A Better SMT Language: Design & Tooling

Bachelor Thesis Project Description
Department of Computer Science
ETH Zürich

Nico Darryl Hänggi
supervised by Dr. Malte Schwerhoff

April 16, 2020

1. Introduction

In the last decade, SMT solvers have become an essential part of computer-aided verification and analysis of software. In order to prove the correctness of a program, SMT solvers leverage the power of satisfiability modulo theories (SMT), which form a generalization of the Boolean satisfiability problem (SAT) by extending it with first-order logic theories. SMT problems are expressed in first-order logic formulas, for which a SMT solver determines its satisfiability. While there exist numerous different SMT solvers, the Z3 SMT solver [5] is among one of the most popular ones. In Z3, formulas are described using the SMT-LIB2 standard [2], which features a simple syntax consisting of Lisp-style s-expressions.

The Z3 SMT solver forms the basis of many program verifiers, including Boogie [1], F* [7] and the Viper verification infrastructure [6] developed at ETH Zürich. The Viper verification infrastructure, for instance, introduces an intermediate language called Viper that provides simple imperative constructs, as well as statements for managing permission-based reasoning. Silicon [4], one of Vipers automated verifiers, translates these programs into logical queries, which are then discharged using Z3.

However, these program verifiers are rendered significantly less useful if they introduce bugs in the SMT encoding. Unfortunately, finding those bugs is extremely challenging, because the encoding is not very human-friendly. To this end, the project is concerned with reducing the encoding gap between what the high-level program code actually does and how said code is represented in SMT-LIB2. To accomplish this, we introduce a novel SMT language.

2. Problem statement

Developers of program verifiers often need to understand and modify the generated SMT-LIB2 encodings in order to reason about the correctness of the translated resource assertions and to experiment with different SMT encodings. However, this task is complicated in two major ways.

First of all, SMT-LIB2 features a simple syntax composed of Lisp-style s-expressions. Due to the fully parenthesized prefix notation, parsing SMT-LIB2 is very easy and extremely efficient. In particular, for a given reference `x` and a Set `xs`, consider the s-expression given in Listing 1.

```
1 (assert (implies (in x xs) (and (acc(x.f)) (x.f > 10))))
```

Listing 1: A Viper assertion in prefix notation, ensuring that if `x` is an element of `xs`, then the permission to the field `x.f` is being held and its value is greater than `10`.

However, as can be observed, because of the form of s-expressions, the encoded logical assertion that needs to be proven is obscured and the focus of the SMT code instead lies on the Lisp-style syntax.

Secondly, the generated SMT encoding is represented using a language of first-order logic and hence requires to express foundational concepts of programming languages such as reading or writing variables. The lack of a simple encoding for such basic language concepts further complicates the resulting SMT code and leads to a large encoding gap between the program code to be verified and the SMT-LIB2 encoding.

As a result, the SMT encoding generated by program verifiers is usually quite hard to understand and without a deep understanding of the tool that generated said encoding, it's almost impossible for tool implementors to work with and reason about the resulting encoding. To make matters worse, oftentimes the generated SMT files have a length exceeding tens of thousands of lines of code.

3. Solution approach

We now propose a solution approach that tries to eliminate these issues by introducing a novel SMT language that supports efficient conversion from and to SMT-LIB2. This new language will be designed in two main steps: First, a more human-friendly SMT syntax will be chosen, drawing inspiration from existing languages such as Boogie and Viper. Secondly, we will enable users to customize parts of the new syntax to account for verifier-specific encodings. In the following, we will explain both steps in greater detail.

3.1. Design of a new SMT language

As hinted at by Listing 1, reasoning about s-expressions can be rather tedious. Working with a SMT language that is closer to the intermediate verification language of a program verifier is desired, as it shifts focus towards the logical assertions present in the program. Thus, in a first step, we design a more human-friendly SMT language that allows us to transform back and forth between SMT-LIB2 and our more convenient SMT language. In order to allow for a broad application of our syntax, we ensure that it is not tool-specific.

We will now illustrate this in a simple example. For instance, consider the SMT-LIB2 encoding shown in Listing 2, which is used to define a new variable `vs` by concatenating the three lists `xs`, `ys` and `zs`.

```
1 (assert (equal vs (concat xs (concat ys zs))))
```

Listing 2: Simple SMT-LIB2 encoding of a nested list concatenation.

Listing 3 shows an example of what the syntax of this new SMT language could look like when applied to Listing 2. This representation forms the basis for potential extensions that enable customization of the language syntax.

```
1 assert vs = concat(xs, concat(ys, zs))
```

Listing 3: Representation of Listing 2 using our human-friendly SMT language.

