Adding Debugging Functionality to Viper's Symbolic Execution Backend

Practical Work Report

Andrea Keusch
Supervised by Prof. Dr. Peter Müller, Dr. Marco Eilers, Lea Salome Brugger

March 12, 2024

1 Introduction

Viper [4] is an intermediate language for program verification, the process of formally proving the correctness of programs with respect to a given specification. It has two backends: a symbolic execution backend (Silicon) and a verification condition generation backend (Carbon). We work on the Silicon backend, which takes a Viper program as input and verifies it using symbolic execution [2]. This verification technique steps through the program as during program execution but keeps a so-called symbolic state (instead of a concrete state). Each variable is assigned a concrete value, if it is known, or a symbolic expression otherwise. Additionally, it stores a path condition, which is the collection of all preconditions, assumptions, branch conditions and other constraints that are known to hold. Whenever an assertion (or invariant or postcondition) is encountered, Silicon tries to prove it under the assumption of the path condition. In order to do so, it invokes an SMT solver, by default Z3 [3]. If the solver can successfully prove the assertion, this means that the statement holds for every concrete execution. If not all assertions can be proven, the error is reported and the user has to resolve it somehow. To do so efficiently, information about the error and tools that enable analysis are needed. Providing those is the main goal of the project.

Providing Debugging Information in a User-Readable Format. The first step of this project is to provide as much information as possible to the user. This includes the program state (e.g. variables, their (symbolic) values and information about the heap) and the path conditions. We will attach this information to the error object of each failed assertion and make it possible to display it in a user-readable format. To achieve this, the information must be provided in a format that can be understood on Viper level. Internally, Silicon uses a different representation called `Term`s that refers to functions, values and

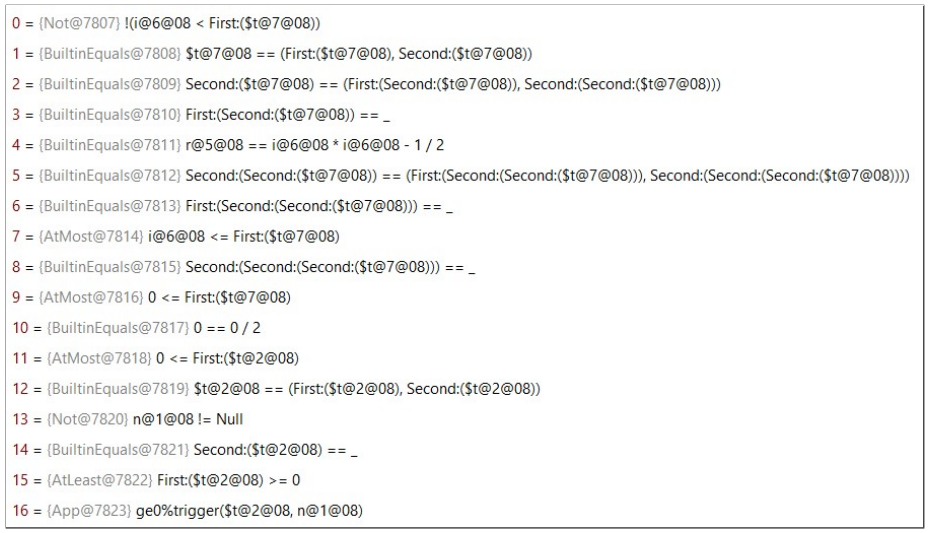

Figure 1: Assumptions collected during verification of Listing A in Viper's internal representation.

concepts that do not exist on the Viper level. `Term`s are generally not user-readable, see Figure 1 for some examples. Hence, we have to modify Silicon such that it maintains the Viper-level representation. While our motivation is to enable debugging, there might be other cases where having access to Viper level representations is useful. In addition to that, we want to structure the path conditions in a way that makes them easier to parse and understand for users, for example, by grouping assumptions that belong together. Chapter 3 discusses how we collect and represent the relevant information.

