Visual Debugging for Symbolic Execution

Master's Thesis

16 September 2018

Alessio Aurecchia
`aalessio@student.ethz.ch`

supervised by
Arshavir Ter-Gabrielyan

Chair of Programming Methodology
Department of Computer Science
ETH Zürich

Abstract

In recent years, tools for program verification have made significant progress and are becoming more widely used. Yet, they still lack effective facilities to allow investigating and understanding verification errors, especially when the input program makes use of more advanced language features, such as quantified permissions in Viper.

We think the most effective way to help a user in identifying the source of a verification error is by employing a visual debugging approach, therefore we want to provide a technique to automatically produce small, visual counterexamples based on the information provided by a symbolic execution engine.

We conducted a feasibility study to understand whether we could effectively generate counterexamples via bounded modeling with Alloy, a language and analyser for software modeling. The main idea behind this methodology is that Alloy, because it performs a bounded search, is not affected by the problems involving quantifier instantiation, and is therefore able to provide complete concrete models in situations where the SMT solver would not.

In our technique, we encode the information about a symbolic execution state into a model and use Alloy to generate instances of it. These instances can then be used to build a visual diagram of the program's state. When the state being modeled is one where a verification error occurred, and we additionally encode the last failed query performed to the SMT solver, then the instances Alloy generates are counterexamples to the failed verification.

As part of our feasibility study, we have implemented a subset of the technique described in this thesis into a tool, integrated with the Viper IDE. This proof of concept for a debugger demonstrates that our approach for visualizing counterexamples to verification failures in the context of symbolic execution is a feasible one and is worth exploring in more detail.

Contents

1 Introduction

In recent years, tools for program verification have made significant progress and are becoming more widely used. The verification environments built around them provide features that help the user in writing valid specifications easily and, more importantly, they provide a way to work with the verifier in an interactive fashion. Yet, these tools still lack effective facilities to allow investigating and understanding verification errors, especially in the context of more advanced language features, such as quantified permissions[1], in the case of the Viper language[2].

1.1 Motivation

Determining the source of a verification error can still be tedious. The errors are found by the SMT solver[3], that “sits” at the lowest layer of most automated verification toolchains, and they are specified with respect to the SMT encoding of the program being verified. This makes it generally difficult to relate these problems back to the higher level of abstraction that the user works at. In addition to that, due to the inherent incompleteness of SMT solving, verification may fail for reasons other than the program being incorrect (e.g. for formulas involving quantifier or non-linear integer arithmetic), and the solvers may not be able to come up with models for more complex problems.

The IDE built around the Viper framework [2] has been largely expanded as part of Ruben Kälin’s thesis [4] and allows fast feedback on the state and result of the proof when working with the verifier. Moreover, the project provides an early prototype for a Symbolic Execution debugger, laying down a solid foundation to build a more advanced debugger on. This debugger is limited in that it does not support visualizing structures defined via quantified permissions and its abilities of providing counterexamples to a verification error are lacking, mainly because of only visualizing symbolic states and having to rely on the SMT solver to provide the model for building a counterexample.

In the presence of advanced language features, this approach is not powerful enough to provide the user with effective visual representations of the problem, therefore we want to explore alternative ways of building counterexamples with the information provided by the symbolic execution engine, and without necessarily having to rely on models from the SMT solver.

1.2 Previous Work

The original debugger offered support for navigating through the various states of the symbolic execution performed by Silicon, as well as inspecting their internal information. In addition to that, its graphical representation of the symbolic state supported visualizing local variables, functions, predicates and objects on the heap. These features were intended to provide a visual counterexample to a verification error, in order to help the user identify where the problem with the specification might be. This approach is limited by the fact that all the information about local variables or heap chunks is symbolic. For this reason, there is a limited amount of facts that can be learned directly about the values of local variables or whether, for example, references are aliases. The debugger tries to discover some of this information from explicit facts in the path conditions of the state being inspected.

The *VeriFast Program Verifier* [5] is a tool for the verification of C and Java programs annotated with preconditions and postconditions written in separation logic, and is implemented via symbolic execution, which make it, at least in principle, very similar to Silicon. The VeriFast IDE also provides debugging features similar to those provided by the original Viper Debugger: it allows navigating through the states explored by symbolic execution and displays the store, the heap, and the path conditions for each one of them, but does not provide a visual representation of this information.

The *Symbolic Execution Debugger (SED)* [6] is a platform for symbolic execution that allows debugging programs by exploring their symbolic execution tree interactively. In addition to that, SED also allows verifying programs (or parts of them) when JML[7] specifications are provided. The tool is implemented on top of the Eclipse IDE and uses KeY's Symbolic Execution Engine [8]. The system allows visualizing information about the current symbolic state being executed, such as the symbolic stack and path conditions. In case of potential aliasing between local variables, SED allows displaying graphs of the memory layout and provides a slider to inspect all the possible aliasing configurations.

Alloy [9, 10] is a language and an analyser for software modeling. It allows describing sets of structures via the use of constraints and then finding instances of these models, or counterexamples to assertions about facts in the model. The search space for these instances is limited to a *scope* defined by the user. The tool displays these structures graphically as an interactive graph. It is possible to "go to the next instance" so that all instances in the scope can be inspected manually. Moreover, Alloy performs symmetry-breaking to avoid showing multiple instances of the model that would essentially be equivalent. We are going to investigate whether Alloy can be used as the main tool to solve our problem of counterexample generation.

1.3 Main Objective

The overall objective we would like to achieve with this project is to find an effective way of visually representing dynamic structures defined via the use of quantified permissions. We will conduct a feasibility study in order to understand whether a bounded modeling approach can be employed to build these visualizations and to identify counterexamples when a verification error occurs.

1.3.1 Updating the existing infrastructure

The current version of the Viper IDE only provides features for writing Viper code, not for debugging it. The code in other parts of the framework has evolved and the compatibility with the debugging features of the IDE has been broken.

The first task of this project is to understand the requirements for the design of an infrastructure that would allow us to add debugging features to the IDE and then to actually put it in place. The architecture should allow extensibility and should not force other components of the system to depend on debugging features they do not need. The logging infrastructure of Silicon will also need to be updated in order to provide the debugger with all the information needed to visualise the verification states.

1.3.2 Visualization of Quantified Permissions

In order to achieve our main objective, we will devise a technique that will allow us to encode the information about a symbolic execution state into an Alloy model. The model will then be used to generate instances of the symbolic state

Our technique for modeling symbolic states will be integrated into a broader pipeline, that will allow solving some limitations of working with Alloy alone. In this thesis, though, we will only focus on the part of the pipeline that deals with modeling the program's state.

One important distinction between the approach of the previous debugger and the approach we take with the new one is that, in this new project, we focus on providing concrete counterexamples, whereas the visualization provided by the original debugger only showed symbolic states, where the value of variables were not known. This means that we also always display visualizations where the aliasing relation between reference is known.

Finally, we will conduct an evaluation to understand whether our technique is practically effective in visualizing counterexamples and whether it has fundamental limitations, both in terms of the approach and of its implementation.

1.4 The Structure of this Report

In this section, we give a short overview of the contents and the structure of this report.

Chapter 1 introduced the main motivation behind the project, it briefly described related work that has been done in the field, and it outlined the goals we would like to ultimately achieve.

In Chapter 2, we explain our approach to solving the problem of counterexample generation. We describe both the more general pipeline we envisioned and the subset that we focused on as part of this thesis.

Chapter 3 presents the translation of Silicon terms that appear in the symbolic execution trace into Alloy, and describes the technique we use to encode the symbolic state into a model that can be used to generate counterexamples.

In Chapter 4 we discuss the previous architecture of the system and how it was updated to enable integrating the new debugger extension. In chapter 5 we briefly describe the features offered by the debugger.

In Chapter 6 we evaluate the models produced by the debugger by considering some concrete use cases and we discuss its limitations in terms of the technique and its implementation. Then, we compare our approach with that of other program verifiers based on symbolic execution that offer debugging features.

Finally, in Chapter 7 we draw conclusions about the project and briefly list some potential interesting topics for future work.

2 Approach

In this chapter we discuss the approach we have decided to take in order to tackle the problem of counterexample visualization for programs that use quantified permissions. First, we discuss the problem of verification failures when quantifiers are involved. Then, we describe our general approach to solving this problem (including ideas that we have not implemented in the current system). Finally, we go into more details on the parts of the pipeline that this thesis focuses on.

2.1 Verification Failures

When a failure occurs in the context of quantified permissions, it is likely that the SMT solver was also not able to produce a definite answer to the query that it was presented, e.g. because of a quantifier-related incompleteness. In these situations we usually don't get candidate models for the failure from the solver, and even if we did they would only be partial models.

During symbolic execution, the verifier performs many different queries to the SMT solver, trying to prove that in the current symbolic state the next operation is safe and satisfies the specification. In general, the verifier is trying to prove that the following formula holds.

$$PC \models PO \quad \equiv \quad \neg PC \vee PO$$

Where PC are the facts known about the current state (path conditions), and PO is the proof objective, corresponding to the next step we want to verify in the program. In order to prove this fact, the verifier would query the SMT solver with the negation of this formula, hoping to find that it is unsatisfiable.

$$Q := \neg(\neg PC \vee PO) \quad \equiv \quad PC \wedge \neg PO$$

If $sat(Q)$ is indeed false, then the original formula holds and symbolic execution proceeds to the next step. In the other cases we have a problem. If $sat(Q)$ is true, then we definitely have a verification failure, but we also have a model for the failure from the solver, which can be used to build a counterexample. On the other hand, if the solver is not able to prove or disprove $sat(Q)$ (a common outcome in verification failures with quantifiers), we still have to consider this a failure, but we get no model (or only a partial one). In this third case, we need to find an alternative way for building a counterexample.

2.2 Bounded Modeling

If a concrete counterexample to the program's specification does exist, then it has to satisfy the formula $PC \land \neg PO$. The intuition behind our approach is that, though the SMT solver may not have been able to deal with the problem because of unbounded quantifiers, we could still try performing the search in a *bounded scope* and we might find an instance satisfying the formula nonetheless. This is based on the idea, referred to as the *Small Scope Hypothesis*, that if an assertion is invalid, it is likely to have a small counterexample, so by exploring all the small cases, we are likely to find one [10].

We decided to use *Alloy*[9, 10] as a tool to perform our bounded search. Alloy is a language and analyser for software modeling, it allows describing models via the use of constraints and then finding instances (or counterexamples) of them. The search space for these instances is limited to a *scope* defined by the user. The tool allows "going to the next instance" so that all instances of the model whithin the current bounds can be inspected. In addition to that, Alloy performs symmetry-breaking by default, so that equivalent instances are shown only once.

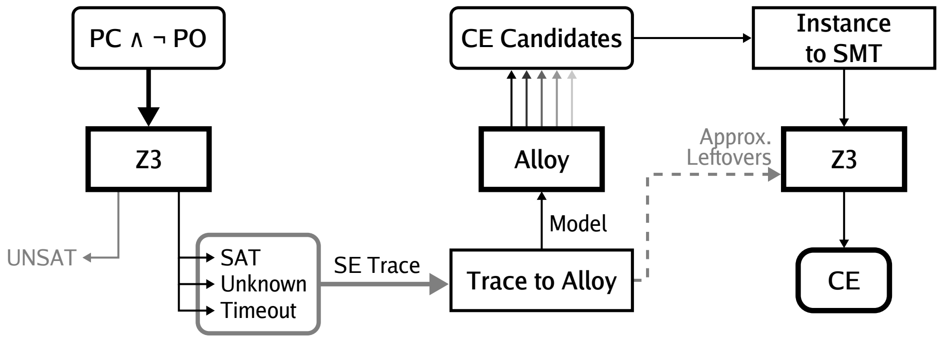


Figure 2.1: A diagram of the pipeline in our counterexample generation approach.

The "pipeline" of our approach for counterexample generation is shown as a diagram in Figure 2.1. When a verification failure occurs, we can take the informations from the symbolic execution and encode them into an Alloy model to generate a counterexample. Consider the formula $PC \land \neg PO$. In our case, PC corresponds to the information about the symbolic execution state the verifier was in when the verification failure occurred. $sat(PC \land \neg PO)$ is the last query that was performed to the SMT solver and could not be falsified. Both conjuncts of the formula can be retrieved from Silicon as part of the symbolic execution trace.

Once we have retrieved the symbolic execution trace, we can use it to encode an Alloy model of a symbolic state we are interested in, for example the one where the failure occurred. Chapter 3 describes this translation in detail. With our model, we can run an analysis in Alloy, via its API, and get counterexamples back.

Depending on the modeling technique we use, not all instances generated by Alloy might be actual counterexamples. If we apply approximations when we build our model of the symbolic state, we may end up generating instances that do not satisfy $\neg PO$, which therefore are not counterexamples to the specification, or we may end up with instances that do not satisfy PC, meaning that they contradict some of facts known to the symbolic execution engine.

With our technique, we perform an over-approximation of the counterexample space (modulo bounds). This means that all counterexamples within the bounds specified when performing the analysis of the model will be identified, but there might also be additional instances found by Alloy which are not actual counterexamples (i.e. they are *spurious counterexamples*).

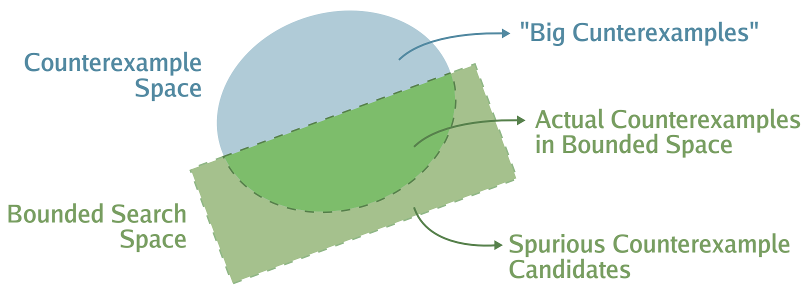

"Big Cunterexamples"
Counterexample Space
Actual Counterexamples in Bounded Space
Bounded Search Space
Spurious Counterexample Candidates

Figure 2.2: A diagram showing the relation between the whole counterexample space, the bounded space we explore, and the spurious counterexamples that might result from approximating the formula.

Figure 2.2 shows the relation between the whole counterexample space and what we can explore via our bounded search. Note that the "Big counterexamples" area, that we do not explore, is potentially infinite, depeding on the specification in the original program. If we did not approximate the formula, then the bounded search we perform would only explore the *actual counterexamples in the bounded space*, if instead we were to approximate parts of the formula, we would allow Alloy to generate *spurious counterexample candidates*, which are not part of the *counterexample space*

If we decide to use approximations, then we need an additional phase at the end of our pipeline, where we employ an oracle to distinguish actual counterexamples from spurious ones. In our case, Z3 can act as the oracle: we can take the counterexample candidates generated by Alloy and all the "leftover" constraints that were approximated or ignored when building the model, and encode them back into an SMT2 formula. This new formula could then be analysed by the SMT solver in order to filter out spurious counterexamples: for these, the formula would not be satisfiable as they would violate the constraints that were previously approximated. In our current implementation the “filtering phase” of the pipeline is not implemented.

In addition to the facts coming from the symbolic execution trace, our approach would allow for additional facts to be added to the encoded model. These might be used to constrain the counterexamples to a more manageable size (for example, by limiting the size of sets) or to explicitly exclude non-interesting configurations. The ability for users to specify additional constraints for the counterexample search at the Viper level could be integrated as part of the graphical interface of the debugger. This feature has not been implemented in the current project, but a similar effect can be achieved by adding preconditions or assumptions to the program.

In the current implementation, we do not retrieve Z3’s partial model, but delegate counterexample generation to Alloy alone. It would be of course possible to encode the facts found by Z3 as additional constraints into the Alloy model. It is important to note that we have no guarantees about the completeness of Z3’s model in the cases where the verification did not return *sat*.

In these sections, we explained our approach with the final goal of generating counterexamples. This applies to situations where there is a mismatch between the program specification and the failed assertion (the specification is too weak or the assertion is too strong), as well as cases where there is an implementation problem. The pipeline also allows to generate **examples** of the program’s symbolic states (by not encoding the failed proof objective into the Alloy model) This may still be beneficial, for example to find unexpected situations even before a verification failure occurred.

3 Translation to Alloy

In this chapter, we describe our approach to translating the information about a symbolic execution state, that we get from Silicon's symbolic execution trace, into a model in the Alloy language. This model can be used to generate counterexamples (or just examples) of the program's specification.

When defining generic translation rules we will use the following informal notation, where by *Silicon term* we mean literals, variables and expressions that Silicon uses to encode facts about the program.

- translate(t) denotes the translation of a Silicon term t following the rules described in Section 3.1.
- sanitize(i) denotes the sanitization of identifier i by replacing all `@` symbols appearing in it with an underscore.
- *Italic* text denotes either arbitrary Silicon terms or values computed during the translation. In this second case their meaning is explained in the accompanying text.
- Strings in a **`monospaced, bold`** font represent plain text in either Viper or Alloy. They are fixed input or output strings.
- Ellipses are used to denote a repetition of the previous pattern, for example a concatenation of terms with the same operator.

As an example, the following rule describes the translation of a Viper conjunction into Alloy: a conjunction of n Viper terms is translated by applying the translate function to each of the terms, conjoining them by **`&&`** and wrapping them in parentheses.

translate($term_1$ **`&&`** … **`&&`** $term_n$) =
    `(`translate($term_1$) **`&&`** … **`&&`** translate($term_n$)**`)`**

Silicon's symbolic values have names of the form `v@01@0` and are translated just by replacing the `@` symbol with an underscore, but for the sake of readability and space-saving we have simplified the names by removing their last part in the snippets presented in this chapter.

3.1 Modeling Terms

In this section we describe the rules used to translate Silicon terms that appear in the symbolic execution trace.

3.1.1 Binary Operations

Binary operations are translated based on the sort of their operands and on their operator.

When we implement operations via predicates (e.g. for permissions), we explicitly declare a fresh name for the result. This is because we want to have precise control over the fact that an instance representing the result does exist. If we were implementing these operations via Alloy functions, rather than predicates, there would be no way to ensure that the result of an operation is always present in the current universe, other than adding an explicit generator axiom stating that a result exists for each pair of operands. The problem with generator axioms is that they explode the complexity of the model, since they force the existence of results which may not be needed, hence why we chose to declare the results explicitly.

Operations involving terms of sort **`Set`**, **`Seq`**, or `Multiset` are translated according to the following scheme

$$\begin{aligned}&\mathsf{translate}(term_1\ \ op\ \ term_2) = \\ &\qquad opPred\texttt{[}\mathsf{translate}(term_1)\texttt{, }\mathsf{translate}(term_2)\texttt{, }term_{res}\texttt{]}\end{aligned}$$

where *opPred* is a predicate that implements operation *op* for the specific type of the operands. For example, the union operation between sets is implemented by the `pred_union` predicate. Here, $term_{res}$ is a fresh name in the signature of the result's type, and is passed to the predicate so it can be constrained as the actual result value.

The operations on collections that result in a Boolean type, are translated to binary predicates without the result parameter. These include, for example, the **`in`** or `subset` operations on sets. The translation has the form

$$\mathsf{translate}(term_1\ \ op\ \ term_2) = opPred\texttt{[}\mathsf{translate}(term_1)\texttt{, }\mathsf{translate}(term_2)\texttt{]}$$

Operations on permissions are also translated to predicate calls of the form

$$\begin{aligned}&\mathsf{translate}(perm_1\ \ op\ \ perm_2) = \\ &\qquad \textbf{\texttt{perm\_}}opPred\texttt{[}\mathsf{translate}(perm_1)\texttt{, }\mathsf{translate}(perm_2)\texttt{, }perm_{res}\texttt{]}\end{aligned}$$

Again, each operator *op* has a corresponding predicate implementing it. The third parameter, $perm_{res}$, is a fresh instance for the result, but in this case it is used in the translation of *all* operations.

Binary equalities where both operands have Alloy's built-in `PrimitiveBoolean` type (for example, because the operand is translated into a predicate call) are translated with a double implication, because the equality operator cannot be used on `PrimitiveBoolean`. On the other hand, Binary equalities between operands of type `Bool` (our custom signature for boolean types) are translated directly, since `Bool` is a signature like any other. For all other binary operations involving Booleans, the operands are wrapped into a `LogicalWrapper` such that Alloy's logical operations (=>, <=>, ...) can be applied to them directly. The special handling of Booleans is described in more details in Section 3.5.

Currently, integers are implemented via a custom `Integer` signature, that simply wraps a value of the built-in **`Int`** type. Operations between integers are implemented by accessing the wrapped value directly. Arithmetic operations are implemented via the built-in functions `plus`, `minus`, `mul`, `div`, and `rem`. The resulting code is a simple function application:

$$\begin{aligned}&\text{translate}(term_1\ \ op_{arith}\ \ term_2) =\\ &\qquad term_{res}\,\mathtt{.val\ =\ } fun_{arith}\mathtt{[}\text{translate}(term_1)\mathtt{.val\ \ ,\ }\text{translate}(term_2)\mathtt{.val]}\end{aligned}$$

Here, $term_{res}$ is a fresh instance of the `Integer` signature, that represents the result of the operation. Comparisons are also applied directly to the wrapped values. The relational operators are the same in Viper and Alloy. The operands are translated recursively. The translation is as follows:

$$\begin{aligned}&\text{translate}(term_1\ \ op_{comparison}\ \ term_2) =\\ &\qquad \mathtt{(}\text{translate}(term_1)\mathtt{.val}\ \ op_{comparison}\ \ \text{translate}(term_2)\mathtt{.val)}\end{aligned}$$

The built-in integers are limited in bit-width by the value set in when declaring the **`run`** command, therefore the results of some operations might not be representable. This problem is discussed in Section 6.3.

3.1.2 Unary Operations

There are only a handful of unary operations that we have to translate: the logical negation operator, and the cardinality operations on collections.

The negation operation is translated directly with Alloy's `!` operator, unless the operand has sort **`Bool`**, in which case the operation is translated as

`isFalse[`translate(*operand*)`]`

This is because we need to represent Booleans via a custom signature, which cannot be used directly in logical operations. The issue is discussed in Section 3.5.

The cardinality operations on collections are of the form

$$\text{translate}(\texttt{|}\,term_{coll}\,\texttt{|}) = \\ \quad coll\texttt{\_cardinality[}\text{translate}(term_{coll})\texttt{]}$$

Where *coll*`_cardinality` is a function defined in the preamble the implements the cardinality operation for the specific collection type (e.g. `set_cardinality`).

3.1.3 And, Or, Ite

Conjunctions, disjunctions, and if-then-else expressions are translated with their equivalent in the Alloy language.

$$\begin{aligned}
&\text{translate}(term_1 \;\texttt{\&\&}\; \dots \;\texttt{\&\&}\; term_n) = \\
&\quad \texttt{(}\text{translate}(\text{LogicalWrapper}(term_1)) \;\texttt{\&\&} \\
&\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
&\qquad \text{translate}(\text{LogicalWrapper}(term_n))\texttt{)} \\
&\text{translate}(term_1 \;\texttt{||}\; \dots \;\texttt{||}\; term_n) = \\
&\quad \texttt{(}\text{translate}(\text{LogicalWrapper}(term_1)) \;\texttt{||} \\
&\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
&\qquad \text{translate}(\text{LogicalWrapper}(term_n))\texttt{)} \\
&\text{translate}(cond \;\texttt{?}\; then \;\texttt{:}\; else) = \\
&\quad \texttt{(}\text{translate}(\text{LogicalWrapper}(cond)) \;\texttt{implies}\; \text{translate}(then) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\quad \text{translate}(else)\texttt{)}
\end{aligned}$$

To ensure that the terms in conjunctions, disjunctions and in the condition of the if-then-else expression are of type `PrimitiveBoolean`, they are wrapped in a LogicalWrapper before being translated, which makes sure the resulting operation has the right type. This issue and the need for a LogicalWrapper are discussed in more details in Section 3.5.

