace the condition of the branch with (initialized to “unknown” so that neither of the branch is considered unreachable by the verifier).

In the program from Figure 4, this helps and the new transformation creates the program in Figure 6. However, a change to stmt3b in Figure 4 will result in the program depicted in Figure 7 where the marked RestoreState is still obsolete for one of its branches.
In order to push the RestoreState down into the appropriate branch, we must actually cache the branch, evaluate the condition to a term, then create cache (two) states, one where the condition holds and one where it does not and pass the corresponding state as lastState to the appropriate branch during our transformation if we can compress all statements before the branch. This should allow the transformation to create the program in Figure 8.

One potential issue could be the invalidation of the if-branch’s entry. If a statement before an if-branch fails, then that statement’s entry is invalidated and transformation will stop there.
If a statement after a branch fails, then the if-branch is not affected in our case because Silicon does not join branches. Therefore, it is sufficient to invalidate an if-branch’s statement cache entry if the evaluation of the condition throws an error - e.g. because it dereferences a possibly null variable.
7 Conclusion & Future Work

In this document, we explored the idea of an incremental verifier in symbolic execution based environments. We introduced attributes as information wrappers that can be attached to members or statements, the verified-if attribute as a tool for simplifying proofs and the assertable statement as a way to assign a boolean expression to the projected outcome of an assertion with side-effects. We then used these extensions and detailed the ideas behind our coarse- and fine-grained caching approaches and discussed some issues with our initial implementation, as well as potential solutions. In particular, we discussed the fact that our evaluation has shown our initial implementation’s handling of ite-statements to result in an increase of the time taken to re-verify a program due to obsolete RestoreState statements.

Future implementations might also want nested statement caches and caching of loop body statements. However, as this would require us to inject even more (and possibly cascaded) RestoreState statements into the program and given the impact they have on the amount of work required by the prover, the merit of such nested statement caches remains unclear. Perhaps the optimisation mentioned in 5.3 or a different optimisation could alleviate some of the strain put on the prover.

Future implementations will also want to compute the entity checksums from code rather than from user-annotated strings. Some care should be taken to make these checksums insensitive to purely textual changes, such as the addition or removal of comments. Future implementations may also choose to replace the hash of the statements’ pretty-printed representations in the computation of statement checksums with a computation insensitive to certain re-orderings of expressions or renaming of variables.

Future implementations will want the members’ UIDs to be computed and attached to members in a way transparent to the programmers.

Finally, future implementations might want a way to detect weaker pre-conditions.

Acknowledgements

I should like to extend my special thanks to Malte Schwerhoff and Valentin Wüstholz for their invaluable help in their position as both, supervisors and facilitators.

I should further like to thank Prof. Peter Müller for enabling the thesis.
References


Appendices

A Declaration of Originality
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Incremental Symbolic Execution

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):
Koradi

First name(s):
Roger

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Basadingen, 20. August 2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.