Basic User Interaction. Even with access to all the provided information, the user likely has to experiment with additional assumptions and assertions. Instead of doing this by modifying the code, we want to provide a tool that enables the user to interact directly with the symbolic state. The interactions include modifying the path condition by adding and removing assumptions and modifying the to-be-proven assertion. Since the symbolic state is attached to the error, verification does not have to be rerun. Instead, the prover can be invoked directly, which is faster and more convenient. This technique can also help to circumvent timeouts because the path condition and assertion can be simplified in a way that speeds up the proof. Through these interactions, the user gains valuable insights about the error and how to resolve the issue. In Chapter 4, we introduce this tool, which we call the "interactive debugging mode".

2 Background on Viper's Symbolic Execution Backend

Viper's symbolic execution backend (Silicon) is composed of two main components. First, we have the so-called Silver module. It takes a Viper program as input and parses it into an abstract syntax tree (AST). The second one is the Silicon module, which takes such an AST and performs symbolic execution on it. In the following, we describe the relevant parts of each component. For more details please refer to the papers by Müller at al. [4] and Schwerhoff [5].

2.1 Silver Module

The input to Silver is a Viper program. As a first step, the `Parser` takes this Viper program and parses it into a `PProgram`. Next, the `Resolver` runs, which invokes the `NameAnalyser` and the `Typechecker`. The `NameAnalyser` identifies the different scopes in the program and for each one it creates a map from variables to their types. The `Typechecker` then uses this scope map to infer the type of each expression and to verify the type safety of the program. If the resolver is successful, the `Translator` translates the `PProgram` into an `ast.Program`, which is an AST consisting of `ast.Stmt`s, `ast.Exp`s and other nodes. This AST represents the initial Viper program in a way that users can still understand it. Therefore, we will use this representation to provide debugging information to the user.

2.2 Silicon Module

Silicon gets the `ast.Program` as input and proceeds by verifying each component using symbolic execution. For this project, the verification of function and method bodies are of importance. The `Executor` takes a statement (`ast.Stmt`), e.g. the method body, and performs symbolic execution on it. In this process, expressions (`ast.Exp`) are evaluated to `Term`s by the `Evaluator`. `Term`s are the internal representation used by Silicon. In a lot of cases, they are not user-readable and might contain concepts that do not exist on the Viper level, for example, inverse functions and heap snapshots. The executor and evaluator update the `State` and add assumptions to the `PathConditionStack`.

The `State` represents the symbolic state. It consists of the `Store` and the `Heap`. The `Store` is a map from AST variables to the `Term` that represents its current value. This value often is not a concrete value but an internally used variable. The evaluator, for example, uses this mapping to evaluate AST variables to terms. The `Heap` is a list of `Chunk`s, each of which represents a heap location. Among other information, chunks store a `Term` depicting the permission that is currently hold to this location. This permission is used in several places, for example, to compute new permissions or to check whether there is sufficient permission for updating a heap location. The `State` is an immutable data structure. Every update actually creates a new instance of the state.

The `Prover` is the interface to the SMT solver. The functions `assume` and `assert` assume and assert a Boolean expression, respectively. The function `fresh` returns a new internal variable, for example, `fresh('x', Bool)` might return a variable named `x@02@08`. The added identifiers can be seen as a version, which brings us to an important concept. Viper variables are mutable and therefore might change their values during execution. Assume you explore an execution path that enters a branch with condition $x = 1$, modifies some variables and then enters another branch with condition $x = 2$. Adding these two conditions like that would lead to an unsatisfiable path condition, which would allow us to prove anything (including `False`). If we use variable versions instead, we add, for example, the assumptions $x@02@08 = 1$ and $x@04@08 = 2$. The contradiction has vanished and we have a reasonable path condition. Therefore, it is crucial to keep track of variable versions.

The `PathConditionStack` is the data structure that keeps track of all assumptions and branch conditions that have been recorded during verification on the current execution path. It is organized in layers (`PathConditionStackLayer`). A new layer is pushed, for example, when entering a branch and popped when jumping back to a branching point in order to investigate a different branch. The assumptions and branch conditions are stored in the individual layers. `PathConditionStack` provides functions that add assumptions to a layer, return the combination of all assumption of all layers, push and pop layers.

The `Decider` is basically a wrapper around the `PathConditionStack` and the `Prover`. Its `assume` function ensures that those two data structures are always in sync with respect to the recorded assumptions. The `assert` and `fresh` functions are delegated to the prover. Moreover, the decider implements functions that push and pop path condition stack layers.