3.1.4 Let expressions

Let expressions exist both in Viper and Alloy, so their translation is almost identical. The only difference is that the encoded body is potentially expanded with additional facts, introduced during its translation to constrain fresh variables. The facts are added inside the body because they may reference one of the variables bound by the actual let expression.

$$\begin{aligned}
&\text{translate}(\texttt{let}\; var_i \;\texttt{=}\; term_i\texttt{,}\; \dots \;\texttt{in}\; body) = \\
&\quad \texttt{(let}\; (\text{sanitize}(var_i) \;\texttt{=}\; \text{translate}(term_i)\texttt{,}\; \dots) \;\texttt{|} \\
&\qquad\qquad addFacts\dots \;\texttt{\&\&}\; \text{translate}(body)\texttt{)}
\end{aligned}$$

3.1.5 Combines

As explained in Schwerhoff's thesis [11, sec. 3.2], snapshots are used to represent the values of heap locations to which an assertion has permissions. In Silicon, snapshots are represented as nested pairs forming a binary tree, built via the *pair* function:

$$pair : Snap \to Snap \to Snap$$

These nested pairs, in the symbolic execution trace, take the form of *Combine* operations, having two snapshots as arguments and being themselves an object of type *Snap* (their combination). Silicon makes use of *Sortwrapper* (or *box/unbox* functions, as they are referred-to in the thesis) to wrap object of various types before combining them in a snapshot. They are discussed in the next section.

Encoding combine operations into the Alloy model is important to ensure that the generated counterexample is valid. Consider the program in Listing 3.1. The verification fails because, unless `x != null`, we don't know for certain that `x.v` is equal to 3. Without the information "stored" in the combines, we wouldn't know of the relation between the two fields in the predicate, so Alloy could come up with counterexamples where `x.v` is indeed different from 3, but where `x.r` might be different from **`null`**.

```
field r: Ref
field v: Int

predicate foo(x: Ref) { acc(x.r) && acc(x.v) }

method test(x: Ref)
  requires acc(foo(x))
  requires unfolding acc(foo(x)) in x.r != null ==> x.v == 3
{
  unfold acc(foo(x))
  assert x.v == 3  // Verification fails
}
```

Listing 3.1: An example of a program where encoding the combine information is necessary to ensure that we generate correct counterexamples.

In this example, the Symbolic Execution Log gives us the following information about the symbolic state when had the verification failure:

Heap	Store
`x@1.v` : $Int \to$ `$t@7 # ` W	`x` : $Ref \to$ `x@1`
`x@1.r` : $Ref \to$ `$t@6 # ` W	

but we learned about the relation between `x.r` and `x.v` in a different symbolic state, when we temporarily unfolded the predicate. This is encoded in the path condition

```
!($t@5 == Null) ==> ($t@4 == 3)
```

and the link with the current symbolic state is given by other two path conditions, that include the combine operations for the snapshots.

```
($t@19 == Combine(SortWrapper($t@6, Snap), SortWrapper($t@7, Snap)))
($t@19 == Combine(SortWrapper($t@5, Snap), SortWrapper($t@4, Snap)))
```

Here, `$t@19` is the snapshot of the `pair` predicate, when it was folded. With these additional facts, we learn that the symbolic values in the current state are equal to the symbolic values in the **`unfolding`** expression.

Combines are binary operations in the symbolic execution trace, and are translated via the `combine` helper function, defined in the preamble (see Section 3.7.1). The result of their translation has the form

$$\text{translate}(\texttt{\textbf{Combine(}}\mathit{term}_1\texttt{, }\mathit{term}_2\texttt{\textbf{)}}) = \\ \quad \texttt{\textbf{combine[}}\text{translate}(\mathit{term}_1)\texttt{, }\text{translate}(\mathit{term}_2)\texttt{, }\mathit{term}_{res}\texttt{\textbf{]}}$$

Where term_{res} is a fresh instance in the `Snap` signature. The `combine` predicate makes sure that the fresh snapshot is an instance of the `Combine` signature, so it hase the `left` and `right` fields, and constrains them to be equal to the two snapshots being combined.

3.1.6 Sortwrappers, First, and Second

Sortwrappers are used by Silicon to wrap primitive values in order to be able to use them where snapshots are expected, or the other way round. As explained in the previous section, snapshots summarise values on the heap. Primitive values are wrapped into a snapshot via a $Sortwrapper(v, Snap)$ or unwrapped via, for example, $Sortwrapper(v, Int)$ operations. These terms correspond to the *box* and *unbox* functions defined in Schwerhoff's thesis [11] as:

$$box_S : S \to Snap$$
$$unbox_S : Snap \to S$$

Moreover, Schwerhoff also defines the *first* and *second* functions, to deconstruct pairs of snapshots (*Combines* in our case):

$$first : Snap \to Snap$$
$$second : Snap \to Snap$$

The construction and deconstruction of snapshots is axiomatised as:

$$\forall s_1, s_2 : Snap \cdot first(pair(s_1, s_2)) = s_1 \land second(pair(s_1, s_2)) = s_2$$

These operations are available to us as Silicon terms, where the *box* and *unbox* functions correspond to `Sortwrappers`, and the *first* and *second* functions are the unary `First` and `Second` operations respectively.

Sortwrappers may be translated in two ways, depending on the resulting type. If the sortwrapper should result in an object of *Snap* type (i.e. if it correponds to the *box* operation), then it is translated by declaring a fresh name in the Snap signature and constraining its value via the `sortwrapper_new` predicate.

$$\begin{aligned}&\text{translate}(\texttt{\textbf{SortWrapper(}}\textit{term}\texttt{\textbf{, Snap)}}) = \\ &\quad \texttt{\textbf{sortwrapper\_new[}}\text{translate}(\textit{term})\texttt{\textbf{,}}\ \textit{term}_{\textit{res}}\texttt{\textbf{]}}\end{aligned}$$

Where $\textit{term}_{\textit{res}}$ is a fresh name representing the resuling snapshot. The predicate `sortwrapper_new` is defined in the static preamble of our model, described later on in Section 3.7.1, and makes sure that the `wrapped` relation on the resulting snapshot points to the term being wrapped.

When the resulting type of the sortwrapper is not *Snap*, then we are performing an *unbox* operation and the translation is as follows:

$$\text{translate}(\texttt{\textbf{SortWrapper(}}\textit{term}\texttt{\textbf{,}}\ \textit{sort}\texttt{\textbf{)}}) = \texttt{\textbf{(}}\text{translate}(\textit{term})\texttt{\textbf{).wrapped}}$$

Here, we know that *term* must be an object of type *Snap* that had been previously wrapped, therefore we can access the `wrapped` relation directly.

The *first* and *second* functions correspond to unary operations with the same name and are translated as follows:

$$\begin{aligned}&\text{translate}(\texttt{\textbf{First(}}\textit{term}\texttt{\textbf{)}}) = \texttt{\textbf{(}}\text{translate}(\textit{term})\texttt{\textbf{).left}}\\ &\text{translate}(\texttt{\textbf{Second(}}\textit{term}\texttt{\textbf{)}}) = \texttt{\textbf{(}}\text{translate}(\textit{term})\texttt{\textbf{).right}}\end{aligned}$$

Just like for the unboxing operation, when we are retrieving the first or the second element of a snapshot, then that snapshot must be the result of a *pair* operation (a `Combine` in our case) therefore we can simply access the `left` and `right` relations on that signature.

3.1.7 Quantifiers

Viper's universal and existential quantifiers are translated into Alloy's **`all`** and **`some`** quantifiers, respectively. In general, we sanitize the names of the variables declared in the quantifier and translate their sorts to build the parameters of the Alloy quantified formula, then complete it with the translation of the body.

Sometimes we define additional constraints during the translation of a term, so that we can constrain the values of fresh variables that we introduce. For example, when using predicates to implement some operations.

In case we gather such facts during the translation of the quantifier's body, we append them to the translated body, as additional constraints that have to be satisfied.

A quantifier of the form

$$\textbf{forall}\ (v : Sort, \ldots)\ \texttt{::}\ body$$

is translated to

$$\texttt{(}\textbf{all}\ (\mathsf{sanitize}(v) : \mathsf{translate}(Sort)\texttt{,}\ \ \ldots)\ \texttt{|}\ (addFacts\ \texttt{\&\&}\ \ldots)\,\mathsf{translate}(body)\texttt{)}$$

The same formula, but with **some** instead of **all**, would be used to translate and existential quantifier, except in the case described later in this section, when the quantifier appears inside a negation.

As a more practical example, consider the following snippet in which we have a sequence of sets of integers, constrained so that all the elements in the sequence are the singleton $\{1\}$.

```
var ns: Seq[Set[Int]]
assume forall s: Set[Int] :: s in ns ==> s == Set(1)
```

```
1 one sig fresh_quantifier_vars_0 {
2   temp_1': Set_Integer -> lone Set_Integer
3 }
4 fact { all s: Set_Integer |
5   set_singleton[1, fresh_quantifier_vars_0.temp_1'[s]] &&
6   (seq_in[s, ns] => (s = fresh_quantifier_vars_0.temp_1'[s]))
7 }
```

Listing 3.2: The translation of a quantifier in Alloy when fresh variables are needed.

The translation of the quantifier in the assumption is shown in Listing 3.2. Set singletons are implemented via predicates (see Section 3.2). Therefore in order to "instantiate" one, we need to get a fresh variable that we can constrain by passing it to the predicate. This additional constraint defining the singleton appears on line 5 in the translation, whereas line 6 corresponds to the body of the original quantifier. Note that in this example, the fresh variable that we declare is actually part of relation `temp_1'`. This happens because we are declaring fresh names inside a quantification, therefore we might need a fresh name for each of the distinct assignments of the quantified variable. In this example, there is no need to introduce a relation, because the predicate call does not depend on the quantified variables, so the translation could be implemented as

```
one sig temp_1' in Set_Integer {}
fact { all s: Set_Integer |
  set_singleton[1, temp_1'] && (seq_in[s, ns] => (s = temp_1'))
}
```

However, in the general case the additional constraints do depend on quantified variables, hence the need for declaring fresh names in relations.

These additional constraints might become problematic inside a negated existential quantifier, because Alloy may try to falsify those instead of the facts that appeared in the original body.

Whenever the negation of an existential quantifier is encountered, it is transformed into a universal quantification and the negation is pushed into its body according to the equivalence

$$\neg\exists\, v \in V.\, P(v) \;\equiv\; \forall\, v \in V \cdot \neg P(v)$$

The translation of the term will then proceed normally, but this small reformulation has the consequence that all additional facts will be added to the body of the quantifier outside of the negation, resulting in the translation

$$\neg\exists\, v \in V.\, P(v) \;\rightarrow\; \forall\, v \in V \cdot \mathit{addFacts} \wedge \neg P(v)$$

which correctly models the fact that the constraints we want Alloy to falsify are those from the original formula, not the additional facts.

Another approach to solving this problem, which is not currently implemented in the debugger, would be to declare the additional facts in a separate quantifier, which is equivalent to the other approach.

$$\forall\, v \in V \cdot \mathit{addFacts} \wedge \neg P(v) \;\equiv\; \forall\, v \in V \cdot \mathit{addFacts} \,\wedge\, \neg\exists\, v \in V.\, P(v)$$

Declaring the additional facts in a stand-alone quantifier could also be applied in general, and not only when the quantifier is negated.

3.1.8 Function Applications

Function application terms have the form $\mathit{Application}(\mathit{name}, \mathit{args}, \mathit{resSort})$. Where *name* is the name of the applicable being invoked, *args* is a sequence of argument terms, and *resSort* is the sort of the function.

Applications are translated into the form

$$\begin{aligned}&\mathsf{translate}(\mathtt{\textbf{Application(}}\mathit{name}\texttt{, } \mathit{arg}_1\texttt{, } \ldots\texttt{, } \mathit{resSort}) = \\ &\qquad \mathit{FunSig}\texttt{.}\mathsf{sanitize}(\mathit{name})\texttt{[}\mathsf{translate}(\mathit{arg}_1)\texttt{, } \ldots\texttt{]}\end{aligned}$$

Where *FunSig* is the signature in which a relation for *name* will be declared. This may be signature `Fun` for regular functions defined in the source program, or signature `PTAKEN` for pTaken functions introduced by Silicon (see Section 3.7.7).

Whenever we translate a function application, we also make sure to record the function's type signature, so that it may be declared in the model later, as we will discuss in Section 3.7.8.

3.1.9 Field Lookups

When quantified field chunks are involved, field accesses are represented as a lookup in a certain snapshot map. Consider the following snippet, where we use quantified permissions to denote access to field `val` for all reference in set `nodes`

```
method test(nodes: Set[Ref], n: Ref)
    requires forall r: Ref :: r in nodes ==> acc(n.val)
    requires n in nodes
{
    var v: Int := n.val
    // Encoding the state here
}
```

In the state being modeled, we have the following store, where we see variable v has a lookup term as its symbolic expression.

<u>*Store*</u>

$nodes : Set[Ref] \rightarrow nodes@91$

$n : Ref \rightarrow n@92$

$v : Int \rightarrow Lookup(val, sm@98(), n@92)$

For each field used in a lookup we encounter, we declare a relation with the same name as part of the `Lookup` signature, and the lookup operation is translated into an application of that relation, as follows

```
Lookup.val[sm_98_01, n_92_01]
```

The `Lookup` signature and all the fresh relations we introduce need to be declared in the model. Their declaration will be discussed in Section 3.7.8.

3.2 Built-in Collections

The built-in collection types each have their own abstract signature to represent them. Alloy does support built-in sets and sequences, but it is not possible to declare nested sets or sequences using the **`set`** and **`seq`** keywords. Still, our signature-based representations of the Viper collection ultimately rely on the built-in relations to “store” their elements. This means that we can also implement operations on our custom signature by defining them in terms of operations on the built-in relations.

Ideally, we would like to model operations on the Viper collections via functions, but it is not possible to ensure that new instances are created when they are applied. We want precise control over instantiation in order to ensure that operations always result in a concrete instances.

Because of this reason, we model the operations via predicates and we declare a fresh signature to represent each of their results. The result signature is declared outside of the predicate definition and is passed to the predicate as an additional argument, so that it can be properly constrained. Declaring fresh names externally also requires flattening operations and chaining them one after the oher.

The Viper snippet

```
method areDisjoint(s1: Set[Ref], s2: Set[Ref]): Bool
{
  var c: Int := |s1 intersection s2|
  // Encoding the state here
}
```

would be translated as follows

```
one sig s1_91 in Set_Ref {}
one sig s2_92 in Set_Ref {}
fact { Store.s1' = s1_91 }
fact { Store.s2' = s2_92 }
one sig temp_0' in Set_Ref {}
fact { set_intersection[s1_91, s2_92, temp_0'] &&
       Store.c' = set_cardinality[temp_0'] }
```

Representing the built-in collections concretely rather than axiomatising them, enables us to detect some situations when the verifier raises spurious verification errors due to the incompleteness of the axioms used to model collections in Z3. Some more details are discussed in Section 6.1.2.

An alternative approach to avoid declaring fresh variables manually would be to encode the operations on collections as ternary relations and adding facts to constrain the third term in the relation to represent the result of the operation on the arguments (the other two atoms in the relation). This is possible but risks causing an explosion in the complexity of the model [10, sec. 5.3.1].

3.3 Sorts

Silicon defines symbolic expressions as terms and formulas of a many-sorted first-order logic, and, for each Viper type, it employs a matching sort in the signature of its logic. When we talk about sorts, we refer to Silicon's internal mapping of the Viper types.

During the encoding of a program, sorts are translated to signatures. For simple sorts (without elements) the mapping is the following:

- The **Ref** sort is translated into the `Ref` signature, whose encoding is discussed in Section 3.7.2.

- The Snap sort is translated into the Snap signature, defined in the preamble (see Section 3.7.1).
- The **Int** sort is modeled via a custom Integer signature. This is discussed in Section 3.7.1.
- The **Bool** sort is translated as the Bool signature, provided by the util/boolean module, included in Alloy. A discussion about the treatment of Boolean-typed values can be found in Section 3.5.
- The **Perm** sort is modeled via the Perm signature, defined in the static preamble (Section 3.7.1). In Section 3.4 there is a discussion about ways of modeling fractional permissions, their advantages and disadvantages.

Types "with elements", such as the collection types or field value functions, are translated by declaring fresh signatures based on the types involved. For example, the Viper sort **Set**[**Set**[**Ref**]] would be translated by recursively encoding the element signatures, having the following result

```
sig Set_Ref extends Set {} {
  set_elems in Ref
}
sig Set_Set_Ref extends Set {} {
  set_elems in Set_Ref
}
```

The procedure is also applied to the other sorts with elements, mentioned above.

User sorts, defined via domains, are translated slightly differently to avoid name clashes. As an example, the user sort Array would be translated into the signature declaration

```
sig User_Array' {}
```

Both the collection sorts and the user sorts are declared in the "gathered facts" section of the model, described in Section 3.7.8.

3.4 Permissions

Values of Viper's **Perm** type type are represented by instances in the Perm signature. As mentioned in Section 3.1, the operations on Perm instances are abstracted via predicates, that update the actual internal representation of the permission objects.

We experimented with two representations for permissions: one where we explicitly model numerator and denominator fields, and one where all permissions are symbolic and we only model the relations between them.

With the first approach, the definition of the Perm, R, and W signatures relies on Alloy **Int**. The definitions are the following.

```
abstract sig Perm {
    num: one Int,
    denom: one Int
} {
    num >= 0
    denom > 0
}
one sig W in Perm {} {
    num = 1
    denom = 1
}
one sig Z in Perm {} {
    num = 0
    denom = 1
}
```

The predicates that implement operations between these permissions effectively compute those operations on rational numbers (defined by `num` and `denom`). The implementation of these predicates is shown in Appendix B.

In second approach, permissions are just symbols and have no fields.

```
abstract sig Perm {}
one sig W in Perm {}
one sig Z in Perm {}
```

Information about these instances is encoded in relations, declared in the `PermRelations` signature. In this case, the predicates that implement the operations between permissions keep these relations up to date based on the facts we know about a certain permission value.

```
one sig PermRelations {
  eq: Perm -> Perm,
  lessthan: Perm -> Perm,
  add: Perm -> Perm -> lone Perm,
  new: Int -> Int -> lone Perm
} {
  all a:Perm | perm_equals[a, a]
}
```

The fact that in both the approaches the interface with permissions is defined by the predicates that implement their operations, means that we are free to change the way we implement the `Perm` signature without having to change the rest of the model. Both approaches have their advantages and drawbacks, discussed in Chapter 6. The main issue with the first approach is the boundedness of Alloy integers, whereas with the second technique the need to approximate some operations becomes problematic.

3.5 Treatment of Booleans

Alloy lacks a user-accessible Boolean type. The language does have a type for Boolean-valued expressions, called `PrimitiveBoolean`, but this is not accessible to the user: it is not possible to declare a relation which contains Boolean terms. The system offers a utility module that implements Boolean values in the Alloy language and that we use to model Boolean values from Viper programs. The problem with this approach is that there is a disconnect between the `Bool` signature that we use to model Viper's Boolean type and the type that can be used in assertions (`PrimitiveBoolean`), requiring an additional mechanism to allow us to mix them together.

Consider the following example, where we use a variable of **`Bool`** type in the left-hand side of an implication in the method's precondition

```
method test(b: Bool, v: Int)
  requires b ==> v > 3
{
  assert true
}
```

When modeling the symbolic state on the assertion, we would like to express the path condition gathered in the method's precondition with the following fact in Alloy's model:

```
fact field { b => v > 3 }
```

We encounter a problem if we try doing this: we cannot declare variable `b` with type `PrimitiveBoolean` and, at the same time, it is not possible to use any expression of a type other than `PrimitiveBoolean` in a logical operation (the implication, in this case). In this example, we can solve this problem by declaring the variable with type `Bool` (from the `util/boolean` module) and use the `isTrue` predicate whenever we need a `PrimitiveBoolean` type. We model the path condition as

```
fact { isTrue[b] => v > 3 }
```

This problem also happens in the inverse direction, for example when we want to call a function with the result of a logical operation, like in the following snippet.

```
function f1(b: Bool): Int
method foo(v: Int) {
  var i: Int := f1(v > 0)
  assert true
}
```

Here, the path condition from the function call is translated by introducing a conditional expression[10, app. B.7.7] that returns the appropriate `Boolean` atom.

```
fact { i = Fun.f1[ Unit, ((v > 0) implies True else False)] }
```

These type of situations occur in various moments during the translation of the program facts. Whenever we know that either only the `PrimitiveBoolean` type or the `Bool` type are expected, we insert auxiliary `LogicalWrapper` or `BooleanWrapper` operations, that wrap the expressions being translated and eventually transform the result, in case the type of the sub-expression is not the one expected by the outer expression.

The insertion of the `LogicalWrapper` happens in the following moments:

- When translating an expression that will finally end up in a stand-alone fact (for example a path condition).
- When translating the body of a quantifier.
- When translating the condition of an "if-then-else" expression.
- When translating each of the elements in an `And` or an `Or` term.
- When translating the operands of a boolean binary expression, unless they both have type `Bool` and the operation is a binary equality.

The insertion of a `BooleanWrapper` happens:

- When translating the value of a store variable that is a logical operation.
- When translating the arguments of an application where the expected type is Boolean.

3.6 Required Information

The translation of a symbolic verification state into an Alloy model has several prerequisites. In our case, these are all provided to us via the *Symbolic Execution Logger*. This tool is integrated into Silicon and provides information about the actions performed during verification and about the symbolic states they were applied to. More details about it can be found in Chapter 4.

The most fundamental piece of information for the translation is the execution trace, which contains the *Store*, the *Heap*, and the *Path Conditions* for each state generated by by Silicon during symbolic execution of the program.

If verification failed due to an SMT incompleteness, then we need to encode the problematic assertion into our model, as it is essential to distinguish counterexamples to the program's specification from other instances of the symbolic state being modeled. More details on this are discussed in Chapter 2.

Each time Silicon removes permissions from heap chunks, the amount removed from the chunk for a specific reference is abstracted into a *pTaken* function. These functions are needed to determine how much permission is left in a certain heap chunk (if any) and to correctly model presence or absence of fields. The translation of these functions is described in Section 3.7.7.

If one or more user-defined domain types are used (for example arrays), their axioms have to be encoded in the model as well, in order to ensure that the generated instance respects the desired semantics.

Function postconditions are axiomatised in Silicon. The resulting axioms have to be retrieved in order to correctly model function results.

3.7 Structure of the Model

We translate a symbolic state into a single Alloy model, in which we have conceptually separate parts, each encoding a different aspect of the original program. The model is structured according to the parts described in the following sections.

In practice, we can distinguish two big groups in the model:

- Definitions for facts from the symbolic state (encoding store, heap, path conditions and permissions), in the first four sections after the preamble.
- All the other definitions, that encode additional facts, functions axioms, domains, etc.

We will be using the program in the following snippet as a running example to show how each section of the model is translated. Its complete encoding can be found in Appendix C.

```
1  field val: Int
2  method example(nodes: Set[Ref], n1: Ref, n2: Ref)
3  requires forall r: Ref :: r in nodes ==> acc(r.val)
4  requires n1 in nodes
5  requires n2 in nodes
6  {
7    exhale acc(n1.val)
8    var v: Int := n2.val  // Verification fails here
9  }
```

Listing 3.3: In this Viper snippet we have a set of references, `nodes`, and we have full permission to the `val` field for each of the references in this set. We know that `n1` and `n2` belong to `nodes`, and we have permission to their field, at least initially. On line 8, we exhale all permissions for field `val` from `n1`, and on line 10 we try reading `val` from reference `n2`. The verification fails because `n1` and `n2` *may be* aliases, therefore we might not have permission to `n2`'s `val` field anymore.

3.7.1 Preamble

The first part in the model is a static preamble, which contains signature definitions for the built-in types. Operations on them are implemented via predicates and functions, that are also defined in the preamble.

In the preamble we define signatures for Silicon snapshots and the operations involving them. Snapshots are used in Silicon to abstract over values of heap locations to which an assertion includes permissions. Heap snapshots are combined into a binary tree structure via the *combine* operation, which takes two snapshots and returns another one representing their pair. The values in the leaves of this structure are either the empty *Unit* snapshot, or heap values, contained in a *SortWrapper*. These wrappers allow using a value of any type where a snapshot-typed object is expected. Snapshots are needed in the debugger because they encapsulate information which may be needed to produce a valid counterexample, as discussed in Section 3.1.5. A more thorough explanation of how snapshots work can be found in Schwerhoff's thesis [11, sec 3.1].[1]

The abstract snapshot type is represented by the `Snap` signature. The `Unit` signature represents empty snapshots. The `SortWrapper` signature represents *sortWrappers*, and is effectively a container of objects of other sorts. The `sortwrapper_new` predicate is used to state the constraint that a newly-declared sortwrapper contains a certain value. The `Combine` signature represents combinations of two snapshots, and the `combine` predicate is used to constrain an instance of the signature to be a combination of two specific snapshots.

```
abstract sig Snap {}
one sig Unit extends Snap {}
abstract sig SortWrapper extends Snap {
  wrapped: one univ
}
abstract sig Combine extends Snap {
  left: one Snap,
  right: one Snap
}

pred sortwrapper_new [ e: univ, sw: Snap ] {
  sw in SortWrapper
  sw.wrapped = e
}
pred combine [ l, r: Snap, c: Combine ] {
  c.left = l && c.right = r
}
```

[1] Note that, in the thesis, the *combine* and *sortWrap* functions are defined as *pair* and *box* respectively. We use the names that these operations have when they appear in the symbolic execution trace and in Silicon's implementation.