3.2. Language syntax customization

So far, we have not accounted for any verifier-specific encodings in our new SMT language. Consequently, a large part of our encoding consists of logical assertions that could be implicitly expressed as part of the language syntax. In order to allow for this possibility, users will be able to customize parts of our new SMT language syntax.

```
1 vs := concat(xs, ys, zs)                    /* option 1 */
2 vs := xs ++ ys ++ zs                        /* option 2 */
```

Listing 4: Two different textual representations for incorporating the fact that `concat` is associative into the language syntax.

For instance, consider Listing 3 once again. For now, let us focus on the nested method call to `concat`. It is obvious that the method `concat` is associative, as it only appends one list to the end of the other. Therefore, we want to leverage the associativity of `concat` and eliminate the purely syntactical differences between otherwise equivalent expressions. To do so, we want to enable

the user to define a customization for the language syntax by representing these nested method calls equally. In Listing 4, we have shown two out of many other possible textual representations to achieve an equivalent syntax. For a more in-depth analysis and conversion of an actual Viper-generated SMT encoding, please refer to the example illustrated in *Appendix A*.

4. Core Goals

The core goal of this project is to design an alternative syntax for SMT encodings that helps implementors of program verifiers to understand and reason about the underlying SMT code more easily.

- **Design an improved source syntax.** Based on inspiration from related languages such as Viper and Boogie, design an improved and more convenient source syntax for the SMT language. The novel source syntax shall be designed in a way to allow conversion from and to SMT-LIB2. Test the capabilities of this new syntax by evaluating it on a list of Viper-generated SMT files and adapt the language if required.

- **Develop CLI tool support.** Investigate possible high-performance implementations for parsers and pretty-printers, such that SMT encodings with tens of thousands of lines of code can be handled efficiently and quickly. Based on these findings, implement a high-performance parser and pretty-printer for the new SMT source syntax. The architecture of the CLI tool is designed in such a way that extending the tool with a future Visual Studio Code [3] plugin is facilitated. Furthermore, the CLI tool must support automated conversion from and to SMT-LIB2.

- **Evaluate tool performance.** The performance of the CLI tool is evaluated under a set of well-defined examples and possible culprits are identified. In a next step, the performance of the tool is profiled in regards to the SMT encodings generated by Silicon in order to guarantee real-life usefulness.

- **Conceptualizing customizable language syntax.** Analyse real-world SMT encodings and compile a list of potential language features that we wish to be able to customize. Next, extend the SMT language to support different textual representations for the collected features (cf. *3.2 Language syntax customization*). In addition, the architecture of the CLI tool should be extended *conceptually*, such that it is possible for its users to define a custom syntax for certain language features, which both parser and pretty-printer then adhere to.

5. Extension Goals

- **Implementing customizable language syntax.** Implement the general architecture for supporting different textual representations according to the specification. On top of that, implement the custom syntax representation for a subset of all the collected language features identified earlier.

- **Adding Viper-specific textual representation.** After having implemented the foundation for a custom syntax, identify and implement Viper-specific language syntax features we want to support.

- **Building a Visual Studio Code Extension.** While the CLI tool is sufficient for basic usage, providing a Visual Studio Code extension in order to further streamline the development process would be highly beneficial for increased user experience. Potential extension features for the novel SMT language include:
  - syntax highlighting
  - direct integration of useful analysis features (e.g. data-flow analysis) along with possible visualizations
  - on-the-fly conversion from and to SMT-LIB2, directly invoking the Z3 SMT solver

References

[1] Mike Barnett et al. "Boogie: A modular reusable verifier for object-oriented programs". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 364–387.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The SMT-LIB standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[3] Microsoft Corporation. *Visual Studio Code*. URL: `https://code.visualstudio.com/` (visited on Apr. 14, 2020).

[4] Schwerhoff Malte. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". Ph.D. Thesis. ETH Zürich, 2016.

[5] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: vol. 4963. Apr. 2008, pp. 337–340.

[6] Peter Müller, Malte Schwerhoff, and Alexander J Summers. "Viper: A verification infrastructure for permission-based reasoning". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2016, pp. 41–62.

[7] Nikhil Swamy et al. "Verifying Higher-order Programs with the Dijkstra Monad". In: *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI '13. 2013, pp. 387–398.

A. Illustrating example using Silicon

After having demonstrated the core issues regarding SMT-LIB2 encodings with a simple example, we will now explore some of the difficulties we have to face when working with and modifying an actual SMT-LIB2 encoding generated by Silicon. We illustrate this with the help of the Viper verification infrastructure. Consider the following Viper program in Listing 5, which is used to express permissions to an unbounded number of heap locations using quantified permissions.

```
1  field f: Int
2
3  method foo(xs: Set[Ref], y: Ref, z: Ref)
4    requires forall x: Ref ::
5      x in xs ==> acc(x.f) && x.f == 10
6    requires y in xs
7    requires z in xs
8  {
9    y.f := y.f - 1
10
11   var v: Int := z.f
12   assert y == z ==> v == 9
13 }
```

Listing 5: A simple Viper program. The first precondition denotes a quantified permission and ensures access permissions to all the elements in the set, as well as ensuring a field value of `10` for all elements in the set. The second and third precondition denote that references `y` and `z` are part of the set.