The `Verifier` is the central data structure which will be passed around during verification. Of importance for this project is only that it contains the `Decider`. The verifier, decider, path condition stack and its layers are mutable data structures. Hence, if we need to remember their state at a given point during verification, we need to clone them or store the required information explicitly.

3 Providing Debugging Information in a User-Readable Format

The first project goal is to collect the path conditions and the state in a Viper representation such that users understand them. A suitable presentation are the AST expressions, `ast.Exp`, created by the Silver module. However, during verification, Silicon transforms `ast.Exp`s into the internal `Term` representation

and the `ast.Exp`s get lost. We intervene by keeping track of the Viper expressions and attaching them to verification errors. In this chapter, we will discuss how the relevant information is collected during verification, where it is stored and how we attach it to the verification error.

3.1 Final Expressions

We want to provide the assumptions as AST expressions. However, the original AST expressions are missing some information, namely the variable versions which we mentioned in Section 2 when discussing the `fresh` function. It is important that the user can distinguish the different variable versions. To achieve this, we modify the original AST expression resulting in what we call the "final expression". Below, we describe how the final expression differs from the original one.

First, we replace variables in the original expression by one with a version, using `ast.LocalVarWithVersion`. The version is obtained from the `fresh` function of the prover. Initially, it just returned a term (e.g. `x@02@08`). We extended it such that it also returns an AST variable with version (e.g. `x@02`). This implicitly creates a bijective mapping between a `term.Var` and an `ast.LocalVarWithVersion`, which is required to evaluate the user inputs correctly during interactive debugging. Note that we do not need the last part of the term's variable name (here: '@08') because it is constant and hence, does not provide additional information. Ignoring it makes the expressions more readable.

Heap locations can also change their values. However, using the same approach as with the variables is not possible due to aliasing. Instead, we use a feature that is already built-in, namely heap snapshots. Every time a field is accessed, we create a heap snapshot and store it in the snapshot map maintained by the state. The snapshot has the label 'debug@' followed by a unique identifier. The field access of the original expression is then wrapped in a labelled `old` expression, for example, `old[debug@21](x@2.val)`. This is our final expression. In the interactive debugging mode, these labels can be used as if they were part of the original program.

3.2 Storing the Final Expressions

In Section 3.1, we introduced the final expressions. Now, we describe where these expressions are created and stored. Conceptually, whenever a `Term` is created, modified or stored, we need to do the same operation for the final expression. In the following, we describe the main places where we made the according adjustments. Note that we focused on providing final expressions for our debugging functionality. There might be other places where having access to a final expression is not yet provided but would be useful.

The main place where `Term`s are created is the evaluator. The `eval` functions take as input the original `ast.Exp` and return the corresponding `Term`. We modified these functions such that they apply the modifications described in Section 3.1 to the original `ast.Exp` and return this final expression as an additional return value. Another example on where we added `ast.Exp` as a return value are the `buildPathConditions` functions.

The final expressions are stored along the corresponding `Term` in several places. First of all, we extended the chunks, which represent heap locations, such that they maintain the permission and arguments of that chunk as final expressions in the fields `argsExp` and `permExp`. These fields are used in several places in order to compute a new permission or check for sufficient permission. Moreover, we modified the store such that it maps variables to a tuple of `Term` and final expression. This mapping is mainly used in the evaluator when evaluating an `ast.AbstractLocalVar`.

3.3 Data Structure: DebugExp

For debugging purposes, the final expressions alone do not provide all information that we would like to have and they do not offer grouping of related assumptions. Hence, we introduce a new data structure called `DebugExp`, which incorporates all information to a single assumption or a group of assumptions. In this section, we describe this data structure in detail. Section 3.4 elaborates on the hierarchical structure which allows for grouping of assumptions and Section 3.5 describes where the `DebugExp`s are stored.

`DebugExp` has the following fields:

- `id: Int` - a unique ID used to simplify interaction later on
- `description: Option[String]` - an optional description. It is mainly used to provide a description to a group of `DebugExp`s. In most other cases, the description is set because there is no expression available, for example, for internal assumptions.
- `originalExp: Option[ast.Exp]` - the original AST expression meaning that the variables have no version. At the moment, this field is not used anywhere but we attach it for completeness.
- `finalExp: Option[ast.Exp]` - the final AST expression as discussed in Section 3.1.
- `term: Option[Term]` - the term that is given to the prover (the SMT solver). It should be equal to the final expression in the sense that the final expression would be translated to exactly this term. For groups of `DebugExp`s, the term on the top-level might be empty because the terms are stored in the children.