The preamble also defines (or imports) all signatures needed to encode Viper's built-in types (**`Perm`**, **`Int`**, and **`Bool`**) and collections (**`Set`**, **`Seq`**, and `Multiset`).

To model Viper's **`Perm`** type we declare the `Perm` signature and define a series of predicates (e.g. `perm_plus` or `perm_less`) that implement the operations on them. We experimented with two implementations of fractional permissions using, different techniques. A more detailed discussion of the two implementations is presented in Section 3.4.

To model Viper's integers we declared the custom `Integer` signature. It is just a wrapper with a field `value` holding a built-in Alloy integer. The definition is as follows:

```
abstract sig Integer {
  value: one Int
}
```

The reason behind introducing this wrapper has to do with modeling total functions and the way Alloy instantiates the built-in integers. A detailed discussion of this problem can be found in Section 6.3.3. Operations between `Integer` objects are implemented via the built-in functions provided by alloy and by accessing the wrapped field `value` directly.

We import the `boolean` module, which provides a definition for Boolean values, used to encode Viper's **`Bool`** sort, and a series of utility functions to work with them. The `ternary`, and `relation` modules are also included to provide helper functions for the manipulation of relations.

We define signatures `Set`, `Seq`, and `Multiset` to model the built-in collections. Again, operations on these types are implemented via predicates. More details about the implementation of built-in Viper collections can be found in Section 3.2.

The complete definition of all these signatures can be found in Appendix B.

3.7.2 Modeling References

In the second part of the model, we define the reference signature, which represents Viper's **`Ref`** type. The definition is not part of the static preamble because it has to include a relation for each of the fields that a reference might have access to in the symbolic state being modeled.

Each of the fields known from the symbolic heap is added to the signature with multiplicity **`lone`**. Note that not all fields in the original Viper program are necessarily modeled in the reference signature: if no permission to a field is held in the symbolic state being modeled, then that field is not declare in the `Reference` signature. At the same time, the **`lone`** multiplicity expresses the possibility that a field may not to exist on an instance: not *all* references in the program might have permission to it all fields.

Additionally, a helper field `refTypedFields'` of type **`set`** `Ref` is added to the signature and constrained to be the union of all fields of sort **`Ref`**. This relation will be used to encode a reachability constraint later (see Section 3.7.10).

Along with the `Ref` signature, we also declare `NULL` extending `Ref`. `NULL` can be used wherever a reference can, but it is constrained to have no fields at all.

```
sig Ref {
  val: lone Integer,
  refTypedFields': set Ref
} {
  refTypedFields' = none
}

one sig NULL extends Ref {}
fact { NULL.refTypedFields' = none && no NULL.val }
```

Listing 3.4: Encoding of the reference signature for the running example in Listing 3.3.

For our running example, the reference signature is encoded as shown in Listing 3.4. Since there are no reference-typed fields in the program, `refTypedFields'` is defined to be the empty set, **none**. The `NULL` signature is followed by a constraint that ensures it does not possess `val` field.

3.7.3 Modeling the Store

The store contains all the local variables accessible in a symbolic state. These may "point to" a concrete value, or to a term (usually a `VariableTerm`). For example, inspecting the symbolic state at the location marked in the following snippet

```
field val: Int
method store(a: Array)
  requires forall i: Int ::     0 <= i && i < len(a)
                            ==> acc(loc(a, i).val)
{
  var i: Int := 1
  var v: Int := loc(a, i).val
  // Encoding the state here
}
```

would result in a store containing both types of values

$\underline{Store}$

$a : Array \rightarrow a@1$

$i : Int \rightarrow 1$

$v : Int \rightarrow Lookup(val, sm@10(), loc(a@1, 1))$

Local variable a has symbolic value $a@1$, variable i holds the concrete value 1, and the value of variable v is a more complex term (in this case, a lookup of field *val* on the reference resulting from the call $loca(a@1)$ in the snapshot given by $sm@10$).

In the translated model we declare a special `Store` signature with multiplicity **one** that holds a field for each of the symbolic store's variables. In addition to that, we also declare a set called `refTypedVars'` to encode the union of all store variables with type **Ref**, as shown in the encoding for the running example in Listing 3.5. Like for the `refTypedFields'` set in the reference signature, this is used later in constraining references to be reachable from the store. We add an apostrophe to the name of all variables to prevent name clashes with built-in Alloy keywords (otherwise declarations such as `var some: Int` in Viper could become problematic).

```
one sig Store {
  nodes': one Set_Ref,
  n1': one Ref,
  n2': one Ref,
  v': one Integer,
  refTypedVars': set Ref
} {
  refTypedVars' = n1' + n2'
}

one sig nodes_3 in Set_Ref {}
fact { Store.nodes' = nodes_3 }
one sig n1_4 in Ref {}
fact { Store.n1' = n1_4 }
one sig n2_5 in Ref {}
fact { Store.n2' = n2_5 }
one sig v_11 in Integer {}
fact { Store.v' = v_11 }
```

Listing 3.5: Encoding of the store for the sunning example in Listing 3.3

For the running example in Listing 3.3, the encoding of the Store appears in Listing 3.5. First, we declare the `Store` signature with all the known variables, then we add a series of facts to constrain their values to the ones that we have in the symbolic store.

In building these constraining facts we may encounter new objects, which need to be declared. In this case, we find the symbolic values for all the variables, so we declare a new signature for each one of them before the fact they are used in (note that the order of declarations does not matter in Alloy, we just keep the declarations close together so they are easier to find).

Note that all the additional types (in this case `Set_Ref`) are declared later on in the model. Alloy does not put any restriction on the order of declarations: as long as all needed signatures are declared somewhere, everything is fine. Note also that the names of symbolic values are sanitized (by replacing the @ symbol with an underscore) before being used to define new signatures.

3.7.4 Modeling Heap Chunks and Permissions

The third section in the encoded model is where we declare heap chunks and permissions. Heap chunks are used to express information about a heap locations: which symbolic value(s) they have, how much permission we hold, what is the receiver (in case of a field chunk) or predicate (in case of a predicate chunk). From the heap in the symbolic execution trace, we know all chunks for which there might be at least some permission. We may have different types of heap chunks in the heap, such as field chunks, predicate chunks, quantified field chunks, etc. The translation of each heap chunk type is performed slightly differently.

Field Chunks

For field chunks we first add three main constraints. The first one relates the field of the receiver to the symbolic value we have in the heap, the second one encodes the permission held for that specific receiver, and the third one states that we do not hold more than full permission. As usual, encoding a fact might require declaring fresh names and introducing additional facts to constrain them.

```
field val: Int
method example(r: Ref)
  requires acc(r.val)
{
  // Encoding the state here
}
```

Listing 3.6: An example of a snippet with a simple field chunk.

Consider the snippet in Listing 3.6. In the heap have a field chunk `r@02.val: Int -> $t@03 # W`, where `r@02` is the symbolic receiver of the field access (reference `r` in the snippet), `$t@03` is the symbolic value of the field, and `# W` denotes the fact that we have full permission to this chunk. The resulting encoding is

```
one sig t_03 in Integer {}
fact { r_02.val = t_03 && PermFun.val[r_02] = W }
fact { perm_at_most[PermFun.val[r_02], W] }
```

Here, we constrain field `val` on reference `r_02` to be equal to `t_03`, a fresh name for an integer representing the symbolic value in the heap chunk. In addition to that, we encode the currently held permission via the relation `PermFun.val`. This is a freshly-declared relation named after the field and of type `Ref -> lone Perm`, which represents the permission held for that specific field and reference. Whenever we encounter a field chunk, we add such a function (unless one for that field existed already) to encode permissions.

We keep track of all references and fields for which we have some permission, so we can constrain the entries in the permission function to be only the references for which we do have some permission. Assume that we had permission to field `val` of symbolic references `r_01` and `r_02`, then we would encode the following constraint

```
fact { (PermFun.val).univ = r_01 + r_02 }
```

In which we state that the domain of the `PermFun.next` relation corresponds just to those two references.

Predicate Chunks

Encoding predicate chunks requires two additional signatures: one to represent the predicate itself, and one containing relations that can be used to constrain the permission to predicates. For example, consider the following snippet

```
method example(r: Ref)
  requires acc(wrap(r))
{
  // Encoding the state here
}
```

Access to predicate `wrap` results in the following heap chunk

```
wrap(SortWrapper($t@16, Snap); r@15) # W
```

where the first argument of `wrap`, before the semicolon, is a snapshot representing the predicate and all the other values after the semicolon are the arguments. The permission we hold is denoted by the expression after the hash symbol. In this case we have full permission.

The predicate chunk is encoded with the declarations in Listing 3.7. First, we define an abstract signature `pred_wrap` to represent instances of the predicate. Then, we introduce `Preds.wrap` to map the snapshot of the heap chunks to actual predicate instances. We want to to make sure that there are no instances other than the ones we explicitly add to this relation, therefore signature `pred_wrap` is constrained to be equal to the

```
 1 abstract sig pred_wrap {
 2   arg0: one Ref
 3 }
 4 one sig Preds {
 5   wrap: Snap -> lone pred_wrap
 6 }
 7 fact { pred_wrap = Preds.wrap[Snap] }
 8 one sig t_16 in Snap {}
 9 fact { one Preds.wrap[t_16] => Preds.wrap[t_16].arg0 = r_15 }
10 fact { PermFun.wrap[t_16] = W }
11
12 fact { (PermFun.wrap).univ = t_16 }
13 fact { all s: Snap |
14     perm_less[Z, PermFun.wrap[s]] <=> one Preds.wrap[s]
15 }
```

Listing 3.7: Encoding of a predicate chunk.

atoms contained in the `Preds.wrap` relation. The multiplicity **`lone`** makes this relation a partial function, which is required since not necessarily all snapshots should refer to a predicate.

After declaring the relation between `pred_wrap` and the tuples in `Preds.wrap`, on line 9 we add a fact constraining the arguments of the predicate to the ones we have from the heap chunk. Just after it, we encode the permission we hold to the chunk. The `PermFun.wrap` function is a freshly-declared permission function, which we will declare at the end of the model, as part of the gathered facts described in Section 3.7.8.

Finally, like we did for field chunks, we constrain the instances in the permission function to be the snapshots we know from the heap and state that a predicate instance exists if and only if we have at least some permission to it (on line 14).

Quantified Field Chunks

To encode quantified field chunks we have to declare multiple facts. A quantified field chunk expresses permissions to all fields specified by the original quantified permission assertions and no permission to all other locations. The heap for the program in Listing 3.3, *before* exhaling permissions (i.e. on line 7), would contain the quantified heap chunk

```
QA r :: r.val: FVF[Int] -> $t@7 # (inv@10(r) in nodes@3) ? W : Z
```

Which denotes full permission to field `val` for all references that are part of the set `nodes` in the snippet (when `inv@10(r) in nodes@3` is true) and no permission for all other references.

Silicon expresses the amount of permission held from the point of view the references on the heap, and introduces inverse functions on these references to refer back to the objects that were used in the specification. Here, you can see that Silicon is using the function `inv@10(r)` to relate them to the nodes we had in the specification. The idea behind this mechanism is explained in Schwerhoff's thesis [11, sec 4.2.1]. In this particular case, `inv@10(r) = r`, because we were quantifying over references in the original specification, but in general this is not always the case. Silicon always introduces two definitional axioms for the inverse functions. For our chunk, they are the following.

```
QA r@9 :: r@9 in nodes@3 ==> inv@10(r@9) == r@9
QA r   :: inv@10(r) in nodes@3 ==> inv@10(r) == r
```

The first one relates quantified variables from the source quantifier to the result of the inverse function, and the second does the same, but from the perspective of the domain of the inverse function.

In the chunk we also see that the symbolic value is not just an **`Int`**, but a `FVF[`**`Int`**`]`. This is because the quantified field chunk represent access to a potentially unbounded number of heap locations, so the value information is represented in a snapshot map (`$t@7` in this case) that maps any receiver in the domain of the heap chunk to the value of his field. Snapshot maps are used when expressing field lookups into quantified chunks, as discussed in Section 3.1.9.

In the heap that we have from the running example in Listing 3.3, at the location of the verification failure, we have the following quantified field chunk

```
QA r :: r.val: FVF[Int] -> $t@7
    # (inv@10(r) in nodes@3) ? W : Z - pTaken@12(r)
```

Here you can see that the permission expression is more complex, involving a `pTaken@12` function. Whenever we exhale permissions from an object involved in a quantified field chunk, like in the example, Silicon introduces `pTaken` functions to denote the amount of permission removed for a specific reference. In this case, the newly added `pTaken` function makes sure that the permission value of the chunk evaluates to `W` for all references in the set, except for `n1`, for which the permission value should be zero. The definition of `pTaken@12` is shown in the following snippet.

```
pTaken@12(r) :=
  (r == n1@4) ? PermMin( (inv@10(r) in nodes@3) ? W : Z, W )
              : Z
```

The function evaluates to `W` only for reference `n1@4`. The `PermMin` operation computes the minimum between the permission we hold in the chunk this function was defined for and the permission we are trying to take away (both `W` in this example), to ensure that we don't take away more pemission than we actually hold. In case we had a fragmented heap it would be possible to take away the permission we are exhaling from multiple chunks. The translation of these functions is described in Section 3.7.7.

The result of encoding the heap chunk from the running example is shown in Listing 3.8. The encoding proceeds as follows. First, we translate the axioms for the inverse functions introduced by Silicon. Each one of them is a term, so we follow the procedure described in Section 3.1. After that, we encode the chunk itself, by introducing a `PermFun.val` relation, that maps references and snapshot maps to permission amounts. We constrain `PermFun.val[r, t_7]` to be equal to the translation of the permission amount from the quantified chunk. In this case, we need to introduce a signature to hold fresh names because we are inside a quantification, as explained in Section 3.1.7.

```
 1 fact { all r_9: Ref |
 2   set_in[r_9, nodes_3] => (Fun.inv_10[r_9] = r_9) }
 3 fact { all r: Ref |
 4   set_in[Fun.inv_10[r], nodes_3] => (Fun.inv_10[r] = r) }
 5
 6 one sig t_7 in FVF_Integer {}
 7 one sig fresh_quantifier_vars_0 {
 8   temp_1': Ref -> lone Perm
 9 }
10 fact { all r: Ref |
11     perm_minus[ (set_in[Fun.inv_10[r], nodes_3] => W else Z),
12                 PTAKEN.pTaken_12[r],
13                 fresh_quantifier_vars_0.temp_1'[r] ] &&
14     fresh_quantifier_vars_0.temp_1'[r] = PermFun.val[r, t_7]
15 }
16 fact {
17   all r: Ref |
18       ( some fvf: (t_7) |
19             one PermFun.val[r, fvf] and
20             perm_less[Z, PermFun.val[r, fvf]] )
21       <=> (one r.val)
22 }
23 fact {
24   all r: Ref, fvf: (t_7) |
25       one PermFun.val[r, fvf] =>
26         (perm_at_most[PermFun.val[r, fvf], W])
27 }
```

Listing 3.8: Encoding of the heap chunks and permissions for the running example in Listing 3.3

After having encoded the permission for the chunk, we add two more facts. The first one, starting on line 16, constrains field `val` on any reference to exist only if there exists a snapshot map, amongst the ones we know from the heap, for which we do have permission in the quantified heap chunk. The second fact, starting on line 23, constrains the permission for the chunk to be at most **write**.

Mixing Chunks and Other Types of Chunks

Of course, in general it is possible to require permissions to the same fields by mixing both quantified field chunks and normal fields chunks, or to require permission in "different steps" (e.g. `acc(r.val, p1) && acc(r.val, p2)`). In these cases, Silicon performs a simplification of the chunks for us. For example, in the snippet

```
requires forall r: Ref :: r in nodes ==> acc(r.val)
requires n1 in nodes
requires acc(n2.val)
```

we would get two quantified heap chunks

```
QA r :: r.val: FVF[Int] -> $t@69 # (inv@72(r) in nodes@65) ? W : Z
QA r :: r.val: FVF[Int] -> sm@74() # (r == n2@67) ? W : Z
```

This makes it easier to work with them as we can translate both field chunks independently and predicate the existence of the field on reference objects by considering all snapshot maps:

```
fact {
  all r: Ref |
      ( some fvf: (t_69 + sm_74) |
          one PermFun.val[r, fvf] and
          perm_less[Z, PermFun.val[r, fvf]] )
      <=> (one r.val)
}
```

There are other types of heap chunks: quantified predicates and magic wands, which are not covered as part of the current translation technique. A discussion about them can be found in Section 7.1.

3.7.5 Modeling Path Conditions

Each symbolic state contains a set of path conditions, which represent the facts known to be true at that point in the verification of the program. Path conditions in the symbolic execution trace are simply Silicon terms of sort Boolean and are translated directly into Alloy facts, one fact per path condition.

The translation of terms is described in Section 3.1. Like for the encoding other parts of the program, it is sometimes necessary to declare fresh names for objects (for example results of operations), and to insert additional facts to constrain these fresh values. For a path condition PC, the declaration resulting from its translation looks roughly like this:

$$
\begin{array}{l}
freshName_1 \\
\qquad \vdots \\
freshName_n \\
\textbf{fact \{ } addFact_1 \textbf{ \&\& } \dots \textbf{ \&\& } addFact_n \textbf{ \&\& } \mathsf{translate}(\mathsf{LogicalWrapper}(PC)) \textbf{ \}}
\end{array}
$$

The actual path condition PC is wrapped in a `LogicalWrapper` before being translated in order to ensure that the final type of the translated value is `PrimitiveBoolean`. The need for this wrapper is discussed in Section 3.5. Note that the additional facts, $addFact_i$ could very well be declared at the top level, as stand-alone facts, but in the current implementation we keep them in the same fact they were generated for. Declaring them at the top level is certainly something to consider in future updates of this technique.

As an example, consider inspecting symbolic state 1 in the following snippet, just inside the outmost if statement.

```
method example(s1: Set[Ref], s2: Set[Ref])
{
  if (|s1| > 0) {
    // Symbolic state (1)
    if (|s1 union s2| > 0) {
      // Symbolic state (2)
    }
  }
}
```

If we reach that location, we know that the cardinality of the set has to be greater than zero, resulting in a single path condition: $SetCardinality(s1@2) > 0$. This path condition would be translated into the following Alloy facts

```
one sig temp_0' in Integer {}
fact { set_cardinality[s1_10_01, temp_0'] }
one sig temp_1' in Integer {}
fact { temp_1'.value = 0 }
fact { (temp_0'.value > temp_1'.value) }
```

The inspection of symbolic state 2 inside the next if statement gives us an additional path condition: $SetCardinality(s1@2 \cup s2@3) > 0$. This new condition requires a fresh instance to represent the union, as well as fresh instances to represent all of the integers, like in the previous snippet. The constraint is translated into the following declarations.

```
one sig temp_1' in Set_Ref {}
fact { set_union[s1_10_01, s2_11_01, temp_1'] }
one sig temp_0' in Integer {}
fact { set_cardinality[temp_1', temp_0'] }
one sig temp_2' in Integer {}
fact { temp_2'.value = 0 }
fact { (temp_0'.value > temp_2'.value) }
```

In both examples, since the resulting operation already had `PrimitiveBoolean` type, the additional `LogicalWrapper` did not affect the translation.

3.7.6 Modeling Domain Axioms

Viper domains allow to define custom types and their behaviour. To ensure that counterexamples involving user types are modeled correctly, all the axioms defined in a domain need to be encoded in the Alloy model as well.

Domain axioms are Silicon terms, so they are translated following the procedure described in Section 3.1 and declared as top level facts in the model.

As an example, translating the length axiom from the following (shortened) domain definition for arrays

```
domain Array {
  // ...
  function len(a: Array): Int
  axiom lenPositive {
    forall a: Array :: len(a) >= 0
  }
}
```

gives us the following fact:

```
fact { all a: Array | (Fun.len[a] >= 0) }
```

3.7.7 Ptaken Functions

As explained in Section 3.7.4, Silicon introduces pTaken functions when permissions are removed from quantified heap chunks. These are functions with a parameter of type **`Ref`**, and a result of type **`Perm`**, corresponding to the amount being removed.

In the symbolic execution trace, the pTaken functions are represented as pairs of Silicon terms of the form $(Application(name, r : Ref, Perm), body)$. Each pTaken function is encoded as a relation of type `Ref -> one Perm` in the `PTAKEN` signature, with an additional fact to restrict the elements in the relation to respect the logic of the function's body.

In case Silicon had generated n such functions, we would get a series of declarations like

```
one sig PTAKEN {
  $pTaken_1$: (Ref -> one Perm)
      ⋮              ⋮
  $pTaken_n$: (Ref -> one Perm)
}
fact { all r: Ref | translate($pTaken_1(r)$ == $body_1$) }
  ⋮          ⋮                ⋮              ⋮
fact { all r: Ref | translate($pTaken_n(r)$ == $body_n$) }
```

In our running example from Listing 3.3, we exhale all permissions to field `val` on reference `n1` and, since the permissions were given in a quantified field chunk, Silicon introduces a pTaken function to encode the fact that we have no permissions on `n1` anymore. The pTaken function has the following definition:

```
pTaken@12(r) :=
    (r == n1@4)
        ? PermMin( (inv@10(r) in nodes@3) ? W : Z, W )
        : Z
```

And is encoded into the following Alloy declarations

```
one sig PTAKEN {
  pTaken_12: Ref -> one Perm
}
one sig fresh_quantifier_vars_1 {
  temp_2': Ref -> lone Perm
}
fact { all r: Ref |
  perm_min[ set_in[Fun.inv_10[r], nodes_3] implies W else Z, W,
            fresh_quantifier_vars_1.temp_2'[r] ] &&
  perm_equals[
    PTAKEN.pTaken_12[r],
    (r = n1_4) => fresh_quantifier_vars_1.temp_2'[r] else Z
  ]
}
```

The first signature contains the declaration for the relation that represents the pTaken function, and the last fact constrains the elements in the relation to be valid according to the original body. The other fact and the additional signature are needed to declare fresh names inside quantifier, as explained in Section 3.1.7.

3.7.8 Facts Gathered During the Translation

While translating the program, we may encounter some objects that need to be declared. These are usually functions and sorts. In both cases they may come from user declarations, from terms in the program, or they may even have been introduced in the translation itself.

Permission Functions

First, we declare all the permission functions that we introduced when encoding the heap. All functions are declared as relations in the `PermFun` signature, their type depends on the type of heap chunks for which they have been defined. For example, if we had a field chunk for field `val`, then we would get the following declaration, with a permission function of type `Ref -> lone Perm`.

```
one sig PermFun {
  val: (Ref -> lone Perm)
}
```

If we instead had a quantified field chunk for an integer field `val`, then the declaration would also be parametrised by the snapshop map (of type `FVF[Int]`). For the running example in Listing 3.3 we would get a declaration like this one.

```
one sig PermFun {
  val: (Ref -> FVF_Integer -> lone Perm)
}
```

For predicates, instead, the permission function has an instance type `Snap` as its first argument, so if we had a predicate called `list`, its permission function would look like in the following snippet.

```
one sig PermFun {
  list: (Snap -> lone Perm)
}
```

Functions

After permission functions, we encode regular functions (meaning functions from the original program, not introduced during the translation) and their postcondition axioms.

Functions are declared as n-ary relations in the `Fun` signature. For example, A function with two parameters of type **`Int`** and a return type of **`Ref`** would be translated into a relation of type `(Snap -> Integer -> Integer -> Ref)`, where the additional parameter of type `Snap` is added by Silicon during its axiomatization. A simple function with postconditions, like in this snippet

```
function f(i: Int): Int
  ensures result == i

method test1() {
  var i: Int := f(0)
  assert i == 0
}
```

is translated to the following two declarations, where the additional fact encodes the postcondition of the function. Postcondition axioms always have the form of quantifiers over the argument types of the function and contain a let expression as their body. They are translated following the normal term translation procedure described in Section 3.1.

```
one sig Fun {
  f: (Snap -> Integer -> one Integer)
}
fact {
  all s: Snap, i_16: Integer |
      let result_17 = Fun.f[s, i_16] | (result_17 = i_16)
}
```

In the case of a heap-dependent function the axiom declaration would be slightly different. Consider the following heap-dependent function:

```
field val: Int
field next: Ref

function hdf(r: Ref)
  requires acc(r.val) && acc(r.next)
  ensures result == r.val
```

The postcondition of the function will be defined with respect to its snapshot, as follows:

```
QA s, r@2 :: let result = hdf2(s, r@2) in
      (result == SortWrapper(First(s), Int))
```

Where snapshot *s* is expected to always be a pair (a *Combine*) and `First` denotes the extraction of the first element. As explained in Section 3.1.6, this can be encoded without additional functions in the model, by using the `wrapped` relation in the SortWrapper and the `left` relation on signature `Combine`.