In this case, both reading and writing to `y.f` and `z.f` succeeds because the preconditions guarantee exclusive permissions to all elements in the set. However, decreasing the value that is stored in the heap location `y.f` by `1` could lead to also updating the location `z.f` at the same time, because `y` and `z` might be aliases. Thus, whenever `y` and `z` are aliases, we know that after the statement on line 9 is executed, both `y.f` and `z.f` point to an address on the heap holding the integer value `9`. Therefore, Silicon is able to prove that the assertion $x = y \Rightarrow v = 9$ is valid and hence the method `foo` verifies successfully.

In Silicon, logical queries are discharged by generating SMT-LIB2 expressions that are given to the Z3 SMT solver. Let us consider a *simplified* excerpt of the SMT-LIB2 code shown in Listing 6, which is generated by Silicon during the program verification step of Listing 5. In lines 1 to 4, the arguments to the `foo` method are declared along with a snapshot map to represent the initial mapping between receiver expressions and their partial heap values. For the sake of simplicity, the SMT encoding for the preconditions of the method `foo` are omitted. We assume the required assertions to ensure the preconditions to hold. In lines 8 to 10, the update to the heap representation is reflected by declaring a new snapshot map and then ensuring that the field lookup of `y` returns its old value decreased by `1`. The assertion declared in lines 11 to 15 ensures that the values of all other heap locations are framed across the assignment, and hence remain unchanged. Furthermore, lines 17 to 19 then declare a new variable `v` and bound its value to the value stored in the field `f` of variable `z`. Finally, in lines 21 to 22, $!(x = y \Rightarrow v = 9)$ is asserted, which allows Z3 to verify the correctness of our Viper program.

```
1  (declare-const xs Set<Ref>)
2  (declare-const y Ref)
3  (declare-const z Ref)
4  (declare-const sm@1 SnapshotMap)
5
6  ; ... assuming preconditions are asserted
7
8  ; y.f := y.f - 1
9  (declare-const sm@2 SnapshotMap)
10 (assert (= (lookup_f sm@2 y) (- (lookup_f sm@1 y) 1)))
11 (assert (forall ((r Ref)) (
12   (implies
13     (not (= r y))
14     (= (lookup_f sm@2 r) (lookup_f sm@1 r)))
15   )))
16
17 ; v := z.f
18 (declare-const v Int)
19 (assert (= v (lookup_f sm@2 z)))
20
21 ; y == z ==> v == 9
22 (assert (not (implies (= y z) (= v 9))))
```

Listing 6: Simplified SMT-LIB2 encoding for proving correctness of method `foo` in Listing 5.

The resulting SMT encoding in Listing 6 is quite hard to understand and without profound knowledge about how Silicon encodes logical queries, its meaning is heavily obscured. To exacerbate these issues, for the sake of illustration purposes, Listing 6 is shown in an extremely simplified form. The actual encoding is much more intricate and will be even harder to understand and reason about.

In order to eliminate the aforementioned issues, we introduce a more human-friendly SMT language. Transforming the encoding from Listing 6 to said language gives us the code shown in Listing 7.

```
var xs: Set[Ref]
var y: Ref
var z: Ref
var sm@1: Snapshot

// ... assuming preconditions are asserted

var sm@2: Snapshot
assume lookup_f(sm@2, y) == lookup_f(sm@1, y) - 1
assume forall r: Ref :: r != y
  ==> lookup_f(sm@2, r) == lookup_f(sm@1, r)

var v: Int
assume v == lookup_f(sm@2, z)
assert !(y == z ==> v == 9)
```

Listing 7: Transformation of the SMT-LIB2 encoding shown in Listing 6 into a more human-friendly SMT language.

While our novel representation is already much simpler to understand, it still encodes basic concepts about imperative programs. In a final step, we are able to simplify our SMT excerpt even further by leveraging our knowledge about how Silicon handles heap representations. The resulting SMT code is shown in Listing 8.

```
var xs: Set[Ref]
var y: Ref
var z: Ref

// ... assuming preconditions are asserted

y.f = y.f - 1

var v: Int = z.f
assert y == z ==> v == 9
```

Listing 8: Resulting SMT encoding by applying our understanding of Silicon heap representations to the SMT code shown in Listing 7.

As can be observed, the final SMT encoding is almost identical to the original Viper program from Listing 5. While achieving such a best-case scenario is the ultimate goal, we do not expect to be able to reproduce such a clean representation for all possible language concepts.