- `isInternal_: Boolean` - Some assumptions are a consequence of how Silicon works underneath. These are usually not meaningful to users and hence, are labelled as "internal". Examples of such cases are snapshot definitions and inverse function axioms.
- `children : InsertionOrderedSet[DebugExp]` - a list of children which makes the data structure hierarchical, see Section 3.4. It allows us to group `DebugExp`s, for example, all preconditions of the function `add(a, b)` are collected under one `DebugExp` with the description "precondition of `add(a,b)`".

The expressions (`originalExp` and `finalExp`) might be empty for two reasons. First, the top-level node of a group of `DebugExp`s usually has neither expressions nor terms since those are stored in the children. Second, for some internal `DebugExp`s, we were not able to find a suitable expression. In those cases, the description field is filled instead.

The `DebugExp` data structure further provides a `toString` function. It takes as argument a `DebugExpPrintConfiguration`, which specifies what information to include in the string. At the moment, one can choose between (1) including or excluding internal `DebugExp`s and (2) up to which hierarchy depth the children should be included. For the latter, one can set a default value and additionally set individual depths for certain `DebugExp` IDs (which override the default). Per default, internal `DebugExp`s are printed and the hierarchy depth is set to 5. Those values can be configured by the user during interactive debugging, see Section 4.4.

The function `removeChildrenById(ids: Seq[Int])` iterates recursively over the children and removes all `DebugExp`s whose ID are given as input. This will also be used in the interactive debugger.

3.4 Hierarchical Structure of DebugExp

Sometimes multiple assumptions belong together, for example, when having multiple preconditions or loop invariants. Another example would be that during evaluation of a predicate unfolding several assumptions might be added. The user should be able to quickly identify that those are related. Grouping assumptions using a `DebugExp` is possible thanks to the hierarchical structure (i.e. the `children` field). There are two ways of creating a hierarchical `DebugExp`. One option is to create a `DebugExp` and set the `children` field directly. The other way is to use the `startDebugSubExp()` and `finishDebugSubExp(String)` functions, which we provide in the `Decider`. All assumptions that are encountered between two matching calls of `startDebugSubExp()` and `finishDebugSubExp( String)` will be collected in a dedicated list in the path condition stack layers. Once `finishDebugSubExp( String)` is called, this list is attached as children to a new `DebugExp` which is then added to the assumptions. The string argument is the description of the top-level `DebugExp`. Nested calls to those two

functions are supported and create arbitrarily deep hierarchies. An example of how the hierarchy is displayed to the user is provided in Figure 3 under "[39] Loop invariant".

3.5 Collecting the Assumptions

During verification, Silicon collects the assumptions and branch conditions using the `assume` and `setBranchCondition` functions of the `Decider`. The terms are stored in a layered path condition stack, see Section 2. In this project, we want to store more information about those assumptions. More concretely, we store all assumptions as `DebugExp`s and we also have the Viper representation of branch conditions.

For this purpose, we modified the `assume` and `setBranchCondition` function such that they take as additional argument a `DebugExp` and a pair of `ast.Exp` (the original and final expressions), respectively. They are stored in a list in the path condition stack layer similar to the terms.

In addition to assumptions, Silicon works with axioms and definitions. Examples are axioms about built-in types and functions such as sets and sequences, min and max functions. Moreover, they contain the signatures of the functions, methods and predicates of the input Viper program. They are in a sense additional assumptions which are added directly to the prover (skipping the decider). As a result, they do not appear anywhere in the path condition stack. Without them, it is not possible to use functions since their pre- and postconditions are not available and we would not be able to prove anything. Hence, we will need these axioms and definitions for our interactive debugging mode. To that end, everything that is emitted to the prover during startup is stored in a list directly in the prover. We will pass those emits to the prover used during interactive debugging.