The small difference in the encoding of this postcondition is that we have to introduce an additional "guard": in our Alloy encoding only `Combine` instances have the `left` and `right` relations, therefore considering all snapshots (which include, for example, Unit) would falsify the quantifier's body. We introduce the check `'s in Combine'` just inside the body of the quantifier, so that the result is constrained only for snapshot pairs.

The function and post-condition are finally encoded as:

```
one sig Fun {
  hdf: (Snap -> Integer -> Ref -> one Integer)
}
fact {
  all s: Snap, r_2: Ref | s in Combine =>
      let result_3 = Fun.hdf[s, r_2] |
          result_3.value = s.left.wrapped.value
}
```

Lookups

Lookup functions are used in Silicon to encode field lookups in quantified field chunks. Just like regular functions, lookups are encoded into Alloy as relations, but they are always ternary, with the first element being the snapshot map of the chunk, the second one the reference for which we are accessing the field, and the third for the value of the field. A lookup function for field `val` would be encoded into a relation with the same name, declared in the `Lookup` signature. Along with them, we add a fact to constrain their result to be equal to the field on the actual `Ref` object. In our running example, the field lookup function would be encoded as

```
one sig Lookup {
  val: (FVF_Integer -> Ref -> lone Integer)
}
fact { all fvf: FVF_Integer, r: Ref |
  (one PermFun.val[r, fvf] and perm_less[Z, PermFun.val[r, fvf]])
        => (Lookup.val[fvf, r] = r.val)
}
```

Sorts

Lastly, in the "gathered facts section", we encode user-declared sorts and other additional sorts that we discovered during the translation. A user sort such as `Array` would be simply translated into a signature `sig User_Array' {}`, where the name is altered to avoid collisions with other signature declarations. Amongst the "other sorts" we have to declare we have the dynamic signatures generated for the Viper collections. In the encoding of our running example we would have the following declarations for Viper's `Set[Ref]` and the field value function used in lookups.

```
sig Set_Ref extends Set {} {
  set_elems in Ref
}
sig FVF_Integer {}
```

3.7.9 Failed SMT Query

Whenever verification fails because the SMT solver is not able to disprove a fact, this failed assertion is reported to us in the symbolic execution trace, in the verifiable (method, predicate or function) in which it occurred. This failed assertion encodes an important property that distinguishes counterexamples to the verification from other valid but non-failing instances of the program. This is discussed in more detail in Chapter 2.

The failed SMT query is a normal Silicon term and it follows the same translation procedure as path conditions. The only additional step that we take before translating the assertion is to negate it.

```
freshNames...
fact { addFacts... && translate(LogicalWrapper(¬failedAssertion)) }
```

Since Silicon failed to prove that this fact always holds, which lead to a verification failure. We are interested precisely in modeling these problematic situations, where the assertion is falsified, hence why we negate the term before encoding it.

3.7.10 Reachability and Signature Restrictions

When generating instances, we want to make sure that only meaningful references are generated and that there are no “stray” ones, not being used in the rest of the program. To achieve this, we add a special constraint for references, specifying that they are in the store, that they are reachable from it (via fields of other references), or that they belong to some collection (sets, sequences, multisets, but also sort-wrappers).

The constraint is defined in the following fact, stating that signature `Ref` corresponds to the union of all the other relations, all resulting in some set of references. Here we can see the `refTypedVars'` and `refTypedFields` relations that were defined in the encoding of the store and of the `Ref` relation.

```
fact { Ref = Store.refTypedVars'.*refTypedFields' +
       NULL +
       (SortWrapper.wrapped <: Ref) +
       RefTypedSet.set_elems +
       RefTypedSeq.seq_rel[Integer] +
       RefTypedMultiset.ms_elems +
       RefTypedFunctions[SortSignatures] }
```

The last four members of this union may not be present or may represent multiple relations. For example, in case our program uses both **Set**[**Ref**] and **Set**[**Set**[**Ref**] variables, then the `Set_Ref.set_elems` and `Set_Set_Ref.set_elems` relations would appear in the union.

The last member of the union represents function applications with a result of type **Ref**. Here the relation representing a reference-typed function would be partially applied with the signatures corresponding to the sort of its arguments. As an example, if a Viper program uses a function with the signature `function f1(i: Int, b: Bool): Ref`, then in the constraint we would have an element `Fun.f1[Integer, Bool]`.

Note that Alloy does not have the concept of function application. When we write something like `Fun.f1[Integer, Bool]` we are defining a join between a relation `f1` of type `f1: Integer -> Bool -> lone Ref` and the elements in the two signatures `Integer` and `Bool`. This practically corresponds to all possible combination of arguments which could be used to call the function, therefore the expression represents all possible results of the function.

We also add a constraint for all other signatures that represent a Viper type. This is done to prevent Alloy from generating instances of signature which are not actually used in the program. During the translation we keep track of all variables we encounter and of their sort (including those that we introduced ourselves), so that at the end we can restrict the sort signatures to contain exactly the variables we have recorded. For the running example in Listing 3.3, the signature restrictions are the following.

```
fact { Set = Store.nodes' }
fact { FVF_Integer = t_7_01 + sm_13_01 }
fact { Snap = t_6_01 + temp_0' + temp_1' + t_8_01 +
       temp_2' + Unit }
fact { Perm = PermFun.val[Ref, FVF_Integer] + temp_3' +
       temp_4' + W + Z }
fact { Seq = none }
fact { Multiset = none }
```

We can clearly see that we have both symbolic values from the original program and temporary variables that were declared to hold results of operations. Moreover, since there are no sequences or multisets in the example, their signatures are constrained to the empty set (**none**).

4 Architecture

In this section we discuss the architecture of the Viper IDE extension at the time the project started, and how we changed it in order to both support the features of the new debugger and to allow extending it in the future.

4.1 The Existing Infrastructure

The Viper IDE is implemented as an extension for the Visual Studio Code editor. Architecturally, it comprises three main components: the language client, the language server, and the debugger adapter. Additionally, the language server starts and communicates with *ViperServer*, an external component that manages the instantiation and execution of the verification backends. ViperServer and the verification backends are downloaded and setup by the extension automatically, but are not part of it.

The fist components, the language client and language server, are decoupled from each other and communicate via the *Viper Protocol* and the *Language Server Protocol*[1]. They implement the two parts needed by Visual Studio Code for a language server to work. On the other hand, the code that constitutes the debugger exists both in the client and the server components and is tightly coupled with them.

There are two problems with the modularity this design:

- Components that have orthogonal responsibilities are coupled together and depend on each other to function.
- It is not possible to *only* install the components providing integration with the verification backends (language client and server), but one must also install the debugger, though it might not be needed.

If we envision integrating the verification environment with new tools in the future, then this type of architecture is not ideal. Moreover, both the old debugger and the new one only target the Silicon backend, therefore the design of the infrastructure should

[1] `https://code.visualstudio.com/docs/extensions/example-language-server#_a-brief-overview-of-language-server-and-language-server-protocol`

accommodate the possibility of having multiple debugger extensions installed at the same time, one for each of the verification backends. Moreover, the architecture should allow not making the debugger an explicit dependency of tools that have nothing to do with symbolic execution (e.g. the Carbon backend).

Another thing to note is that the infrastructure of the main extension has been partially reworked after the completion of Kälin's thesis, but the code for the debugger has not been kept up to date, so the original debugging features are not active in the current version of the extension.

In addition to that, the debugger also directly depends on external tools, in order to retrieve the symbolic execution trace. When silicon is run with the `--ideModeAdvanced` flag, it writes the symbolic execution trace to a file in a temporary location. The debugger then reads the trace from this file. Retrieving the symbolic execution trace this way is less than ideal, because of I/O performance reasons. This problem can be solved easily with the new architecture, where the backends and the IDE communicate via message passing through ViperServer.

4.2 The New Infrastructure

The most important architectural change, with respect to the implementation of the previous debugger, is that the new one "lives" in a completely separate extension. This design choice was made to avoid forcing users to install the debugger extension if they have no need for its features, especially since the debugger only targets the Silicon backend and because it is currently in an experimental state. We will refer to the existing extension (which integrates VS Code with the verification backends) as the *main extension*, and to the new one as the *debugger extension*.

The debugger depends on the main extension to work correctly, therefore lists it as an explicit dependency in its extension manifest file. Visual Studio Code makes sure that this dependency is satisfied by preventing the debugger to be installed if the main extension is not present.

Since the debugger is separate from the main extension (but still requires information about the verification process in order to function), we had to update the main extension to support communication with other extensions. In addition to that, the debugger also needs to interact with a panel in the user interface, which is also a completely separate component. In the following sections, we will discuss how these separate parts of the system communicate with each other.

A diagram of the infrastructure outlining the main components and the communication channels between them can be seen in Figure 4.1.

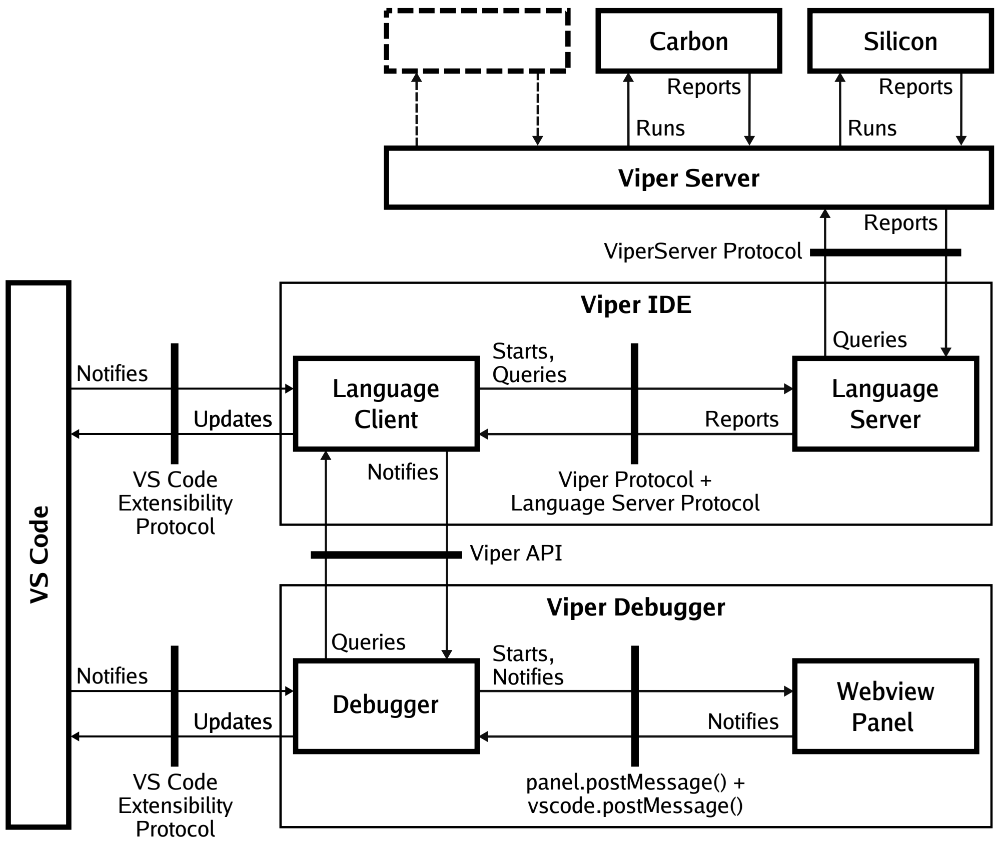


Figure 4.1: The architecture of the system. The *Viper IDE* and *Viper Debugger* boxes denote, respectively, the main Viper extension and the new debugger extension. They both interact, independently, with Visual Studio Code. Viper Server is responsible for running the verification backends.

4.2.1 Communication with the Main Extension

Visual Studio Code extensions can return an object from their `activate`[2] method (the main entry point of each extension). This object can be retrieved and used by other extensions at runtime. The main Viper extension has been updated to export a `ViperApi` object that allows other extensions (in this case the debugger) to interact with the verification process. Though currently this API only provides functionalities specifically needed by the debugger, this approach allows it to be easily extended to support other extensions that can interact with Viper, without having to drastically modify the implementation of the main extension.

The Viper API provides some utility methods that allow the debugger to access the last file that was verified or to retrieve the state of the backend. In addition to that,

[2]https://code.visualstudio.com/docs/extensions/example-hello-world#_generated-code

it also allows other extensions to register a notification handler that is called when a verification terminates or when ViperServer sends messages that the main extension does not know how to handle.

When the Silicon backend is configured with the `--ideModeAdvanced` flag (more details in appendix Appendix A), it will produce additional messages for the debugger, that are delivered to the main extension via ViperServer. The main extension does not know how to handle these additional messages, therefore it will notify all registered handlers for them (if there are any). When the debugger is first started, it retrieves the `ViperApi` object and registers a handler for verification completion events and for additional server messages. Thus, it can receive messages containing debugging information from ViperServer via the main extension, and be notified with verification completion events.

4.2.2 ViperServer and Silicon

The old debugger, as mentioned before, retrieved the symbolic execution trace from a file. ViperServer (along with some other parts of the reporting infrastructure) has been updated to serialize the Symbolic Execution Log to JSON, via the *Spray JSON*[3]library (which was already used to for the rest of the communication with the backends). The trace is now streamed via HTTP by ViperServer, and finally gets to the Debugger after passing through the main extension and the ViperApi.

The Symbolic execution log has also been extended to contain more information, needed by the debugger to operate. These additional pieces of information are the postcondition axioms declared by Silicon for the functions in the program being verified, the `pTaken` macros generated when permissions are removed from heap chunks, and the last query that was performed to the SMT solver before the verification failed (in case the SMT solver was not able to disprove an assertion).

4.2.3 The Webview Panel

The existing debugger provided a side-panel to display the debugger information. The panel was implemented using the *HTML preview* feature of the editor. This feature was originally intended to simply display static HTML documents in the editor, but soon started being used by extension developers to implement complex interfaces, mainly due to flexibility and ease of use of HTML. The HTML preview was not intended for these type of use cases, consequently this approach had its downsides, the two most important ones being the fact that only static contents could be displayed (requiring a full refresh for each update) and that there was no way for the panel to interact with the rest of the editor. When the first debugger was built, this was the only way of extending Visual Studio Code's interface.

[3]`https://github.com/spray/spray-json`

Around the time this project started, the *Webview API*[4] was being developed by Microsoft as a successor for the HTML preview panel, with the aim of providing a way for extension developers to extend the editor's interface, similar to what they were doing with the HTML preview, but with an API designed from the beginning to do so. The main feature of this new API is a way for the panel to communicate back with extensions, making it possible to build interactive interfaces. The Webview API was introduced in the April 2018 release of VS Code, and we decided to use it for implementing the main panel of our debugger.

Note that, whereas the original debugger used VS Code's native debugger API to interface with the editor, we decided not to use it, as it would have provided little benefits. The API was conceived to support more traditional debuggers, with features that are not relevant for our project.

The debugger panel is a *NodeJS*[5] module separate from the rest of the extension. When the debugger extension needs to create a Webview panel for the debugger, it points to the 'debugger.html' file (implementing the interface and logic of the panel) which is run as a stand-alone process.

The debugger extension and the panel communicate by message-passing through the `Webview.postMessage` method (for messages from the debugger extension to the panel) and the `VSCode.postMessage` method (for messages from the panel to the extension). These are simply wrappers around the `window.postMessage()` method used to enable cross-origin communication between window objects in browsers.[6]The debugger extension sends messages to the panel to notify a focus change on the selected verification state, to provide a new heap graph to be displayed, or to provide various diagnostic messages. The panel sends messages to the debugger extension only when it needs to notify that a new verification state has been selected.

[4]https://code.visualstudio.com/docs/extensions/webview

[5]`https://nodejs.org/`

[6]`https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`

5 The Debugger

The debugger can be started via the command palette (which can be opened by typing `CTRL-p` in Visual Studio Code). Only two new commands are provided: *Start Debugger* and *Stop Debugger*. As soon as the debugger is started, it will trigger a new verification of the currently selected Viper file (unless a verification is running already), so that the debug information can be gathered. When the verification terminates and all of the debugging information is received, the debugger panel is populated. When the panel is closed or the *Stop Debugger* command is invoked, all resources acquired by the debugger are disposed and the debugging session terminates.

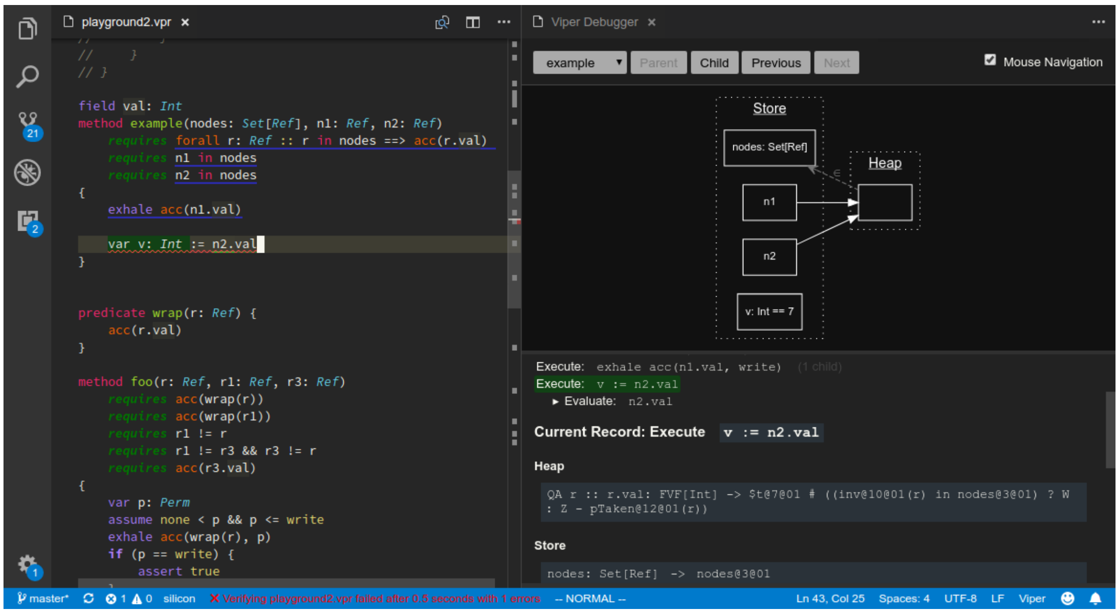
For more information about the debugger's configuration options, please refer to Appendix A.

5.1 The Panel

The debugger panel is divided into three sections. When it is started, only the topmost two sections are visible, and the third one is collapsed. The sections, top to bottom, present, respectively: the heap visualization graph, the symbolic execution information for the currently selected state, and additional diagnostics information. The three sections can be resized and even collapsed completely.

In addition to that, when the debugger is running, parts of the code are highlighted to represent the position of symbolic execution states in the program. The currently selected symbolic state is highlighted with a green background on the code. Children states are highlighted with a light green underline. Siblings of the current state (predecessors or successors in the symbolic execution order) are highlighted with a blue underline. Top-level states, when not currently selected or not highlighted as siblings of the current state, are shown with a dashed grey underline. Navigation through the symbolic states is possible by clicking on the parts of code highlighted in the editor. When multiple states correspond to the same location (for example, a variable declaration and its assignment could have the same location in the code, but are represented by distinct symbolic states), clicking on them will show a pop-up dialog that allows choosing which state to inspect. The colors used to highlight states can be configured with the settings described in Appendix A.

A screenshot of a running debugging session is shown in Figure 5.1.



```
//      }
// }

field val: Int
method example(nodes: Set[Ref], n1: Ref, n2: Ref)
    requires forall r: Ref :: r in nodes ==> acc(r.val)
    requires n1 in nodes
    requires n2 in nodes
{
    exhale acc(n1.val)

    var v: Int := n2.val
}


predicate wrap(r: Ref) {
    acc(r.val)
}

method foo(r: Ref, r1: Ref, r3: Ref)
    requires acc(wrap(r))
    requires acc(wrap(r1))
    requires r1 != r
    requires r1 != r3 && r3 != r
    requires acc(r3.val)
{
    var p: Perm
    assume none < p && p <= write
    exhale acc(wrap(r), p)
    if (p == write) {
        assert true
```

```
✗ Verifying playground2.vpr failed after 0.5 seconds with 1 errors
```

```
Execute:  exhale acc(n1.val, write)  (1 child)
Execute:  v := n2.val
  ▸ Evaluate:  n2.val

Current Record: Execute  v := n2.val

Heap
  QA r :: r.val: FVF[Int] -> $t@7@01 # ((inv@10@01(r) in nodes@3@01) ? W
  : Z - pTaken@12@01(r))

Store
  nodes: Set[Ref] -> nodes@3@01
```

Figure 5.1: A running debugging session. The code being debugged is shown on the left-hand side. Inspectable verification states are highlighted on the source with a blue underline. The currently selected verification state is highlighted with a green background.

On the right-hand side, the debugger panel is visible. In the top half we can see the heap graph visualization and the navigation controls at the very top. In the bottom half we can partially see the information from the symbolic execution state.

5.1.1 Navigation Bar and Heap Visualization

The first section in the panel displays the navigation bar and the heap graph visualization right below it.

The navigation bar contains a drop-down menu that allows to chose the *verifiable* (predicate, function, or method) to inspect. Next to the drop-down menu are four buttons that allow navigating the states in the symbolic execution trace. Finally, on the very right of the navigation bar, the *Mouse navigation* checkbox allows to toggle navigation by clicking on the code directly. Disabling the click event in the editor can be useful to allow editing code without having to stop a debugging session.

The top-most section of the panel displays the graph generated from the current Alloy instance. The graph can be moved in the viewport by clicking and dragging. It is possible to zoom in and out on the visualization by using the scroll-wheel. Whenever a new state is selected, the graph is updated to reflect the new information.

The visualization in this section of the panel is built with the information from the symbolic execution trace and the instance generated from the Alloy model. First, all the atoms that are relevant for displaying the graph are extracted from the Alloy instance. These atoms are those in the signatures for primitive types, collections and functions. Then, the visualization is built starting from the information about the store provided in the symbolic execution trace, and trying to map the variables known in the symbolic state to the concrete values generated by Alloy. For each atom in the reference signature, its fields are populated according to the values in the Alloy instance.

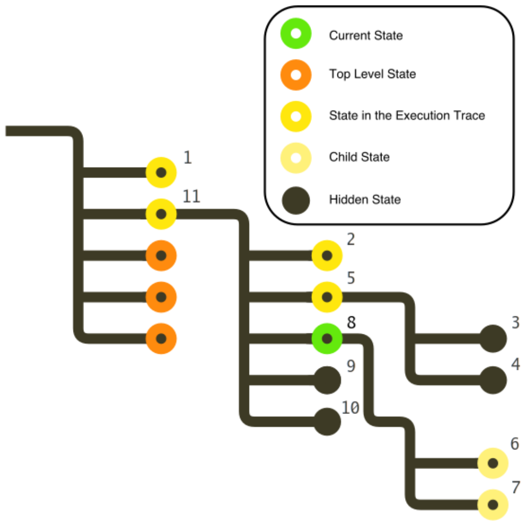
5.1.2 Symbolic State Information

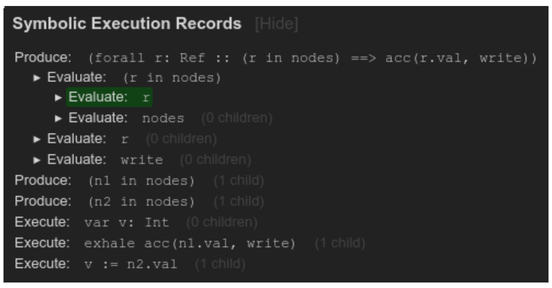
The execution trace section provides information about the pre-state of the currently selected program point and allows navigating through all the available states.

At the top, we have the partial execution tree. This is a list of all the top-level symbolic execution actions performed by Silicon during the verification of the currently selected verifiable. Each entry corresponds to either one of the basic symbolic execution actions (produce, consume, evaluate, and execute)[12] or to other special types of entries that appear in the symbolic execution trace (e.g. a global branch that splits the execution inside a produce action). Each entry also shows the formula on which the action was performed and the number of sub-entries. Clicking on any of the entries instructs the debugger to show the symbolic state at the moment when the action was about to be performed. What we see is always the pre-state of the currently selected action, i.e. the symbolic state of the program on which the action was applied. Selecting an entry will show its children (if there are any). The layout of the visualised information follows the guidelines described by Kälin in his thesis [4, sec. 4.2].

Below that, the panel shows information about the symbolic pre-state of the currently selected action. A symbolic execution state consists of a store, containing all local variables, a heap, and an old heap, which describe the locations to which we have permission, and a set of path conditions, which are facts known to be true about the objects in the program in the current state[12].

The panel first lists all the heap chunks for which some permission is held, after that it shows the store, containing all the local variables, and finally it displays a list of the path conditions that have been collected during the verification of the program, in the current execution trace. This information is part of what is used to encode a model of the current state into the Alloy language (as described in sec. Chapter 3).





Symbolic Execution Records [Hide]

```
Produce:  (forall r: Ref :: (r in nodes) ==> acc(r.val, write))
  ▸ Evaluate:  (r in nodes)
    ▸ Evaluate:  r
    ▸ Evaluate:  nodes  (0 children)
  ▸ Evaluate:  r  (0 children)
  ▸ Evaluate:  write  (0 children)
Produce:  (n1 in nodes)  (1 child)
Produce:  (n2 in nodes)  (1 child)
Execute:  var v: Int  (0 children)
Execute:  exhale acc(n1.val, write)  (1 child)
Execute:  v := n2.val  (1 child)
```

Figure 5.2: A comparison between the schema of the navigation in the execution trace discussed by Kälin (picture from his thesis [4, sec. 4.2]), and the navigation section in the panel of the new debugger. The numbers in the diagram indicate the order in which the actions are completed.

5.1.3 Diagnostics

The diagnostics section is found at the bottom of the panel, but is collapsed by default. It contains additional information that might be useful to investigate problems with the debugger itself, or to inspect the intermediate results of the heap graph generation.

The section provides the following information:

- The raw Symbolic Execution Log received from ViperServer, containing the symbolic state information about the last verification. The log is refreshed with each new verification run.
- The Alloy model generated from the currently selected verification state, as well as a button that allows to copy it into the clipboard, so that it may be run in the Alloy analyser directly.
- The raw message from ViperServer containing the Alloy instance that was generated from the model (if an instance was generated at all).
- The DOT graph encoding the information displayed in the heap visualization.

6 Evaluation

6.1 A Case Study of the Modeling Technique

6.1.1 Showing Counterexamples

Through the use of our proof of concept implementation, we can already identify instances where the debugger allows visualizing counterexamples that highlight a problem in the input program (either in the implementation or in the specification).

Imagine the user is trying to define a graph structure via quantified permissions. She might write the following specification, but inadvertently introduce a bug by mistyping the right-hand side of the implication that was supposed to ensure holding permission to all reachable nodes. Such a specification is presented in Listing 6.1.

```
1 method remove(ns: Set[Ref], x: Ref)
2   requires forall n: Ref :: n in ns ==> acc(n.next)
3   requires forall n: Ref ::
4              n in ns && n.next != null ==> n in ns
5   requires x in ns
6 {
7   // Encode state here
8 }
```

Listing 6.1: A wrongly specified graph, where references reachable via the next field are not necessarily part of the set defining the structure, because of the typo highlighted on line 4.

The visualizer will come up with an example highlighting the problem, like the one in Figure 6.1. Here, we see that there are indeed 3 references in set ns, but there is an additional reference, pointed to by one of the other objects, which has no fields and does not belong to the set. This clearly shows that there is something wrong with our specification and prompts us to check it. In doing that, we realize that in the right-hand side of the implication, in the second precondition, we should have specified `n.next in ns`. Fixing the precondition produces the structures we expect. Note that, in this situation, we were not inspecting a program where a verification failure occurred (though the specification problem might have lead to such a problem later on), but we were simply visualizing one of the instances of the current symbolic state. This highlights the advantage of being able to model our program even when there is no failure, as it could still lead to the discovery of problems with the specification or the implementation.

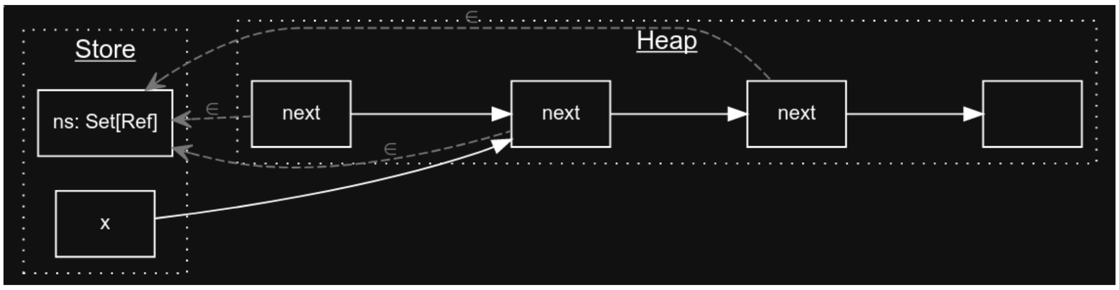


Figure 6.1: An example of the visualization of the wrongly specified graph from Listing 6.1. We expect all reachable nodes to be part of set ns, but the last node does not satisfy this property.

For an example with an actual verification failure, consider the method in Listing 6.2. A user might be convinced, because of the fact that no node in the set nodes is unlinked, that the elements form a ring and each node has to have a predecessor.

```
method ring(nodes: Set[Ref])
  requires forall r: Ref :: r in nodes ==> acc(r.next)
  requires forall r: Ref :: r in nodes ==> r.next != null &&
                                           r.next in nodes &&
                                           r.next != r
{
  var n: Ref;
  assume n in nodes

  var hasPred: Bool := (exists r: Ref ::
                            r in nodes && r.next == n)
  assert hasPred  // Verification fails
  ...
}
```

Listing 6.2: The user requires all elements of nodes to be linked, and expects them to form a ring, but that's not necessarily the case.

Inspecting the symbolic state on the verification failure shows the diagram in Figure 6.2, where all nodes are linked, some of them do form a ring, and n is indeed linked to the rest of them, but no other node points to it. This is a counterexample to the assertion in the original program, showing a situation, which is valid according to the specification, but that may not have been considered by the user.

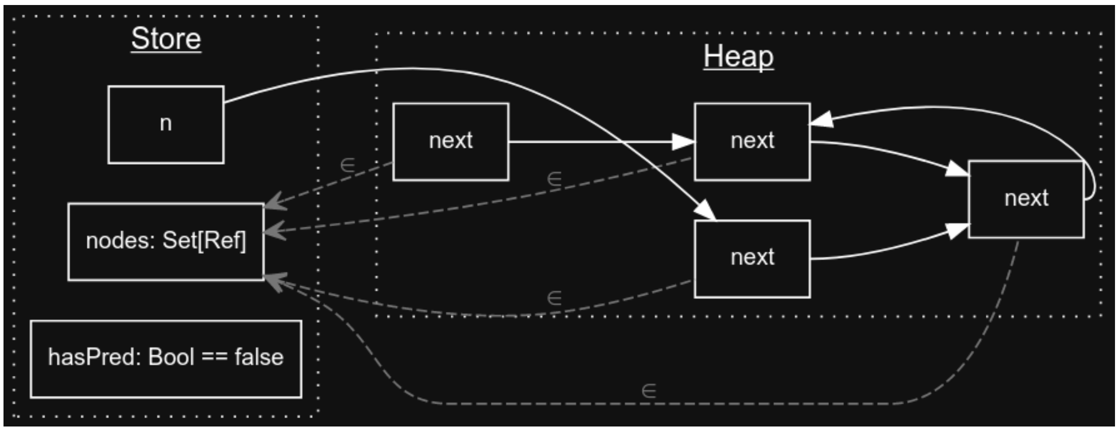


Figure 6.2: A diagram for the failing state in Listing 6.2, where node `n` is actually linked, but has no predecessor since the nodes do not form a ring.

In Listing 6.3 we have another use case. In this snippet, we have a method with two arguments: a set of references, for which we have permission to the `val` and `next` fields, and a reference `n` belonging to that set. The method returns the sum of the value at node `n` and the value of the next node, if it is not null, and at the same time exhales permission from both `val` fields.

Verification of the method fails on line 16, as we may not have enough permission to `n.next.val`. If we visualize the symbolic state at the failure we get the diagram in Figure 6.3. Here, we can see that `n.next` is not null, but points to node `n` itself. Since permission was exhaled for field `val` on `n`, we get the verification error. Adding a precondition specifying that `r.next` is always different from `r` fixes the problem (it is present, but commented-out in the snippet).

Now we will consider an example where no quantified permissions are involved that highlights a fundamental difference between the new debugger and the old one. In Listing 6.4 we have the predicate example from Section 3.1.5 that was used to explain the importance of modeling *combine* operations. In the snippet, we have access to an instance of predicate `pair`, and in the second precondition we learn that if the `ref` field of the wrapped reference were **`null`**, then its `val` field would be equal to 3. In the body of the method we unfold the predicate and try to assert that `val` is 3 (which is obviously not the case).

Inspecting the symbolic state on the verification failure produces the visualization in Figure 6.4. The important thing to notice here is that the counterexample correctly presents a situation where `x.next` is not null, as a consequence of the information learned in the precondition. What we learned in the **`unfolding`** expression happens

```
field val: Int
field next: Ref
method thisPlusNext(nodes: Set[Ref], n: Ref) returns (sum: Int)
  requires forall r: Ref :: r in nodes ==>
                  acc(r.next) && acc(r.val)
  requires forall r: Ref :: r in nodes && r.next != null ==>
                  r.next in nodes
  // requires forall r: Ref :: r in nodes && r.next != null ==>
  //                 r.next != r
  requires n in nodes
{
  sum := n.val

  exhale acc(n.val)
  if (n.next != null) {
    sum := sum + n.next.val  // Verification fails here
    exhale acc(n.next.val)
  }
}
```

Listing 6.3: The method sums the values in field `val` of node `n` and its successor from the set `nodes`, and exhales permissions for both fields. The verification fails because node `n` might point to itself. Uncommenting the additional precondition fixes the problem.

```
field ref: Ref
field val: Int
predicate pair(x: Ref) { acc(x.ref) && acc(x.val) }

method test(x: Ref)
    requires acc(pair(x))
    requires unfolding acc(pair(x)) in
                 x.ref == null ==> x.val == 3
{
    unfold acc(pair(x))
    assert x.val == 3 // Verification fails
}
```

Listing 6.4: An example to show the importance of modeling snapshots. The relation between `x.ref` and `x.val` is captured in different moments and the Alloy model needs to correctly encode the relation between the snapshots to ensure generating valid counterexamples.

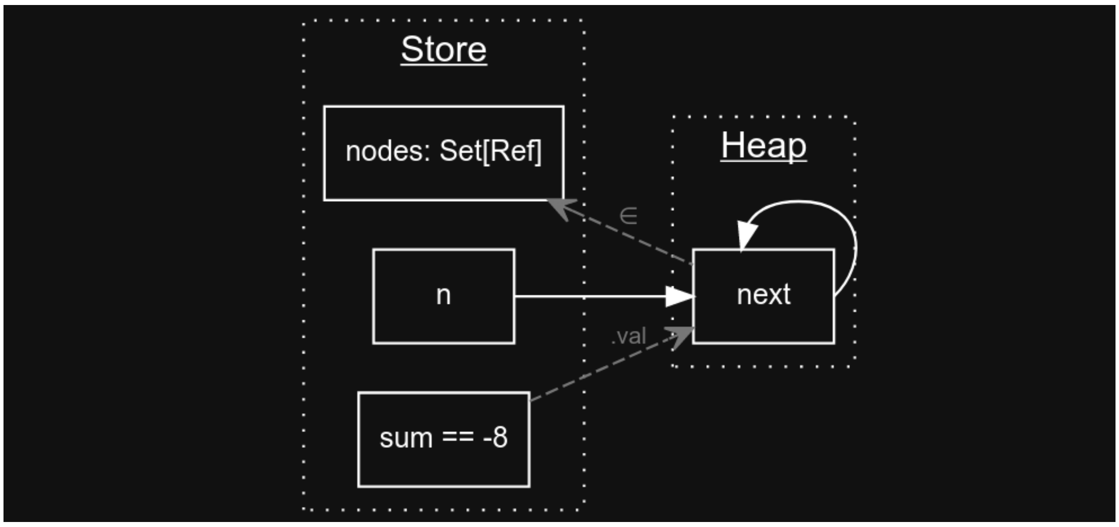


Figure 6.3: A diagram for the failing state in Listing 6.3, where `n.next` is not null, but points to itself.

in a symbolic state where we have access to the values inside the predicate, and the implication is recorded with respect to those symbolic values. These are then abstracted in a snapshot (via a *combine* operation, see Section 3.1.5) when the predicate is folded back. Then, inside the method, when we unfold the predicate again, we learn the relation between that same snapshot but two *new* symbolic values. Without modeling the *combine* operations, there would be no way of relating the new symbolic values back to the ones we had inside the unfolding expression. The old debugger ignored *combines*, therefore it would be unable to deduce this information about the failing state. On the other hand, in the new one, modeling snapshots allows us to learn facts about symbolic values coming from different moments in the execution.

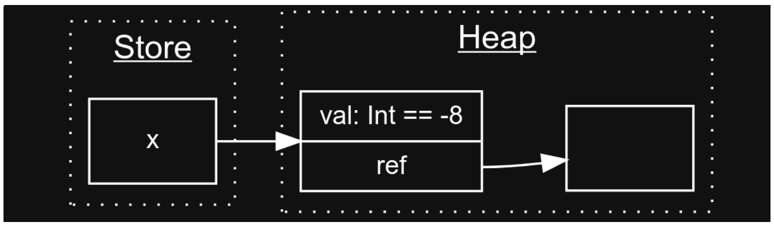
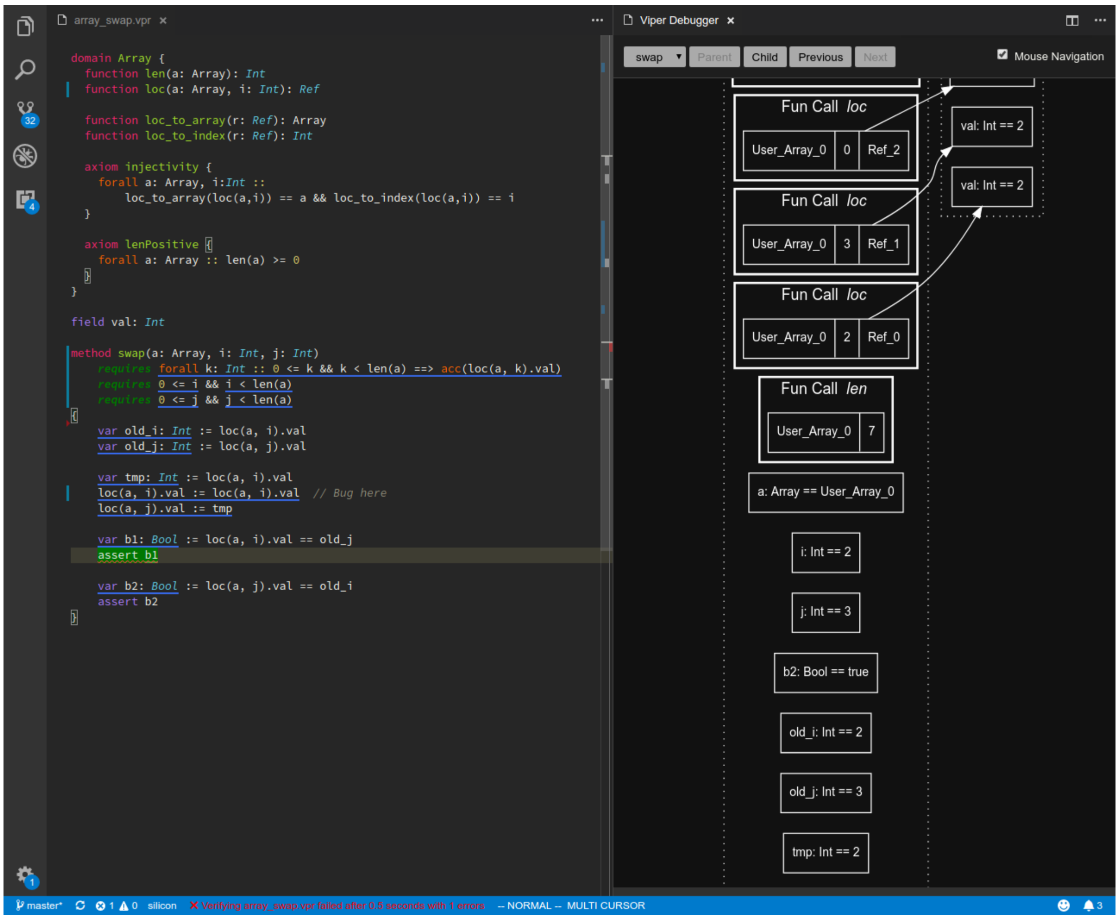


Figure 6.4: A counterexample for the snippet in Listing 3.1, where `n.val` is indeed different from 3, and the instance correctly models the fact that `x.ref` cannot be null.

Finally, we will take a look at an example that uses a custom array domain. Consider the snippet in Listing 6.5. Here, we define an `Array` domain, with function `loc(a, i).val` denoting the *i*-th value in array `a`, and `len(a)` denoting its length. Method `swap` in the snippet is supposed to swap the values at the *i*-th and *j*-th location in the array, but is implemented incorrectly and fails to satisfy the assertion on line 31, stating that the location *i* holds the value that was previously at location *j*. Variables `old_i` and `old_j` have been introduced to work around the lack of support for old expressions in the current implementation of the debugger. The assertions are split over multiple lines (the facts are first assigned to a Boolean-typed variable, then asserted) because of a limitation on the information available in inspectable states, discussed in Section 6.3.5.



```
domain Array {
  function len(a: Array): Int
  function loc(a: Array, i: Int): Ref

  function loc_to_array(r: Ref): Array
  function loc_to_index(r: Ref): Int

  axiom injectivity {
    forall a: Array, i:Int ::
        loc_to_array(loc(a,i)) == a && loc_to_index(loc(a,i)) == i
  }

  axiom lenPositive {
    forall a: Array :: len(a) >= 0
  }
}

field val: Int

method swap(a: Array, i: Int, j: Int)
    requires forall k: Int :: 0 <= k && k < len(a) ==> acc(loc(a, k).val)
    requires 0 <= i && i < len(a)
    requires 0 <= j && j < len(a)
{
    var old_i: Int := loc(a, i).val
    var old_j: Int := loc(a, j).val

    var tmp: Int := loc(a, i).val
    loc(a, i).val := loc(a, i).val  // Bug here
    loc(a, j).val := tmp

    var b1: Bool := loc(a, i).val == old_j
    assert b1

    var b2: Bool := loc(a, j).val == old_i
    assert b2
}
```

```
master*   1  0   silicon   Verifying array_swap.vpr failed after 0.5 seconds with 1 errors   -- NORMAL --  MULTI CURSOR
```

Figure 6.5: A counterexample for the `swap` method in Listing 6.5. Here we can see that the value pointed-to by function `loc(a, j)` has been correctly updated with the new value (2), but `loc(a, i)` still has the its old value because of the bug in the implementation.

Inspecting the symbolic state on the failing assertion, on line 31 of Listing 6.5, yields the visualization in Figure 6.5. In the diagram, we can see that field `val` on the reference at `loc(a, i)`, pointed to by the concrete function call `loc(Array_0, 2)`, holds value 2, which is the value originally at the *i*-th location (as witnessed by `old_i`). This is an example where, both the specification and the failing assertion are correct, instead the failure highlighted by the counterexample is in the implementation of the method.

```
 1 domain Array {
 2   function len(a: Array): Int
 3   function loc(a: Array, i: Int): Ref
 4
 5   function loc_to_arr(r: Ref): Array
 6   function loc_to_idx(r: Ref): Int
 7
 8   axiom injectivity {
 9     forall a: Array, i:Int ::
10         loc_to_arr(loc(a,i)) == a && loc_to_idx(loc(a,i)) == i
11   }
12   axiom lenPositive { forall a: Array :: len(a) >= 0 }
13 }
14
15 field val: Int
16
17 method swap(a: Array, i: Int, j: Int)
18   requires forall k: Int ::     0 <= k && k < len(a)
19                               ==> acc(loc(a, k).val)
20   requires 0 <= i && i < len(a)
21   requires 0 <= j && j < len(a)
22 {
23   var old_i: Int := loc(a, i).val
24   var old_j: Int := loc(a, j).val
25
26   var tmp: Int := loc(a, i).val
27   loc(a, i).val := loc(a, i).val  // Bug here
28   loc(a, j).val := tmp
29
30   var b1: Bool := loc(a, i).val == old_j
31   assert b1  // Verification error
32
33   var b2: Bool := loc(a, j).val == old_i
34   assert b2
35 }
```

Listing 6.5: A method for swapping the *i*-th and *j*-th values in an array. The typo on line 27 leads the assertion on line 31 to fail, since the swap did not actually happen. Variables `old_i` and `old_j` are used to work around the fact that **`old`** expressions are not currently supported by the debugger.

6.1.2 Soundness and Completeness

Consider the following snippet where we have a set of integers and we require that all the integers it contains are 1.

```
method just_a_singleton(ints: Set[Int])
    requires forall i: Int :: i in ints ==> i == 1
    requires |ints| > 1
{
    assert true  // Encoding the state here
}
```

As a consequence of the first precondition, the set can either be empty or $\{1\}$, (since it cannot contain any element other than 1). In addition to that, the second precondition requires that the cardinality of the set be greater than 1, which is in contradiction with the previous fact. This is an incompleteness in Silicon, as asserting false in the body body of the method would result in a verification failure, but if we try inspecting any state in the method's body with the debugger, no instance can be found, because of the contradicting constraints. After commenting out the second precondition or (changing it to `|ints| > 0`) the debugger does generate valid instances of the symbolic state.

Spurious Verification Errors

We observed that, in some situations where we get spurious verification errors, the debugger fails to generate counterexamples, effectively "detecting" contradiction between the error reported by the solver and the specification.

Due to incompleteness in the SMT solver (for example when dealing with quantifiers)[13, 14] and of some of the axiomatisations used by Silicon (e.g. for sets), it is possible that some of the errors being reported are actually spurious. In these situations, the debugger would receive a symbolic execution trace containing a failing assertion and would build a model for the state in which the failure occurred, in order to generate a counterexample. Yet, there are no possible instances for this model, exactly because the verification failure was spurious, and the counterexample generation gives no results.

Such a situation occurs when inspecting the following snippet, presented in Schwerhoff's thesis as an example of incompleteness in the axiomatisation of multisets [11, app. B].

```
method test(ms1: Multiset[Ref], n: Int) {
    inhale |ms1| == n
    var ms2: Multiset[Ref] := ms1 intersection ms1
    // assert ms2 == ms1
    assert |ms2| == n  // Fails without the preceding assertion
}
```

When we inspect the symbolic state corresponding to the failure, we do not observe instances of model we generate, even for large scopes. Running the model manually in Alloy with the MiniSAT solver allows us to inspect its UNSAT core. In there we see that several facts that are in contradiction, including the last SMT query. Removing it from the program allows Alloy to generate instances of the model where the two multisets indeed have the same cardinality. In this case, what allows Alloy to find the contradiction is the fact that we model Viper collections concretely, which means that their cardinality is known and all of their elements are part of the model.

This next example used to be another case where the debugger was able to identify a problem in the case of spurious verification errors. This was true when we were still encoding integers with Alloy's **Int** signature directly. Now, since we use our custom `Integer` signature, this situation is not detected anymore because of the problem with approximating integers, discussed in Section 6.3.3. The example is still discussed because it clearly shows an interesting advantage of modelling **all** integers. This example should be considered in the future when deciding if and how to implement approximations for integers. In the following Viper snippet:

```
function f(): Bool
    ensures (exists n: Int :: n > 0) ==> result == true

method example()
{
    var b: Bool := f()
    assert b  // Verification failure
}
```

Function `f`'s postcondition ensures that if there exists an integer greater than zero, then the result is true. Due to incompleteness with existential quantifiers, Silicon is not able to learn that the result of `f` is *always* true, therefore verification fails.

Again, inspecting the failing state did not generate a counterexample and analysing the UNSAT core in Alloy highlighted problems with the axiom for `f`'s postcondition and the implementation of `isFalse`, used in the encoding of the last failing SMT query: `isFalse[Fun.f1[Unit]]`. Running the analysis after removing the non-proved SMT query used to generate valid instances of the symbolic state.