3.6 Attaching the Debugging Information to the Error

Silicon already provides some functionality to attach information to errors in the `createFailure` method of `SymbolicExecutionRules`. This method creates failure contexts and attaches them to the error. We introduce the data structure `SiliconDebuggingFailureContext`, which records the relevant debugging information, namely the current state, verifier, branch conditions, assumptions (in the form of `DebugExp`s), function declarations, axioms and the failed assertion. Ultimately, this is the data structure that will be passed to the interactive debugger.

4 Interactive Debugging Mode

The second goal of the project is to give users the possibility to interact with the symbolic state in order to debug their Viper programs. To that end, we implemented `SiliconDebugger`, an interactive debugging mode. It provides access to all collected information about all verification errors, as discussed in Section 3. Most importantly, the user can inspect the symbolic state, including all assumptions and the failed assertion, and interact with it by removing/adding assumptions or changing the assertion.

4.1 Enabling Debugging

Most of the debugging functionality is executed if and only if the configuration flag `--enableDebugging` is set. More precisely, the collection of assumptions as `DebugExp`s, the creation of debug heap snapshots in the state and the interactive debugging mode are only available if debugging is enabled. Note that for now, tests cannot run with debugging enabled because the debugger relies on user input and would not terminate in a testing context. Per default, debugging is disabled.

4.2 Starting the Interactive Debugging Mode

So far, the interactive debugging mode is a command line tool which has been implemented in `SiliconDebugger`. Once the `DefaultMainVerifier` completes verification and is ready to report the results, it will automatically start the interactive debugging mode (if debugging is enabled). The Silicon debugging failure contexts (see Section 3.6) are passed to the debugger together with additional data that is required for handling user input, namely the reporter, resolver, program, translator and the main verifier.

4.3 Proof Obligations

The `ProofObligation` is a data structure used in the interactive debugging mode only. It maintains all information required to handle user input and execute the proof, namely the state, verifier, definitions, axioms, branch conditions, assumptions, assertion, print configuration, original error reason, resolver and translator. The user can interact with the proof obligation, for example by adding or removing assumptions. A proof obligation verifies successfully iff under the given assumptions the assertion is guaranteed to hold.

4.4 Interaction Loop

The main part of the debugging mode is an interaction loop. The debugger will enter the debugging loop automatically if at least one verification error was detected.

```
[0]: Assertion r == n.val * (n.val + 1) / 2 might not hold. (gaussianSumRef.vpr@21.12--21.38)

[1]: There might be insufficient permission to access x.val (gaussianSumRef.vpr@34.5--34.10)

Which verification result do you want to debug next [0 - 1] (or q to quit):
0
```

Figure 2: Example of how to choose an error to debug.

1. The first step is to choose which error to debug. If there is only one error this step is skipped. Otherwise the errors are printed and the user chooses one by entering the error ID, see Figure 2.
2. In the background, the `ProofObligation` is initialized, see below for details. This also includes initializing the verifier, prover, resolver and translator.
3. We enter the main loop where the user can interact with the proof obligation repeatedly until quit. Every time the proof obligation changes, it will be printed, see Figure 3. The interaction options are:
   (a) Set assertion (`ass`): The assertion is changed by entering a valid Boolean expression. An example is shown in Figure 4. After successfully translating and evaluating the assertion, the proof is automatically executed and the result is reported to the user.
   (b) Remove assumption (`ra`): This will remove assumptions. To do so, the user provides the IDs of the assumptions to be removed, see Figure 5. The assumptions are then removed from the proof obligation, the decider and the prover without any further checks. Note that removing an assumption will also remove all its children. Removing assumptions can be useful in order to circumvent timeouts in case the user knows that some assumptions are not needed for the proof.
   (c) Add assumption (`aa`): The user enters a valid Boolean expression which will be added to the assumptions. Before adding it, Silicon checks whether the assumption can be proven under the already existing assumptions. This prevents users from adding contradicting assumptions.
   (d) Prove (`p`): This attempts to prove the current proof obligation without any modifications and reports the result.
   (e) Reset (`r`): This resets the obligation to its initial state.
   (f) Change configuration (`c`): This allows the user to change the fields of `DebugExpPrintConfiguration` which controls what information is printed. The configuration is used in `toString` of `DebugExp`.
   (g) Print (`print`): This prints a single `DebugExp` by providing its ID.

```
Original Error:
        gaussianSumRef.vpr@21.12--21.38 (inside gaussian_sum)
        Assertion r == n.val * (n.val + 1) / 2 might not hold.

Branch Conditions:
        !(i@6 < old[debug@2](n@1.val))

Store:
        n -> (n@1@08,n@1)
        i -> (i@6@08,i@6)
        r -> (r@5@08,r@5)

Heap:
        n@1@08.val -> First:($t@7@08) # W

Assumptions:
    [48] !(i@6 < old[debug@2](n@1.val))
    [39] Loop invariant
        [23] Snapshot
        [28] Snapshot
        [29] Empty snapshot
        [30] r@5 == i@6 * (i@6 - 1) / 2
        [31] Snapshot
        [32] Empty snapshot
        [33] i@6 <= old[debug@1](n@1.val)
        [34] Empty snapshot
        [35] 0 <= old[debug@1](n@1.val)
    [36] 0 == 0 * (0 - 1) / 2
    [37] 0 <= old[debug@0](n@1.val)
    [22] unfolded ge0(n)
        [11] Snapshot
        [15] none < write * (write * write) ==> !(n@1 == null)
        [16] Empty snapshot
        [17] old[debug@0](n@1.val) >= 0
        [21] PredicateTrigger(ge0(n@1))

Assertion:
    r@5 == old[debug@4](n@1.val) * (old[debug@4](n@1.val) + 1) / 2
```