The function's postcondition were encoded by the following Alloy fact:

```
fact {
  all s: Snap | let result_0 = Fun.f1[s] |
    (some n: Int | n > 0) => (result_0 = True)
}
```

The existential quantifier was translated by Alloy's **some** quantifier (and still is) which has the same semantics. In this case, though, the quantification happened over a bounded scope, the set **Int**, therefore the constraint is easily proved to be true for all objects in the universe (which contradicts the failing SMT query). Again, this is not the case anymore, with the custom implementation of the `Integer` class, because there is nothing ensuring that there exists an instance of `Integer` for each integer value representable in the current scope.

Another example of spurious verification errors that are reported by Silicon, but that do not occur in the debugger, is caused by the use of quantifier triggers which do not have a matching term in the program. Triggers (or patterns) are terms associated with a quantified formula (both in Viper and in the corresponding SMT quantifiers) that solvers use to determine when to instantiate the body of the quantifier [13]. When a term matching a quantifier pattern is found during the execution, then the body of the quantifier is instantiated with the quantified variables replaced by the corresponding values in the term that matched the trigger. Because the problem of testing whether a certain expression is an instance of a trigger is NP-complete [13], SMT solvers are incomplete when using triggers.

```
domain alwaysTrueDomain {
  function alwaysTrue(i: Int): Bool
  function dummyTrigger(i: Int): Bool

  axiom alwaysTrueAxiom {
    forall i: Int :: { dummyTrigger(i) } alwaysTrue(i)
  }
}

method client() {
  assert alwaysTrue(0)  // Verification fails
}
```

In the previous snippet, verification fails because `dummyTrigger` does not appear concretely in the program, therefore the axiom for function `alwaysTrue` is never instantiated. Since triggers are not encoded in our model (Alloy does not need them), the quantifiers they are defined for are always considered in the analysis and they cause a contradiction with the failed SMT query that we got from Silicon.

6.2 Supported Subset of the Viper Language

At the moment, not all features of the Viper language are supported by the modeling technique described in this report, and the debugger does not implement the whole technique. In this section we discuss what features of the language are supported and which are possible but not yet implemented.

6.2.1 Implemented Features

The current implementation of our technique supports the encoding of the primitive types and the operations between objects of these types, as well as encoding the collections (sets, sequences and multisets) and their operations.

Integers are currently modeled by custom wrappers around Alloy's built-in **`Int`** type, and their operations are performed on the wrapped object directly. This approach suffers from the limitations discussed in Section 6.3.3. In that section we also present alternative techniques to modeling integers that would solve these problems.

Permissions in the current implementation are modeled with explicit numerator and denominator in the **`Int`** signature. This technique, again, suffers from the problems related to the boundedness of integers. In the future, this approach should be abandoned and substituted with the relation-based encoding described in Section 3.4.

In general, the translation of all Silicon terms is supported: binary and unary operations, let and conditional expressions, quantifications, logical operations and applications. They are implemented as described in Section 3.1.

Field lookup operations can be encoded (which happens in the context of quantified permissions with simple fields) can be encoded and are supported by the debugger.

The translation of heap-independent-functions and their application is supported. The current implementation has limitations due to the increasing complexity of the model caused by large relations, but we proposed an alternative modeling approach that would solve this problem in Section 6.3.4.

The encoding of domains, domain functions, and domain axioms is supported. Again, for domain functions, the current implementation has the complexity-related limitation that regular heap-independent functions have, but the same solution can be applied to solve the problem in this situation as well.

Inspection is possible in all symbolic states that are provided by the SymbolicExecutionLogger, including those inside methods, functions, predicates, and while loops.

`forperm` expressions and *inhale-exhale assertions* are both "abstracted away" by Silicon: we only see them as additional path conditions or heap chunks in the symbolic state, therefore we can support them, as long as they do not make use of other features that are currently unsupported.

6.2.2 Non-Implemented Features

The following features features of the Viper language are currently not supported by our modeling technique.

`old` and labelled **`old`** expressions are currently not supported. An idea for tackling this problem is discussed in Section 7.1.

The current implementation of the debugger does not support heap-dependent functions. This is because the additional "guard" described in Section 3.7.8 has not been put in place yet. Therefore Alloy would try to satisfy the postcondition axiom with all snapshots and fail. This is merely a limitation of the implementation, not of the technique.

Nested quantifiers are not supported. The translation procedure should theoretically allow translating them, but this has not been verified in practice.

Quantified predicate chunks are currently not supported, though we have the intuition that the current encoding could be extended to model them by combining the approach for predicates and for quantified field chunks.

Magic wands are not supported, therefore neither of the statements or expressions that apply to them are (`package`, `apply`, `packaging`, and `applying`).

6.3 Limitations

In general, we do not see any fundamental limitation in our modeling technique that would prevent it from working, but there are some more concrete problems that have to be tackled.

6.3.1 Cardinalities

The scope for instance generation is set in Alloy by constraining cardinality of signatures. The statement that instructs the analyser to search for an instance has the following form:

run {} for *baseCount* **but** *n sigName*, ...

Where *baseCount* is the maximum cardinality allowed for a signature, unless that specific signature is given a different cardinality in the constraints appearing after **but**. For example, `run {} for 5 but 2 Ref` would allow any signature in the universe to contain at most 5 atoms, except for the `Ref` signature which would be allowed to contain at most 2 elements. In the presence of subtyping, like in the following snippet:

```
abstract sig A {}
sig B extends A {}
sig C extends A {}

run {} for 4
```

There could not be more than 4 atoms of each specific signature, but at the same time the maximum number of atoms of `A` (including its subtypes) is also bounded by 4. Concretely, $\{B_1, B_2, B_3, C_1\}$ would be a valid configuration, while $\{B_1, B_2, B_3, C_1, C_2\}$ would not be allowed because there are 5 atoms in signature `A`. In case `A` were not abstract, then there could be concrete atoms in it as well, but the total number of atoms would still have to be at most 4. More details about carnalities are given in the Alloy language reference[10, Appendix B.7.5].

Choosing optimal cardinalities is extremely important: specifying too restrictive cardinalities could lead to the model not being satisfiable, whereas specifying too large ones quickly leads to an explosion in the complexity of the model and non-terminating analysis.

Currently, we have no automatic technique for computing optimal cardinality bounds. The optimal cardinality of a signature might depend on many factors, such as the the number of fresh variables declared for it, whether it exists inside collections, whether there is some expression in the path conditions that requires it to be higher than a certain bound, etc. Devising heuristics to at least give a rough approximation of the needed cardinalities is left as future work. A manual solution is also possible, where the user is asked to manually cardinalities before counterexample generation is started.

6.3.2 Increasing Complexity when Removing Permissions

Another source of complexity in the model is the removal of permissions from quantified chunks. Whenever some permissions is exhaled from a quantified chunk, Silicon introduces pTaken functions to ensure that the right amount of permission is removed for the right reference. Exhaling multiple times causes Silicon to introduce multiple pTaken functions. Consider the following snippet:

```
method exhalingMultipleTimes(ns: Set[Ref], n1: Ref)
    requires forall r: Ref :: r in ns ==> acc(r.val)
    requires n1 in ns
    requires |ns| == 4
{
    var p1: Perm
    var p2: Perm
    var p3: Perm
    var p4: Perm
    var p5: Perm
    assume p1 + p2 + p3 + p4 + p5 == write
    assume p1 > none && p2 > none && p3 > none && p4 > none &&
           p5 > none
    exhale acc(n1.val, p1)  // (1)
    exhale acc(n1.val, p2)  // (2)
    exhale acc(n1.val, p3)  // (3)
    exhale acc(n1.val, p4)  // (4)
    exhale acc(n1.val, p5)  // (5)
    assert true             // (6)
}
```

Here we exhale symbolic permission amounts for the same field and receiver reference. Silicon will add a pTaken function for each exhale and Alloy will have to compute the result of these functions in order to determine if a reference has permission to a field or not. We inspected the symbolic state at each location marked in the snippet.

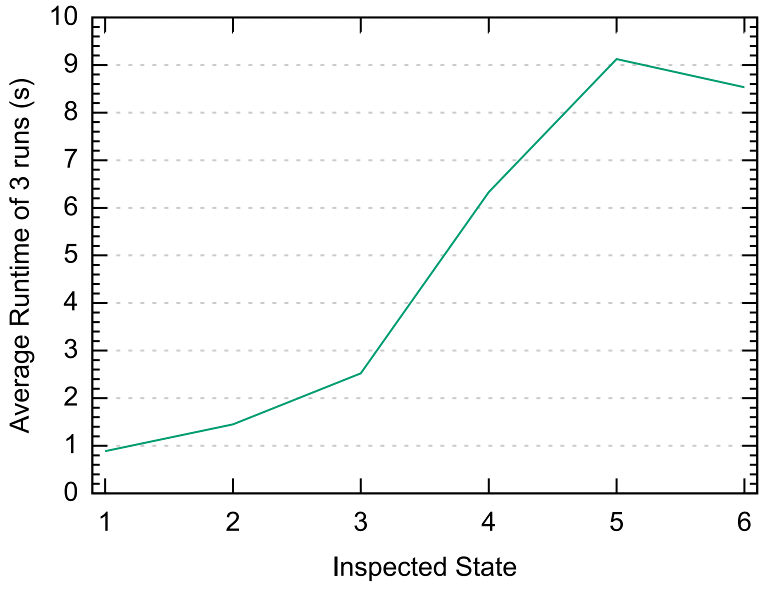


Figure 6.6: The average analysis runtime in Alloy for each inspected symbolic state.

Figure 6.6 shows the time taken to generate an instance for each of the symbolic states we analysed. We can clearly see that the runtime increases more and more for each new pTaken term added, except for the last one. For each fraction of permission exhaled Alloy has to consider a more complicated formula to determine how much permission is left, which increases the time taken to generate an instance. For the last inspection, Alloy takes slightly less time than for the previous one. We speculate that this is because all the permission values have been exhaled and Alloy knows their sum already, therefore it has less constraints to satisfy. When permission amounts are not known precisely, the complexity of the analysis grows.

6.3.3 Limitations of Integers

There are two fundamental limitations with built-in Alloy integers:

1. They are all instantiated, i.e. there exists an instance of the **Int** signature for each of the $2^{bitwidth}$ integer values representable in the universe, which becomes problematic when using total, injective functions.
2. They are bounded, therefore the result of some operations on them cannot be represented in the current universe.

We will discuss both of these problems, some possible solutions for them, and what is currently implemented in the debugger.

All Integers Are Instantiated

As explained in Chapter 3, Viper's **Int** type is encoded using a custom Integer signature, but still relies on a wrapped Alloy **Int** type to implement operations. The reason behind the introduction of this custom signature (rather than using **Int** directly) was to mitigate the first problem with injective total functions that we just described.

Consider the the case where we have an injective total function in the program, for example the loc function used in the integer domain (see Listing 6.5). Its injectivity would be enforced by the following axiom:

```
axiom injectivity {
  forall a: Array, i: Int ::
      loc_to_arr(loc(a,i)) == a && loc_to_idx(loc(a,i)) == i
}
```

This axiom, because it quantifies over integers, would implicitly require having the same number of references and integers in the universe, because for each array-index combination in the arguments of loc, there would need to be a distinct resulting reference to ensure injectivity. This constraint would force us to have a large number of Ref instances in the model (as many as the integers), increasing its complexity. Moreover, most of these reference might not be needed to encode the facts from the actual program.

By declaring a custom Integer signature, we can control precisely the maximum number of instances that could exist in it. Using a custom signature means that the previous axiom would not quantify over the built-in **Int** signature, but over Integer, removing the need to have $2^{bitwidth}$ Ref instances in the universe.

This approach to solving the problem also has limitations, for example the fact that existential quantifiers cannot be modeled soundly in all cases. Consider the following snippet, which may appear in the input program:

```
var b: Bool;
assume (exists i: Int :: i > 0) <==> b
```

The assumption would be registered as part of the path conditions, and finally encoded in the Alloy model as a top-level fact.

```
one sig fresh_qvars {
  temp0: Integer -> lone Integer
}
fact { (some i: Integer | fresh_qvars.temp0[i].value = 0 &&
          i.value > fresh_qvars.temp0[i].value) <=> isTrue[b]
}
```

In this situation there is no guarantee that there will actually be an integer greater than zero in the universe, therefore it is possible for the quantifier to evaluate to false, which means our approximation is unsound. If we were to ensure that we had an `Integer` instance for each integer, then we would defeat their purpose. A solution to the unsoundness problem would be to over-approximate these types of existential quantifiers with *true*, to prevent us from generating invalid instances.

In order to implement this approximation, the translation procedure must be context-aware because we want to over-approximate only the existential quantifiers appearing in a positive context. Those appearing in a negative context should be approximated by $false$ or turned into universal quantifiers, if possible.

$$\begin{array}{rcl}
\exists\, i : Integer \cdot i > 0 & \sim & \mathsf{true} \\
\neg\exists\, i : Integer \cdot i > 0 & \sim & \forall\, i : Integer \cdot \neg(i > 0) \\
\neg(p \land \exists\, i : Integer \cdot i > 0) & \sim & \neg p \lor \forall\, i : Integer \cdot \neg(i > 0)
\end{array}$$

An alternative approach for solving the problem of total functions would be to model them as partial function in Alloy, but this would lead to complications in the rest of the model because missing function results could falsify some constraints, which may lead to unsoundness as well.

Bounded Scope

The second problem with integers in Alloy is that they are bounded. The scope for valid integers when running an analysis is determined by declaring their maximum bitwidth, e.g. by specifying `4 int` in the **`run`** statement.

The fact that the bitwidth is limited means that the result of some operations may not be expressible in the current constraints. For example, setting the bitwidth to 4 allows expressing integers from -8 to +7, but this means that the result of `plus[4, 6]` cannot be represented in the current bitwidth, because $10 > 7$. Alloy gives us two options in these situations: it could allow integers to wrap around in order to remain in the representable bounds (i.e. `plus[4, 6] = -6`) or it could consider all the instances where an overflow occurs as invalid. The first option could potentially lead to the generated instance being inconsistent with the path conditions, making it invalid, therefore it is not a viable solution for us. The second option allows us to keep the operations in the model correct, but prevents us from expressing large integers, because increasing the bitwidth quickly leads to performance degradation.

For example, encoding the snippet in Listing 6.6 and running the resulting model with bitwidths 4, 5, 6, and 7, leads to an important increase in the runtime[1], despite nothing else in the model having changed. The results, shown in Figure 6.7, clearly exhibit an exponential growth of the runtime. The number of expressible integers doubles with each additional bit and Alloy generates atoms for each of them, therefore requires more time to encode the model.

```
field val: Int
method test(ns: Set[Ref])
  requires forall r: Ref :: r in ns ==> acc(r.val) &&
               (exists r2: Ref :: r2 != r && r2 in ns
                                           && r.val == r2.val)
  requires |ns| == 5
{
  assert true
}
```

Listing 6.6: A snippet of code used to test the overhead of increasing the bitwidth when generating executing the model.

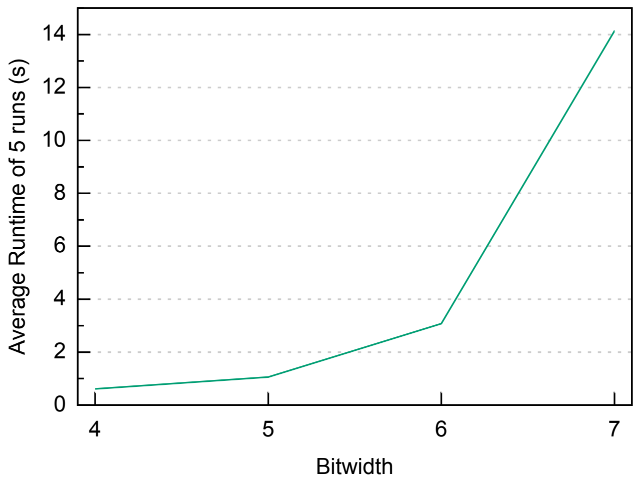


Figure 6.7: The average analysis runtime in Alloy with respect to the integer bitwidth.

An approach to solving this problem could be to completely abstract integers by treating them symbolically and encoding the facts we know about them in relations.

This might lead to the generation of spurious counterexamples, though, since concrete values would need to be approximated. Spurious counterexamples could be filtered out by encoding the resulting instance back into SMT2 and checking it for a verification failure via Z3. The process could be repeated until an actual counterexample is found. More details on this idea and the impact of approximations are discussed in Chapter 2.

An important issue to consider, when approximating integers symbolically, is how to

[1]The benchmark was run via the Alloy API, outside of the IDE, therefore it does not consider the time required to generate the model, which is generally negligible. The tests were run on a system running Debian 9, with a quad-core CPU (Intel i5-3570K, 3.4 GHz) and 8 GB of RAM.

```
abstract sig AbstractInteger {}

one sig IntegerTop extends AbstractInteger {}
one sig IntegerBot extends AbstractInteger {}
sig Integer extends AbstractInteger {
  val: one Int
}

pred int_plus [ i1, i2, i’: AbstractInteger ] {
  // Whenever Bottom is involved we get Bottom
  (i1 = IntegerBot or i2 = IntegerBot)
    => i’ = IntegerBot
    else
  // Whenever Top is involved, but not Bottom, we get Top
  (i1 = IntegerTop or i2 = IntegerTop)
    => i’ = IntegerTop
    else
  // Both operands are positive and the sum would overflow
  (i1.val > 0 and i2.val > 0 and minus[max, i2.val] < i1.val)
    => i’ = IntegerTop
    else
  // Both operands are negative and the sum would underflow
  (i1.val < 0 and i2.val < 0 and minus[min, i2.val] > i1.val)
    => i’ = IntegerBot
    else
  // The sum can be computed in the bounds
  // i’ is an actual integer
  i’ in Integer and i’.val = plus[i1.val, i2.val]
}
```

Listing 6.7: The implementation of a predicate that comnputes integer addition and prevents overflows or underflows by mapping the result to a top ($\top$) or bottom ($\bot$) element respectively. In the snippet, `min` and `max` are helper (nullary) functions, provided by the `integer` module, that denote, respectively the smallest and biggest integer representable with the current bitwidth.

express relations between symbolic integer values and concrete ones, such as the cardinality of a set. The approximation should ensure that when, for example, the cardinality of a set is constrained with respect to some symbolical integer expression, the number of objects that Alloy generates as elements of the set does not contradict other constraints imposed on the symbolic integer value.

Another technique to consider for representing integer values, is to use a wrapper around Alloy's integers (much like in the current implementation), but to additionally introduce top ($\top$) and bottom ($\bot$) elements, that represent, respectively, integers resulting from overflows and underflows. Operations between these objects can be implemented with predicates that check if the operation would result in an overflow or an underflow. This approach would allow not discarding instances where the result of some arithmetic operation could not be represented precisely. An example implementation of the addition operation which considers overflows and underflows is shown in Listing 6.7.

It is important to note that the introduction of the top and bottom elements requires an approximation of comparison operations, in order to avoid getting unsound results. For example, $4 < \top$ is known to always be true, because all concrete values are smaller than "overflowed ones", but $\top < \top$ does not have a clear answer, because we don't know which values the two different tops are approximating. In these situations the comparison should be over-approximated to avoid excluding valid instances of our counterexample space. Applying the approximation, once again, requires knowing if the operation is being performed in a positive or negative context. This has to be determined at translation time. Once the context of the operation is known, this can be implemented by using two different predicates, that check whether both operands are top or bottom and approximate according to the current context.

```
pred int_lt_pos [ i1, i2: AbstractInteger ] {
  (i1 = IntegerTop and i2 = IntegerTop) or
  (i1 = IntegerBot and i2 = IntegerBot) or
  (i1 = IntegerBot and i2 = IntegerTop) or
  (i1 in Integer and i2 in Integer and i1.val < i2.val)
}

pred int_lt_neg [ i1, i2: AbstractInteger ] {
  not (i1 = IntegerTop and i2 = IntegerTop) and
  not (i1 = IntegerBot and i2 = IntegerBot) and
  ( (i1 = IntegerBot and i2 = IntegerTop) or
    (i1 in Integer and i2 in Integer and i1.val < i2.val) )
}
```

Listing 6.8: Example implementation of predicates that over-approximate the less than operation between integers with top and bottom elements.

Listing 6.8 shows an example of two predicates that implement the less than operation between integers and correctly approximate the cases with top and bottom. Using the two predicates, `int_lt_pos[IntegerTop, IntegerTop]` is *true*, because if we had the that comparison in a positive context in the program we would like it to be ignored, whereas `int_lt_neg[IntegerTop, IntegerTop]` is *false*, because in the negative context we ignore values resulting in false.

The problem of bounded integers also affects permissions when using the technique that explicitly models their numerator and denominator (used in the current implementation). With this implementation of the permission objects, operations between them can quickly result in an integer overflow because the fractions are not simplified, leading to large integer values in the numerator or denominator. We can work around this problem by modeling permissions in a symbolic fashion and by keeping track of some relations between them rather than their actual value. Again, this approach requires approximating some values or operations, therefore would need an additional filtering step to exclude spurious counterexamples.

Both for modeling integers and permissions, the approaches that depend on Alloy's built-in integers are more intuitive and easy to implement, but clearly pose strict limitations on what can be modeled. The current implementation of the debugger employs the techniques that depend on Alloy integers, because there is no filtering phase implemented in our pipeline that would allow filtering-out spurious counterexamples. Relation-based encoding of integers and permissions, where the concrete values are completely abstracted-away, would clearly not suffer from the limitations of bounded integers but would potentially result in the generation of spurious counterexamples. For this reason, in future extension of this work, the modeling approaches that use concrete values should be abandoned, and the relation based modeling approach should be used along with the oracle to distinguish spurious counterexamples.

6.3.4 Large relations

In our modeling technique, we represent functions as n-ary relations, where the first $n-1$ elements correspond to the parameters of the function and the n-th element is the return value. These relations are declared as part of the Fun signature (see Section 3.7.8). This approach works for functions with a small number of parameters, but becomes problematic when the arity of the function starts to grow. For example, in the following snippet, the debugger is not able to generate examples of the program's state:

```
function f0(i: Int, j: Int, k: Int, l: Int): Int

method test() {
    var i: Int := f0(1, 2, 3, 4)
    // Encoding state here
}
```

The generation fails in Alloy with the following error:

> "Translation capacity exceeded.
> In this scope, universe contains 33 atom and relations of arity 7 cannot be represented."

A StackOverflow discussion[2]regarding this problem links to a paper where the reason behind it is explained. Kodkod is a constraint solver for relations logic used internally by Alloy. In order to represent a relation of arity k, Kodkod allocates a matrix of size n^k, where n is the number of atoms in the universe. This matrix is represented by a sequential array indexed by a Java integer. Therefore, when n^k exceeds `Integer.MAX_VALUE`, Kodkod is unable to represent our relation [15, sec. 5].

In our example, function `f0` is encoded as a relation, which becomes a tuple `(Fun, Snap, Int, Int, Int, Int, Int)`, where the first element is the signature the relation is declared in (hence the "arity 7" in the error message). There is no simple refactoring that can be applied automatically to the model in order to solve the problem. The only solution is to devise a different encoding, which does not make use of such high-arity relations.

In our case, an alternative approach to modeling functions without using relations, might be to encode them as signatures, with a field for each of the arguments and the result, or to declare a signature for the parameters of each function. For example, a function $f : Snap \to Int \to Int \to Ref$, could be modeled with either of the following encodings:

```
// One signature for each function
sig fun_f {
    a1: one Snap,
    a2: one Int,
    a3: one Int,
    res: one Ref
}

// One "global" Fun signature +
// one signature per arguments configuration
one sig Fun {
  f: Args_f -> one Ref
}
sig Args_f {
  a1: one Snap,
  a2: one Int,
  a3: one Int,
}
```

The current implementation translates functions to relations, therefore it suffers from the problem we just described. Implementing these alternative encodings is certainly required to support a broader subset of Viper programs and should be considered as a topic for future work.

6.3.5 States in the Symbolic Execution Trace

There are currently two limitations with the information provided by Silicon via the symbolic execution trace. Both have to do with the amount of information reported when a verification failure occurs, and both are "by design".

The first limitation has to do with the way the information is gathered by the *symbolic execution logger*, which is responsible for recording the symbolic execution trace. For each fundamental action in the symbolic execution (evaluation, execution, production, or consumption) [12], the state being recorded by the logger is the one to which the action is being applied, effectively the *pre-state*. The effects of the current action will be visible in the pre-state of the next action being executed. This means that whenever a verification failure occurs, for example during the execution of an assert statement, the facts discovered during the execution of the current statement will not be recorded, as their effect would theoretically only be visible in the pre-state of the next statement (which we will not visit since we found a verification failure). Consider the following assert statement, where we compare the *i*-th value of an array with that of variable `old_j`.

```
assert loc(a, i).val == old_j
```

If this assertion were to fail, the we would get a *last failed SMT query* of the form:

$$Lookup(val, sm@27(), loc(a@1, i@6)) = old\_j@17$$

The problem, in this case, is that the snapshot map $sm@27()$ does not exist in the pre-state of the failed assertion because it would only show up in the pre-state of the next statement, therefore it is not constrained with respect to known values and inspecting the failed assertion would lead to a wrong counterexample. A workaround for this problem is to evaluate the expression in a temporary variable before asserting it:

```
var b: Bool := loc(a, i).val == old_j
assert b
```

In this way, all the facts about the objects involved in the fact being asserted are already recorded in the pre-state when the assertion fails. To solve this problem, the symbolic execution trace should be enhanced to contain an additional field with the information in the state currently being built.

[2] `https://stackoverflow.com/questions/21387155/` (Type error has occurred: translation capacity exceeded)

The second limitation is more of a non-obvious design quirk than an actual problem. It happens during the verification of programs with "global branches", and results in some possible verification paths not being reported in the symbolic execution trace. Consider the recursive list implementation in the following snippet.

```
predicate list(this: Ref) {
    acc(this.value) && acc(this.next) &&
        (this.next != null ==> acc(list(this.next)))
}
```

The recursive call in the predicate is "guarded" by a check for `this.next` being null. The symbolic execution of a program unfolding such a predicate causes Silicon to introduce a so-called *global branch* when evaluating the implication. Here, the execution splits to consider the two cases: when `this.next` is null, and when it is not. Silicon would first check the correctness of the program in the *then branch* and then do the same for the *else branch*, but, if a verification error is found while exploring the first branch, then the symbolic execution stops and the *else branch* is not explored, resulting in the symbolic execution trace not including any state in the second verification path.

6.4 Implementation

6.4.1 Performance

The debugger is not currently performant enough to be used on real-world examples. This is due to a couple of reasons: it takes a long time to receive the symbolic execution information from the backend, and not having a good way of approximating the cardinalities when generating the model (see Section 6.3.1) makes constraint solving in Alloy much more complex and slow.

Delivery of the Symbolic Execution Trace

The delivery of the full symbolic execution trace takes a large amount of time. Analysing the graph marking example[3]from the Viper example set yields the timings in Table 6.1. We can see that most of the time is spent waiting for the symbolic execution trace to be delivered to the debugger, after the verification has finished. Further manual testing revealed that the time spent encoding of the path conditions into a JSON object is almost instantaneous and what requires a long time is the actual transfer of the trace to the client.

In the graph-marking example, there are thousands of symbolic states that need to be reported, therefore the JSON message encoding the symbolic execution trace can become quite big. It is yet to be determined whether the bottleneck is on the receiver's

[3]`http://viper.ethz.ch/examples/graph-marking.html`

Event	Time elapsed (s)
Verification terminated	N/A
SymbExLog received by the IDE	68.925
SymbExLog parsed	0.001
SymbExLog logged to panel (diagnostics)	0.740
Internal representation built from SymbExLog	0.968
Model encoded	0.008
Alloy instance generated	1.931

Table 6.1: Duration of the different steps from when the verification has terminated until a Alloy instance has been generated. The *time elapsed* column shows the time elapsed since the previous event.

side (i.e. VS Code) or on the sender's side (ViperServer). This is to be investigated, as it could be problematic for debugging larger programs.

In general, we observed the time taken to build the model to be negligible with respect to the time taken needed by the rest of the system to produce an instance. We observed the encoding time to be the order of tens of milliseconds. On the other hand, the time taken by Alloy to search for an instance satisfying the model varies largely, mainly depending on the cardinalities set in the run command (see Section 6.3.1).

6.4.2 Extensibility

As discussed in Chapter 4, one of the main reasons behind the re-design of the system's architecture was to make it more easily extensible.

The current implementation, where the debugger is separate from the main extension, allows other tools to coexist with it. One can envision adding an new debugger extension for the Carbon backend, that would access the functionalities of the main extension via the ViperApi, just like the current debugger. The debugger itself can be extended to export its own API, which could be used by other extensions to interact with the debugging session.

The translation and other similar operations on terms are implemented via the visitor pattern [16], which makes it easily extensible, such that it possible to add the missing objects which are not supported by the current debugger.

The main extension has been updated to allow registering listeners for "unexpected ViperServer messages" via the ViperApi. With this mechanism, other extensions can be notified when the currently active backend needs to report some additional information which is not destined to the main extension. This is used by the debugger to retrive the symbolic execution trace, emitted by the reporting infrastructure of the backend whenever the `--ideModeAdvanced` flag is set. The same approach could be used by other extensions that require information from the backends or ViperServer itself. ViperServer just needs to be updated to be able to marshal a new message type sent from the reporting infrastructure, and the new messages will be delivered by the main extension to whoever is listening for them, without the need to update its source code.

6.4.3 Modularity

As discussed in Section 6.3, one important factor in the performance, modeling power, and correctness of the debugger depends on the representation we use for the fundamental Viper types.

Given that permissions and operations between them are encoded via the use of Alloy predicates, the rest of the model does not depend on their internal implementation. This makes it possible to change the way they store or approximate their value, without having to update the rest of the system, because their interface can remain unchanged. In fact, we do have two different implementations of fractional permissions (one with explicit numerator / denominator values, and one which relies on relations) which can used interchangeably, without the need to modify other parts of the model.

At the moment, changing the implementation used by one of the types is only possible by modifying the source code to load a different file in the preamble, but this mechanism could easily be made accessible via the debugger's configuration options, to allow loading different encodings in the preamble at runtime.

In the current implementation, integers are modeled by using Alloy's `Int` signature, but one could imagine abstracting the arithmetic and comparison operations via predicates (like it is already done for permissions) in order to enable changing the encoding of integers at runtime as well.

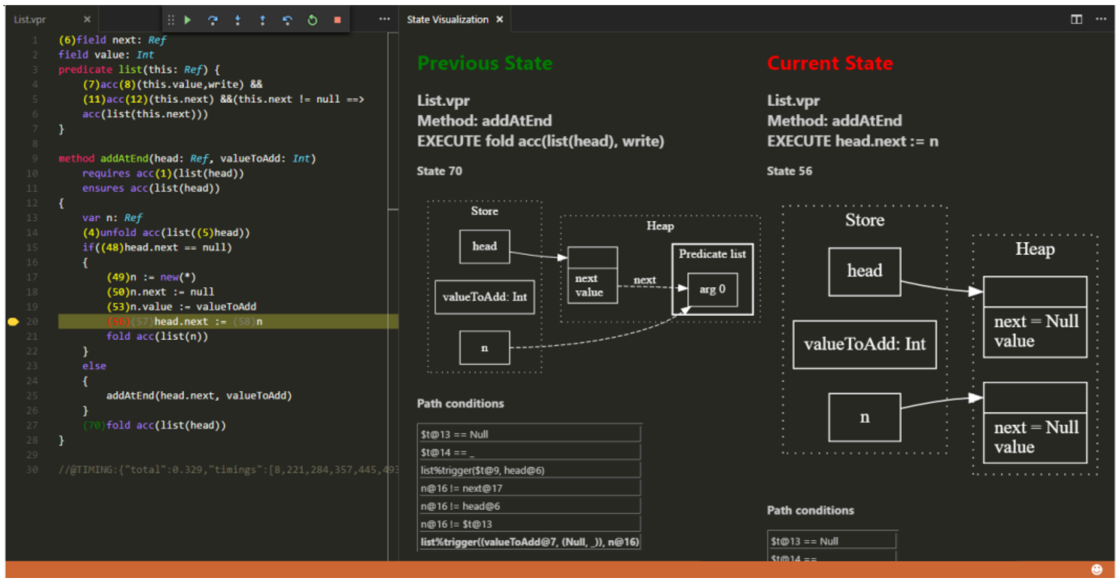
6.5 Comparison to Other Tools

6.5.1 Previous Debugger

The previous debugger is an obvious tool to compare the new debugger with, although internally they work in completely different ways. The original debugger provided a panel in which two verification states could be compared. For each of the two states, the information from the symbolic execution trace was shown. A screenshot of a debugging session in the original debugger is shown in Figure 6.8.

The original debugger provided two different modes: the simplified mode, where the information displayed was filtered to only show a subset of the symbolic states, and an advanced mode, where all verification states were visible.

Our debugger offers only one mode, similar to the advanced mode from the original debugger. In both debuggers a list of the path conditions is shown, though the one



```
(6)field next: Ref
field value: Int
predicate list(this: Ref) {
    (7)acc(8)(this.value,write) &&
    (11)acc(12)(this.next) &&(this.next != null ==>
    acc(list(this.next)))
}

method addAtEnd(head: Ref, valueToAdd: Int)
    requires acc(1)(list(head))
    ensures acc(list(head))
{
    var n: Ref
    (4)unfold acc(list((5)head))
    if((48)head.next == null)
    {
        (49)n := new(*)
        (50)n.next := null
        (53)n.value := valueToAdd
        (56)(57)head.next := (58)n
        fold acc(list(n))
    }
    else
    {
        addAtEnd(head.next, valueToAdd)
    }
    (70)fold acc(list(head))
}

//@TIMING:{"total":0.329,"timings":[8,221,284,357,445,493
```

Previous State

List.vpr
Method: addAtEnd
EXECUTE fold acc(list(head), write)

State 70

Path conditions

$t@13 == Null
$t@14 == _
list%trigger($t@9, head@6)
n@16 != next@17
n@16 != head@6
n@16 != $t@13
list%trigger((valueToAdd@7, (Null, _)), n@16)

Current State

List.vpr
Method: addAtEnd
EXECUTE head.next := n

State 56

Path conditions

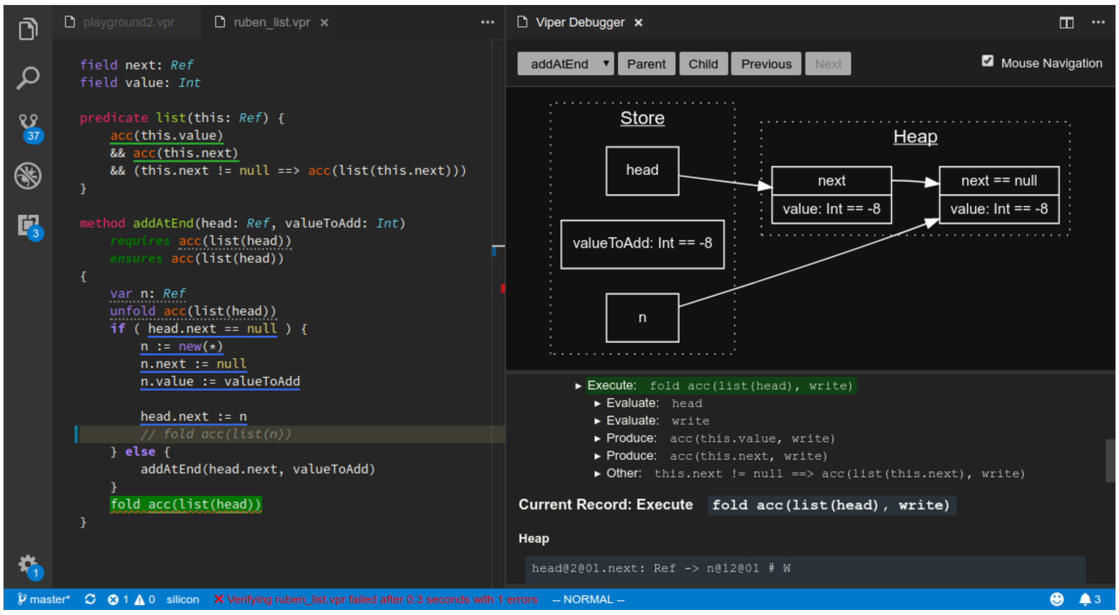
$t@13 == Null
$t@14 ==

Figure 6.8: A debugging session in the original debugger, running in advanced mode. The two columns of the panel show the symbolic execution information of the two selected states.

from the original debugger does not contain all path conditions, because some were explicitly filtered out. The new debugger displays the information about the heap and the store both in textual form, and in the graph. The original debugger shows that information in the graph only. The partial execution tree is shown in both tools, but the one from the original debugger is static, whereas the one in the new debugger can be used to navigate the verification states.

Both debuggers provide a visualization of the store and the heap. The visualization from the original debugger shows the information contained in the symbolic execution trace and additionally tries to infer nullity of the references in the state by checking if they are explicitly known to be null from the path conditions. The new debugger takes a different approach, by encoding the information from the symbolic execution trace in a model and displaying concrete instances generated from that model. This means that the new debugger is able to learn more facts from the path conditions, and visualize objects of which the old debugger was not aware.

One important difference between the diagrams generated by our debugger and those generated by the old one, is that in our case we visualize *concrete* situations, where all variables are assigned a value. In the original debugger, the variables in the program would be shows as symbolic, unless there was a path condition which clearly assigned a concrete value to them. For example, if you compare the screenshots in Figure 6.8 and Figure 6.9, you can see that the `value` field in the visualization from the original debugger is simply reported to be present, whereas in the new visualization, the fields



```
field next: Ref
field value: Int

predicate list(this: Ref) {
    acc(this.value)
    && acc(this.next)
    && (this.next != null ==> acc(list(this.next)))
}

method addAtEnd(head: Ref, valueToAdd: Int)
    requires acc(list(head))
    ensures acc(list(head))
{
    var n: Ref
    unfold acc(list(head))
    if ( head.next == null ) {
        n := new(*)
        n.next := null
        n.value := valueToAdd

        head.next := n
        // fold acc(list(n))
    } else {
        addAtEnd(head.next, valueToAdd)
    }
    fold acc(list(head))
}
```

```
▸ Execute:  fold acc(list(head), write)
    ▸ Evaluate:  head
    ▸ Evaluate:  write
    ▸ Produce:  acc(this.value, write)
    ▸ Produce:  acc(this.next, write)
    ▸ Other:  this.next != null ==> acc(list(this.next), write)
```

Current Record: Execute `fold acc(list(head), write)`

Heap

```
head@2@01.next: Ref -> n@12@01 # W
```

✕ Verifying ruben_list.vpr failed after 0.3 seconds with 1 errors

Figure 6.9: The debugger inspecting the recursive predicate example from Kälin's thesis on a failing state.

are actually assigned a concrete value.

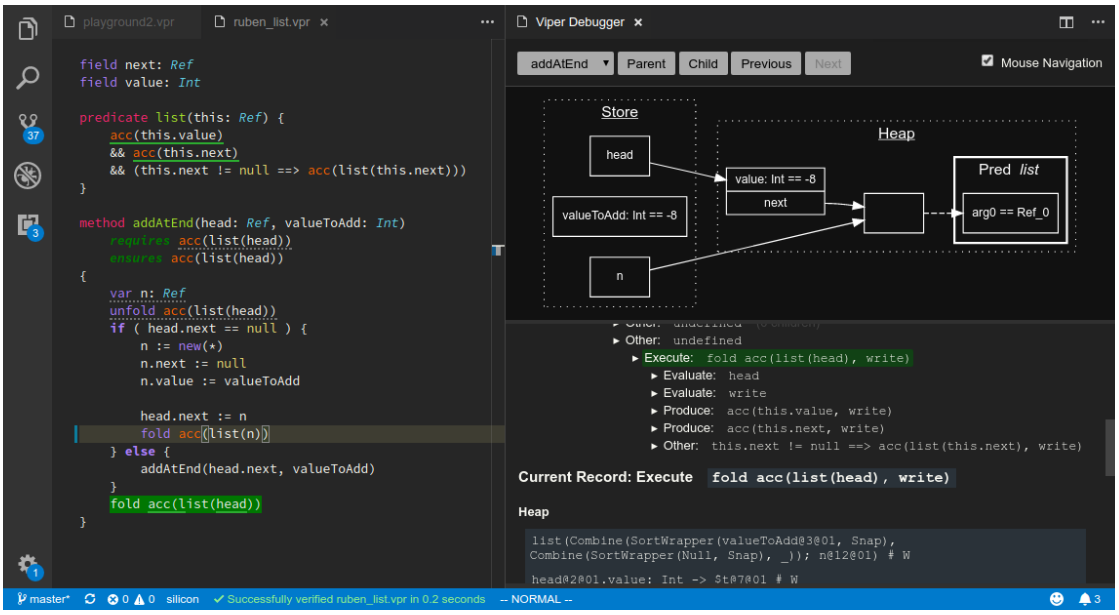
The original debugger provided support to for the visualization of predicates (also recursive) and fields. The new debugger also supports those language constructs and in addition to that it allows visualizing structures defined via quantified permissions and collections, including the elements inside them.

Figure 6.9 shows the example from the original debugger (the same shows in Figure 6.8) in the new debugger. We are inspecting the failing state and the visualization shows a counterexample to the specification: we are trying to fold a recursive predicate, but `head.next` is not folded.

Figure 6.10 shows the same example being inspected in the same location (and following the same path through the program) after the code has been fixed by folding the newly added list node in the then branch. In this case the diagram shows that `head.next` is not null and is indeed folded in a `list` predicate.

6.5.2 VeriFast IDE

VeriFast[5] is a verifier for annotated C and Java programs. It is implemented via symbolic execution and provides an IDE which allows inspecting the symbolic state of a program when a failure is found. Figure 6.11 shows a screenshot of the VeriFast IDE during a debugging session.



```
field next: Ref
field value: Int

predicate list(this: Ref) {
    acc(this.value)
    && acc(this.next)
    && (this.next != null ==> acc(list(this.next)))
}

method addAtEnd(head: Ref, valueToAdd: Int)
    requires acc(list(head))
    ensures acc(list(head))
{
    var n: Ref
    unfold acc(list(head))
    if ( head.next == null ) {
        n := new(*)
        n.next := null
        n.value := valueToAdd

        head.next := n
        fold acc(list(n))
    } else {
        addAtEnd(head.next, valueToAdd)
    }
    fold acc(list(head))
}
```

✓ Successfully verified ruben_list.vpr in 0.2 seconds

```
▸ Other:  undefined
  ▸ Execute:  fold acc(list(head), write)
    ▸ Evaluate:  head
    ▸ Evaluate:  write
    ▸ Produce:  acc(this.value, write)
    ▸ Produce:  acc(this.next, write)
    ▸ Other:  this.next != null ==> acc(list(this.next), write)
```

Current Record: Execute `fold acc(list(head), write)`

Heap

```
list(Combine(SortWrapper(valueToAdd@3@01, Snap),
Combine(SortWrapper(Null, Snap), _)); n@12@01) # W

head@2@01.value: Int -> $t@7@01 # W
```

Figure 6.10: The debugger inspecting the same state as in Figure 6.9, after having fixed the method to correctly fold the newly added list node.

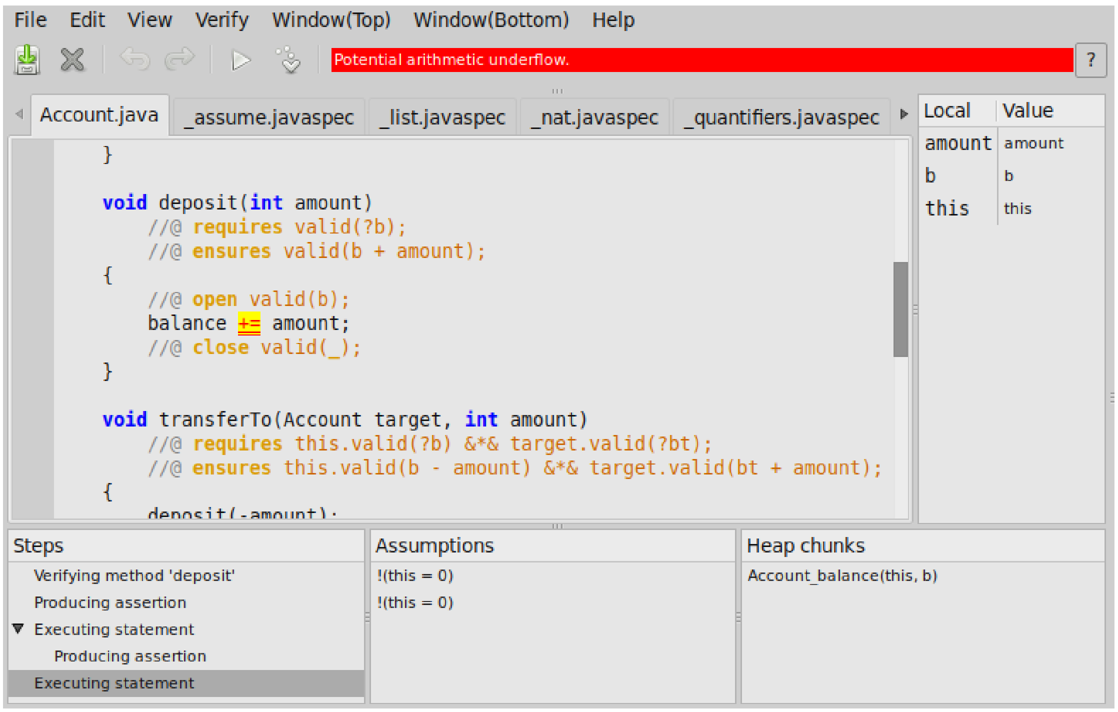
The IDE allows inspecting the execution tree up to the current state. For the current state is displays the path conditions (listed as "Assumptions" in the screenshot), the heap chunks and the local variables in the store (on the right-hand side). Just like for our debugger, the VeriFast IDE displays the key information about a symbolic state and allows navingating the execution tree, but does not try to provide any type of visualization based on the information from the symbolic state.

6.5.3 SED

The *Symbolic Execution Debugger (SED)* [6] allows interactive debugging of programs via symbolic execution. Debugging is performed by exploring the symbolic execution tree of the program. The tool displays information about each verification state such as the symbolic stack and path conditions. In cases the debugger detects multiple possible memory layouts, it allows visualizing the different configurations for local variables only. A screenshot of this feature can be seen in Figure 6.12

The visualization approach of the SED debugger is limited to aliasing between local references, but it allows to visualize all possible combinations of them, a feature that our debugger does not support, as currently it only visualizes one concrete example.

[4]Image from http://i12www.ira.uka.de/key/eclipse/SED/index.html



Potential arithmetic underflow.

```
    }

    void deposit(int amount)
        //@ requires valid(?b);
        //@ ensures valid(b + amount);
    {
        //@ open valid(b);
        balance += amount;
        //@ close valid(_);
    }

    void transferTo(Account target, int amount)
        //@ requires this.valid(?b) &*& target.valid(?bt);
        //@ ensures this.valid(b - amount) &*& target.valid(bt + amount);
    {
        deposit(-amount);
```

Local	Value
amount	amount
b	b
this	this

Steps
- Verifying method 'deposit'
- Producing assertion
- ▼ Executing statement
  - Producing assertion
- Executing statement

Assumptions
- !(this = 0)
- !(this = 0)

Heap chunks
- Account_balance(this, b)

Figure 6.11: Debugging a verification failure in the Verifast IDE

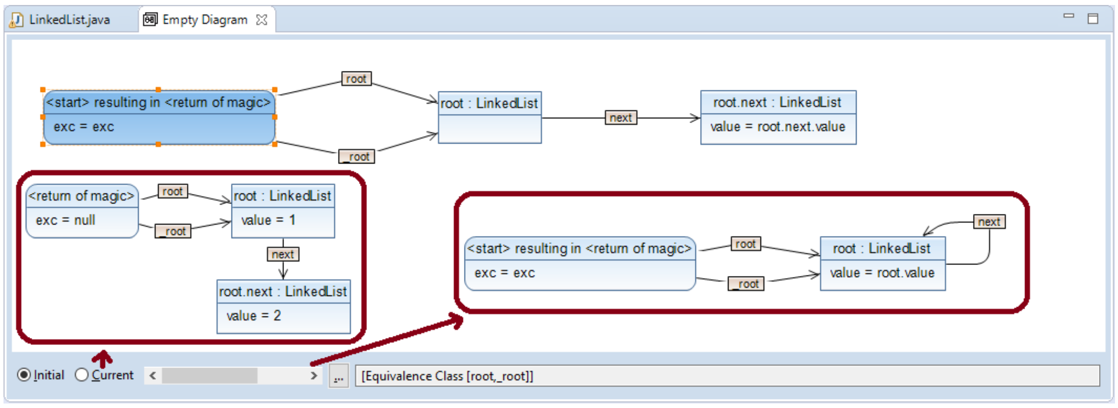


Figure 6.12: Inspecting the memory layouts caused by aliasing in SED. The scrollbar changes which of the two visualizations in the red boxes is shown. Only one concrete aliasing configuration is visible at any time.[4]

7 Conclusion and Future Work

This project started out as a successor to the previous debugger project by Kälin. The original goals were to expand on what had already been done by adding support for quantified permissions, and then expand the editor with a series of features for more easily debugging verification failures. After a while it became obvious that we would have needed a different way of obtaining instances of our model in order to maintain them manageable in size and to be able to quickly add new facts in order to focus on more specific situations. Thus, we began exploring the idea of turning the information we had about the failing symbolic state into an Alloy model, in order to be able to generate instances of our program. After that, the project turned into feasibility study on whether counterexamples could be easily obtained via Alloy and the objective of building an actual usable tool was put aside.