Figure 3: Example of how a proof obligation is displayed. The proof obligation has been created based on the Viper program provided in Listing A.

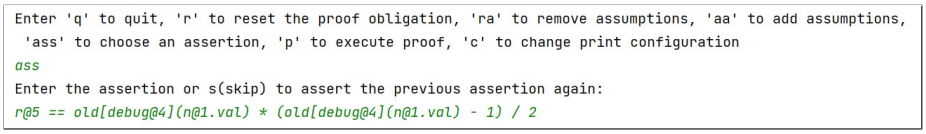

Figure 4: Example of how to set a new assertion.

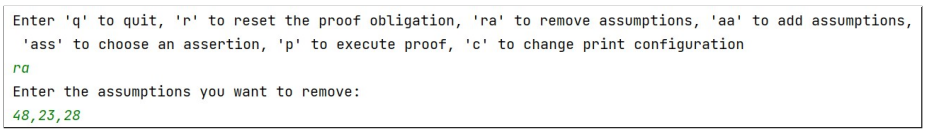

Figure 5: Example of how the user can remove assumptions.

(h) Quit (`q`): This jumps back to step 1, which allows the user to debug other errors.

Valid User Inputs. When adding an assumption or changing the assertion, the user can enter an arbitrary Boolean expression. Valid inputs are all valid Viper expressions. Additionally, the user can enter variables with versions (e.g. `x@2`) and use labels that have been created on-the-fly (e.g. `line@14`). Note that the user can also enter variables without a version, but only if they are in scope at the instruction that caused the verification error. The entered expressions are parsed, type-checked, translated and evaluated as described in Section 2. Some behavior for handling debugging expressions is implemented in dedicated debug classes, which are described in Section 4.5.

4.5 Debug Parser, Resolver, Translator and Verifier

The user needs to be able to enter variables with versions and use labels that were created on-the-fly. To accomplish this, we implemented `DebugParser`, `DebugTranslator`, `DebugResolver` and `DebugTypeChecker`, which we describe in this section. They extend the corresponding classes in the Silver module and are only used in the interactive debugging mode. Extracting the debugging functionality to dedicated classes was important because in the original programs variable and label names must not contain the symbol '@' as this would break the verification process.

The implementations for the `DebugParser` and `DebugTranslator` are straightforward. They inherit all functionality from the superclass and only add the parsing and translating of variables with versions and debug labels.

The `DebugResolver` and `DebugTypeChecker` infer the types and verify the type safety of expressions. In the interactive debugging mode we need to ensure that (1) we can type-check all variables that are in scope at the failed assertion and (2) that all variables with versions can be type-checked.