In this thesis, we explored an approach for the creation of an advanced debugging tool for symbolic execution and implemented it in a proof of concept integrated with the existing Viper IDE.

Though the current implementation is just a proof of concept and not all ideas have been implemented (see Section 6.3), we think it provides evidence that this approach is worth investigating further, to fix its limitations.

The modeling approach we devised allows encoding the information from a symbolic state in the symbolic execution trace into an Alloy model, which can be then run to generate concrete instances of the program's state. This allows the visualization of program examples, but most importantly, the visualization of counterexamples, when the verification has failed.

This modeling technique has been implemented in a tool, integrated with the Viper IDE, which allows inspecting the information available to Silicon during the verification of a Viper program, and to automatically generate a visualizations of the program's symbolic states, without leaving the editor. The current implementation allows visualizing predicates and functions like the previous debugger, but is additionally able to visualize objects defined via quantified permissions.

In order to enable the implementation of the new tool and at the same time maintain the system extensible, we designed and implemented a new architecture which integrates the already-existing Viper extension with the new tool. In this new architecture, the debugger and the main extension are completely separate. The Viper IDE is unaware of the internals of the debugger, and can support other extensions that interact with it at the same time.

Moreover, we updated the backend to provide the additional information that is required for encoding all aspects of the programs symbolic state for the generation of counterexamples. This information is made available along with symbolic execution trace. The delivery mechanism of the trace itself has been completely changed, so that now all the information is delivered via ViperServer as a JSON object.

7.1 Future Work

In this section we briefly discuss some topics which may be interesting to explore in future work on the debugger. These include both extensions to the modeling technique and to the implementation of the tool.

Addressing current limitations Of course, the most obvious and important topic for future work would be to address the limitations of both the modeling approach and of the current implementation discussed in Section 6.3.

Approximation Both in Chapter 2 and in Section 6.3 we discussed the need for approximations and consequently for oracle in the pipeline, that could distinguish spurious counterexample candidates from actual counterexamples. One clear topic for further work is therefore to add support for approximations both in the approach and in the actual pipeline of the system.

Support for user-provided constraints As outlined in Chapter 2, our technique conceptually allows for additional constraints to be added to the model. We think an interesting idea to explore would be that of enabling the user to add new constraints for the purpose of narrowing the counterexample search space to a more specific subset or to exclude situations that may not be of interest.

Modeling multiple states This report describes an approach for translating a single verification state into an Alloy model. Viper allows using **`old`** and labelled **`old`** expressions to refer to the value of heap-dependent expressions at previous point in time. Silicon keeps track of the old heap and resolves labelled **`old`** expressions to a symbolic value for us, but it's not clear how multiple states could be modeled together and at the same time kept consistent with each other. One problem is certainly that, in the symbolic execution trace, we have no distinction between a symbolic value from the current state and one that was "retrieved" from an old state. An idea to support this could be to exploit the `orderings` module provided by Alloy, which allows building dynamic models where multiple symbolic states are represented and related to each other. One important consideration to make is, given that our end-goal is that of building small **visual**, is how to represent the information from multiple verification states in an easy to understand and compact way.

Exploration of counterexample space While the space explored by Alloy is bounded, there still might be many instances that satisfy the constraints of a model. This means that there might be different counterexamples to present to the user. The alloy API that we use in the ViperServer backend allows searching for the "next" instance, once a first one has been found. It would be interesting to investigate ways to integrate this mechanism into the debugger. One important challenge to solve, when researching this feature, is that of devising a way to prevent Alloy from generating multiple instances which only differ slightly one from the other.

Bibliography

[1] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer-Verlag, 2016.

[2] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability modulo theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. 1 edition, 2009. ISBN 9781586039295. doi: 10.3233/978-1-58603-929-5-825.

[4] Ruben Kälin. Advanced features for an integrated development environment. Master's thesis, ETH Zürich, 2015.

[5] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20398-5.

[6] Martin Hentschel, Richard Bubel, and Reiner Hähnle. The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, Mar 2018. ISSN 1433-2787. doi: 10.1007/s10009-018-0490-9. URL `https://doi.org/10.1007/s10009-018-0490-9`.

[7] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 342–363, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-36750-5.

[8] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*.

Springer, 2016. ISBN 978-3-319-49811-9. doi: 10.1007/978-3-319-49812-6. URL `http://dx.doi.org/10.1007/978-3-319-49812-6`.

[9] AlloyTools. alloytools.org, 2017. URL `http://alloytools.org/`.

[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012. ISBN 0262017156, 9780262017152.

[11] Malte Schwerhoff. *Advancing Automated, Permission-Bases Program Verification Using Symbolic Execution*. PhD thesis, ETH Zürich, 2016.

[12] Bart Jacobs, Jan Smans, and Frank Piessens. Verification of imperative programs: The verifast approach. a draft course text. Technical Report Technical Report CW-578, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2010.

[13] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005. ISSN 0004-5411. doi: 10.1145/1066100.1066102. URL `http://doi.acm.org/10.1145/1066100.1066102`.

[14] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73595-3.

[15] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code, 2011.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. URL `http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1`.

A
Configuration

The following lists contains all the configuration options available in the debugger extension, their default value (in parentheses) and their description. Note that all these options are part of the `viperDebuggerSettings` namespace, therefore they should be prepended with it when used in the configuration.

For the debugger to be able to function correctly, the Silicon backend has to be configured in the main extension with the additional `"--ideModeAdvanced"` and `"--numberOfParallelVerifiers 1"` options.

`debugImmediately (false)`
If enabled, the debugger is started immediately, whenever a Viper file is opened.

`logLevel ("INFO")`
The minimum level of importance of the messages to log.
Valid values are `DEBUG`, `INFO`, `WARNING`, and `ERROR`.

`alloySATSolver ("minisatprover(jni)")`
The SAT solver Alloy should use to generate instances of the model.
Valid values are `sat4j`, `minisat(jni)`, and `minisatprover(jni)`

`integerBitWidth (4)`
The bit-width Alloy should use in the constraints for the model.
Valid values are positive integers.

`instancesBaseCount (6)`
The base value to add to the instance count when generating the constraints for the model.
Valid values are positive integers.

`instancesBaseCount (6)`
The base value to add to the instance count when generating the constraints for the model.
Valid values are positive integers.

`highlighting.currentStateBackgroundColor (#007701)`
The background colour of the currently selected verification state.
Valid values are hexadecimal or RGB colours.

`highlighting.currentStateForegroundColor (#EEEEEE)`
The foreground colour of the currently selected verification state.
Valid values are hexadecimal or RGB colours.

`highlighting.topLevelStateUnderlineColor (#707070)`
The underline colour of top-level verification states.
Valid values are hexadecimal or RGB colours.

`highlighting.childStateUnderlineColor (#2CAD30)`
The underline colour for children of the selected verification state.
Valid values are hexadecimal or RGB colours.

`highlighting.siblingStateUnderlineColor (#3668E8)`
The underline colour for siblings of the selected verification state.
Valid values are hexadecimal or RGB colours.

B
Preamble source

The following snippet presents the complete static preamble, that is added at the the beginning of each encoded model.

```
// ===== Preamble (resources/preamble.als) =====
open util/boolean
open util/ternary
open util/integer
open util/relation
abstract sig Snap {}
one sig Unit extends Snap {}
abstract sig SortWrapper extends Snap {
    wrapped: one univ
}
pred sortwrapper_new [ e: univ, sw: Snap] {
    sw in SortWrapper
    sw.wrapped = e
}
abstract sig Combine extends Snap {
    left: one Snap,
    right: one Snap
}
pred combine [ l, r: Snap, c: Combine ] {
    c.left = l && c.right = r
    c.left != c && c.right != c
    c not in c.^left
  c not in c.^right
}
abstract sig Integer {
    value: one Int
}
fact { all i1, i2: Integer | i1 = i2 <=> i1.value = i2.value }
```

```
// ===== Perms (resources/perms.als) =====
abstract sig Perm {
    num: one Int,
    denom: one Int
} {
    num >= 0
    denom > 0
}
one sig W in Perm {} {
    num = 1
    denom = 1
}
one sig Z in Perm {} {
    num = 0
    denom = 1
}
pred perm_new[ n, d: Int, p': Perm ] {
    p'.num = n
    p'.denom = d
}
pred perm_less[ p1, p2: Perm ] {
  mul[p1.num, p2.denom] < mul[p2.num, p1.denom]
}
pred perm_at_most[ p1, p2: Perm ] {
  mul[p1.num, p2.denom] <= mul[p2.num, p1.denom]
}
pred perm_at_least[ p1, p2: Perm ] {
  mul[p1.num, p2.denom] >= mul[p2.num, p1.denom]
}
pred perm_greater[ p1, p2: Perm ] {
  mul[p1.num, p2.denom] > mul[p2.num, p1.denom]
}
pred perm_plus[ p1, p2, p': Perm ] {
  (p1.denom = p2.denom)
    =>
        (p'.num = plus[p1.num, p2.num] &&
         p'.denom = p1.denom)
    else
        (p'.num = plus[mul[p1.num, p2.denom], mul[p2.num,
    p1.denom]] &&
         p'.denom = mul[p1.denom, p2.denom])
}
pred perm_minus[ p1, p2, p': Perm ] {
  perm_equals[ p1, p2 ]
    => p' = Z
    else (
      p1.denom = p2.denom
        => (
          p'.num = minus[p1.num, p2.num] and
          p'.denom = p1.denom
        ) else (
          p'.num = minus[mul[p1.num, p2.denom], mul[p2.num,
   p1.denom]] and
          p'.denom = mul[p1.denom, p2.denom]
        )
    )
}
pred int_perm_div[ p: Perm, d: Int, p': Perm ] {
  p'.num = p.num
  p'.denom = mul[p.denom, d]
}
pred perm_mul[ p1, p2, p': Perm ] {
  p'.num = mul[p1.num, p2.num]
  p'.denom = mul[p1.denom, p2.denom]
}
pred int_perm_mul[ i: Int, p, p': Perm ] {
  p'.num = mul[p.num, i]
  p'.denom = p.denom
}
pred perm_min[ p1, p2, p': Perm ] {
  mul[p1.num, p2.denom] < mul[p2.num, p1.denom]
    => (p' = p1)
    else (p' = p2)
}
pred perm_equals [ p1, p2: Perm ] {
  (p1.num = p2.num && p1.denom = p2.denom)
  || (mul[p1.num, p2.denom] = mul[p1.denom, p2.num])
}
// ===== Sets (resources/set_fun.als) =====
abstract sig Set {
  set_elems: set univ
}
pred empty_set [ s': Set ] {
  no s'.set_elems
}
pred set_singleton [ e: univ, s': Set ] {
  s'.set_elems = e
  one e
}
pred set_add [ s1: Set, e: univ, s': Set ] {
  s'.set_elems = s1.set_elems + e
  one e
}
pred set_cardinality [ s1: Set, i: Integer ] {
  #(s1.set_elems) = i.value
}
pred set_difference [ s1, s2, s': Set ] {
  s'.set_elems = s1.set_elems - s2.set_elems
}
pred set_intersection [ s1, s2, s': Set ] {
  s'.set_elems = s1.set_elems & s2.set_elems
}
pred set_union [ s1, s2, s': Set ] {
  s'.set_elems = s1.set_elems + s2.set_elems
}
pred set_in [ e: univ, s1: Set ] {
  e in s1.set_elems
  one e
  some s1.set_elems
}
pred set_subset [ s1, s2: Set ] {
  s1.set_elems in s2.set_elems
}
pred set_disjoint [ s1, s2: Set ] {
  disjoint[s1.set_elems, s2.set_elems]
}
pred set_equals [ s1, s2: Set ] {
  s1.set_elems = s2.set_elems
}
// ===== Seqs (resources/seq.als) =====
abstract sig Seq {
  seq_rel: seq univ
} {
  isSeq[seq_rel]
}
pred seq_ranged [ from, to: Integer, s': Seq ] {
  let ints = { i: Int, ci: Integer | ci.value = i and
    from.value <= i and i < to.value } |
  #ints = sub[to.value, from.value] and
  s'.seq_rel = subseq[ints, from.value, sub[to.value, 1]]
}
pred seq_singleton [ e: univ, s': Seq ] {
  s'.seq_rel[0] = e
  #(s'.seq_rel) = 1
}
pred seq_append [ s1, s2, s': Seq ] {
  s'.seq_rel = append[s1.seq_rel, s2.seq_rel]
}
pred seq_length [ s: Seq, i: Integer ] {
  #(s.seq_rel) = i.value
}
fun seq_at [ s: Seq, i: Integer ]: one univ {
    s.seq_rel[i.value]
}
pred seq_take [ s: Seq, i: Integer, s': Seq ] {
  let to = sub[i.value, 1] |
  s'.seq_rel = subseq[ s.seq_rel, 0, to]
}
pred seq_drop [ s: Seq, i: Integer, s': Seq ] {
  let to = sub[#s.seq_rel, 1] |
  s'.seq_rel = subseq[ s.seq_rel, i.value, to ]
}
pred seq_in [ s1: Seq, e: univ ] {
  e in elems[s1.seq_rel]
}
pred seq_update [ s: Seq, i: Integer, e: univ, s': Seq ] {
  s'.seq_rel = setAt[s.seq_rel, i.value, e]
}
// ===== Multiset (resources/multiset.als) =====
abstract sig Multiset {
    ms_elems: univ -> lone Int
} {
    all i: univ.ms_elems | gt[i, 0]
}
pred empty_multiset [ ms': Multiset ] {
    no ms'.ms_elems
}
pred multiset_singleton [ e: univ, ms': Multiset ] {
    ms'.ms_elems = (e -> 1)
}
pred multiset_add [ ms1: Multiset, elem: univ, ms': Multiset ] {
    ms'.ms_elems =    { e: univ, v: Int | e in (elem -
dom[ms1.ms_elems]) and v = 1 } +
        { e: univ, v: Int | e in (dom[ms1.ms_elems] - elem) and v = ms1.ms_elems[e] } +
        { e: univ, v: Int | e in (dom[ms1.ms_elems] & elem) and v = add[ms1.ms_elems[e], 1] }
}
pred multiset_cardinality [ ms: Multiset, card: Integer ] {
  card.value = (let s = { c: Int, e: univ | (e -> c) in ms.ms_elems } |
          (sum i: (s).univ | mul[#(s[i]), i]) )
  card.value >= 0
}
pred multiset_difference [ ms1, ms2, ms': Multiset ] {
  ms'.ms_elems = { e: univ, v: Int | e in (dom[ms1.ms_elems] - dom[ms2.ms_elems]) and v = ms1.ms_elems[e] } +
        { e: univ, v: Int | e in (dom[ms1.ms_elems] & dom[ms2.ms_elems]) and
                  e.(ms2.ms_elems) < e.(ms1.ms_elems) and
                  v = minus[e.(ms1.ms_elems), e.(ms2.ms_elems)] }
}
pred multiset_intersection [ ms1, ms2, ms': Multiset ] {
  ms'.ms_elems = { e: univ, v: Int | e in (dom[ms1.ms_elems] & dom[ms2.ms_elems]) and v = min[e.(ms1.ms_elems) + e.(ms2.ms_elems)] }
}
pred multiset_union [ ms1, ms2, ms': Multiset ] {
  ms'.ms_elems = { e: univ, v: Int | e in (dom[ms2.ms_elems] - dom[ms1.ms_elems]) and v = ms2.ms_elems[e] } +
        { e: univ, v: Int | e in (dom[ms1.ms_elems] - dom[ms2.ms_elems]) and v = ms1.ms_elems[e] } +
        { e: univ, v: Int | e in (dom[ms1.ms_elems] & dom[ms2.ms_elems]) and v = add[ms1.ms_elems[e], ms2.ms_elems[e]] }
}
pred multiset_subset [ ms1, ms2: Multiset ] {
  dom[ms1.ms_elems] in dom[ms2.ms_elems]
  { all e: dom[ms1.ms_elems] | ms1.ms_elems[e] <= ms2.ms_elems[e] }
}
pred multiset_count [ ms1: Multiset, e: univ, c: Integer ] {
  c.value = ms1.ms_elems[e]
```

```
}
pred multiset_equals [ ms1, ms2: Multiset ] {
    ms1.ms_elems = ms2.ms_elems
}
```

C

Encoded Running Example

The following listing presents the complete Alloy model encoded from the running example in Listing 3.3 from Chapter 3.

```
sig Ref {
  val: lone Integer,
  refTypedFields': set Ref
} {
  refTypedFields' = none
}

one sig NULL extends Ref {}
fact { NULL.refTypedFields' = none && no NULL.val }

one sig Store {
  nodes': one Set_Ref,
  n1': one Ref,
  n2': one Ref,
  v': one Integer,
  refTypedVars': set Ref
} {
  refTypedVars' = n1' + n2'
}
one sig nodes_3_01 in Set_Ref {}
one sig n1_4_01 in Ref {}
one sig n2_5_01 in Ref {}
one sig v_12_01 in Integer {}
fact { Store.nodes' = nodes_3_01 }
fact { Store.n1' = n1_4_01 }
fact { Store.n2' = n2_5_01 }
fact { Store.v' = v_12_01 }

```

```
// Heap Chunks
// QA r@9@01 :: ((r@9@01 in nodes@3@01) ==> (inv@10@01(r@9@01)
    == r@9@01))
fact { (all r_9_01: Ref | (set_in[r_9_01, nodes_3_01] =>
    (Fun.inv_10_01[r_9_01] = r_9_01))) }
// QA r :: ((inv@10@01(r) in nodes@3@01) ==> (inv@10@01(r) ==
    r))
fact { (all r: Ref | (set_in[Fun.inv_10_01[r], nodes_3_01] =>
    (Fun.inv_10_01[r] = r))) }
one sig t_7_01 in FVF_Integer {}
one sig fresh_quantifier_vars_0 {
  q_temp_0': Ref -> lone Perm
}
fact { all r: Ref | perm_minus[(set_in[Fun.inv_10_01[r],
    nodes_3_01] implies W else Z), PTAKEN.pTaken_13_01[r],
    fresh_quantifier_vars_0.q_temp_0'[r]] &&
        fresh_quantifier_vars_0.q_temp_0'[r] = PermFun.val[r,
    t_7_01] }
fact { all r: Ref | (some fvf: (t_7_01) | one PermFun.val[r,
    fvf] and perm_less[Z, PermFun.val[r, fvf]]) <=> (one r.val) }
fact { all r: Ref, fvf: (t_7_01) | one PermFun.val[r, fvf] =>
    (perm_at_most[PermFun.val[r, fvf], W]) }
one sig Preds {}

// Path Conditions
// ($t@11@01 == Combine(_, _))
one sig t_11_01 in Snap {}
one sig temp_0' in Snap {}
fact { combine[Unit, Unit, temp_0'] }
fact { (t_11_01 = temp_0') }
// ($t@8@01 == Combine(_, $t@11@01))
one sig t_8_01 in Snap {}
one sig temp_1' in Snap {}
fact { combine[Unit, t_11_01, temp_1'] }
fact { (t_8_01 = temp_1') }
// (n2@5@01 in nodes@3@01)
fact { set_in[n2_5_01, nodes_3_01] }
// (n1@4@01 in nodes@3@01)
fact { set_in[n1_4_01, nodes_3_01] }
// ($t@6@01 == Combine(SortWrapper($t@7@01, Snap), $t@8@01))
one sig t_6_01 in Snap {}
one sig temp_2' in Snap {}
one sig temp_3' in Snap {}
```

```
fact { sortwrapper_new[t_7_01, temp_3'] }
fact { combine[temp_3', t_8_01, temp_2'] }
fact { (t_6_01 = temp_2') }
// QA r@9@01 :: ((r@9@01 in nodes@3@01) ==> !((r@9@01 == Null)))
fact { (all r_9_01: Ref | (set_in[r_9_01, nodes_3_01] =>
    !((r_9_01 = NULL)))) }
// QA r@9@01 :: ((r@9@01 in nodes@3@01) ==> (inv@10@01(r@9@01)
    == r@9@01))
fact { (all r_9_01: Ref | (set_in[r_9_01, nodes_3_01] =>
    (Fun.inv_10_01[r_9_01] = r_9_01))) }
// (SetCardinality:(nodes@3@01) == 3)
one sig temp_4' in Integer {}
one sig temp_5' in Integer {}
fact { set_cardinality[nodes_3_01, temp_4'] }
fact { temp_5'.value = 3 }
fact { (temp_4'.value = temp_5'.value) }
// QA r :: ((inv@10@01(r) in nodes@3@01) ==> (inv@10@01(r) ==
    r))
fact { (all r: Ref | (set_in[Fun.inv_10_01[r], nodes_3_01] =>
    (Fun.inv_10_01[r] = r))) }
// ((inv@10@01(n1@4@01) in nodes@3@01) ==> (Lookup(val,
    sm@14@01(), n1@4@01) == Lookup(val, $t@7@01, n1@4@01)))
one sig sm_14_01 in FVF_Integer {}
fact { (set_in[Fun.inv_10_01[n1_4_01], nodes_3_01] =>
    (Lookup.val[sm_14_01, n1_4_01].value = Lookup.val[t_7_01,
    n1_4_01].value)) }

// Permission functions
one sig PermFun {
  val: (Ref -> FVF_Integer -> lone Perm)
}

// Functions
one sig Fun {
  inv_10_01: (Ref -> one Ref)
}

// Lookup functions
one sig Lookup {
  val: (FVF_Integer -> Ref -> one Integer)
}
fact { all fvf: FVF_Integer, r: Ref | (one PermFun.val[r, fvf]
    and perm_less[Z, PermFun.val[r, fvf]]) => (Lookup.val[fvf,
    r] = r.val) }
// Other sorts
sig Set_Ref extends Set {} {
  set_elems in Ref
}
sig FVF_Integer {}

// Macros
// QA r :: (pTaken@13@01(r) == (r == n1@4@01) ? ((inv@10@01(r)
    in nodes@3@01) ? W : Z PermMin W) : Z)
one sig fresh_quantifier_vars_1 {
  q_temp_0': Ref -> lone Perm
}
fact { (all r: Ref | perm_min[(set_in[Fun.inv_10_01[r],
    nodes_3_01] implies W else Z), W,
    fresh_quantifier_vars_1.q_temp_0'[r]]) }
fact { (all r: Ref | perm_equals[PTAKEN.pTaken_13_01[r], ((r =
    n1_4_01) implies fresh_quantifier_vars_1.q_temp_0'[r] else
    Z)]) }
one sig PTAKEN {
  pTaken_13_01: Ref -> one Perm
}

// Constraint from last non-proved smt query
one sig temp_6' in Perm {}
fact { perm_minus[(set_in[Fun.inv_10_01[n2_5_01], nodes_3_01]
    implies W else Z), PTAKEN.pTaken_13_01[n2_5_01], temp_6'] &&
        !(perm_less[Z, temp_6']) }

// No object unreachable from the Store
fact { Ref = Store.refTypedVars'.*refTypedFields' + NULL +
    (SortWrapper.wrapped <: Ref) + Store.nodes'.set_elems }

// Signarure Restrictions
fact { Set = Store.nodes' }
fact { FVF_Integer = t_7_01 + sm_14_01 }
fact { Snap = t_11_01 + temp_0' + t_8_01 + temp_1' + t_6_01 +
    temp_2' + temp_3' + Unit }
fact { Perm = PermFun.val[Ref, FVF_Integer] + temp_6' + W + Z }
fact { Seq = none }
fact { Multiset = none }

run {} for 11 but 4 int, 1 Set, 2 FVF_Integer, 4 Perm
```

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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):

Visual Debugging for Symbolic Execution

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

Name(s):	**First name(s):**
Aurecchia	Alessio

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	**Signature(s)**
Zürich, 16.09.2018	[illegible]

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