To achieve (1), the resolver is initialized with the `NameAnalyser` that was used during the initial verification. This includes the list of scopes. What remains to do is computing the scope of the failed assertion. For that reason, we added an info field `SourcePNodeInfo`, which is attached to all `ast.Exp`s and `ast.Stmt`s during the initial translation in Silver's translator. It contains a reference to the source `PNode`, which is the output of the parser. Using this reference, we can look up the current scope in the `NameAnalyser`. Then, the user can use all variables (in the original form, e.g. without the version) in this scope. During evaluation, these variables will be evaluated to the corresponding term and final expression by looking up the variable in the store.
For (2), we added `debugVariableTypes`, a map from variables with version to their `PType`, to the `Decider`. Every time a fresh variable is requested (e.g. `fresh` of the decider is called), the decider adds the corresponding mapping. The `PType` of a variable is extracted using the `PNode` reference in `SourcePNodeInfo`. We pass this map to the debug typechecker when initializing it. Now, typechecking a variable with version boils down to looking up the `PType` in this map.

The verifier used in the interactive debugger is a `WorkerVerifier`. To initialize it, we pass the `MainVerifier` that starts the interactive debugging mode (the `DefaultMainVerifier` instance). This creates an empty verifier and prover; in other words, they do not have any assumptions yet. First, the prover is set up by passing the emits collected during startup of the initial verification, namely definitions and axioms. Then, we add all assumptions (`DebugExp`s) to the path condition stack and prover using the `decider.assume()` function, similar to what would happen during verification. Note that all assumptions are added to the same path condition stack layer. This is reasonable for the debugging mode because we do not need to handle branches.

5 Conclusion and Future Work

During this project, we built a foundation for a debugging tool for Viper's symbolic execution backend. The relevant information is collected in a structured, user-readable format and attached to the error. Moreover, our interactive debugging mode allows users to interact directly with the symbolic state of each reported error, which provides valuable insights and enables proper debugging. However, there are several tasks that we left for future work.

Choosing an SMT Solver. SMT solvers are not complete and might run into a timeout. Since every SMT solver works differently, it is possible that a statement can be proven by some solvers but not by others. Therefore, adding the possibility to choose an SMT solver (for example, Z3 [3] and CVC5 [1]) on-the-fly will be beneficial. In the `SiliconDebugger` we already added some code that initializes a prover. What remains to do is adding the interaction option and changing the prover in the decider accordingly.

Testing. So far, we only tested manually by looking at the state of the verifier or debugger and comparing it to what we expect it to be. Implementing tests that check some consistency properties is left for a future project. One interesting consistency property is the following: evaluating a final expression (using the debugging classes) should result in the term that it is linked with. This should be checked in several places, namely in the store, the evaluator, the path condition stack layers and the chunks. Another test should verify that all terms that have been collected as assumptions also appear as terms in the `DebugExp` assumptions. Ideally, this should hold for each path condition stack layer. Then it would be nice to verify that the final expressions do not contain any variables without versions and that all field accesses are wrapped in a `LabelledOld` expression. For the debugging mode, one could test whether the correct scope has been computed and that some basic user inputs can be handled.

UI Debugger. At the moment, the debugger is a command line tool. However, it would be nice to have a UI Debugger which is easy to use. Implementing this is left for a future project.

Simplifying the Final Expressions. When a term is created, some simple heuristics are used to simplify them, for example, $true \Rightarrow x > 0$ would be simplified to $x > 0$. Implementing similar heuristics for AST expressions would increase the readability and simplicity of the final expressions. Moreover, it would simplify comparing terms and final expressions.

References

[1] Haniel Barbosa et al. "CVC5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[2] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[3] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[4] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

[5] Malte Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, Zürich, Switzerland, 2016. DOI: 10.3929/ethz-a-010835519.

A Viper Program Example

The Viper program used to demonstrate the interactive debugging mode in Figures 1-4 can be found in Listing A. Neither of the two methods can be verified.

```
field val : Int

method gaussian_sum(n : Ref)
    requires ge0(n)
{
    unfold ge0(n)

    var i : Int := 0
    var r : Int := 0

    while(i < n.val)
        invariant acc(n.val)
        invariant r == i * (i-1) / 2
        invariant i <= n.val
        invariant 0 <= n.val
    {
        r := r + i
        i := i + 1
    }

    assert r == n.val * (n.val+1) / 2
}

predicate ge0(n : Ref)
{
    acc(n.val) &&  n.val >= 0
}

method testRef()
{
    var x : Ref
    x := new() // missing permission
    var y : Int
    x.val := 2
    y := x.val + y
    assert y >= x.val
}
```