Smten: Automatic Translation of High-level Symbolic Computations into SMT Queries

Richard Uhler[1], Nirav Dave[2]

[1] Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA, ruhler@csail.mit.edu

[2] SRI International, Computer Science Laboratory, Menlo Park, CA, USA, ndave@csl.sri.com

Smten: A Meta-Tool

For *developing* SMT-based tools

Developer directly expresses high-level translation concerns

Smten automatically generates optimized translation

Leads to flexible, high performance SMT-based tools with greatly reduced developer effort

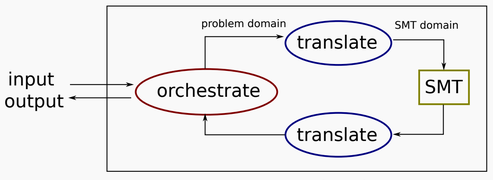
In our case study, Smten reduced lines of code by a factor of 20 while achieving performance comparable to hand crafted translation

Motivation: SMT-Based Tools

Leverage Satisfiability Modulo Theories (SMT) solvers for computer aided verification tasks

Uses include:

- model checking
- program synthesis
- automated theorem proving
- automatic test generation
- software verification



SMT-Based Tool

Challenge: The Translation to SMT

High-level concerns in translation:

- What SMT solver should be used?
- How should high-level structures be represented in SMT?
- How should problem be decomposed into queries?

Implementing translation is tedious and error-prone

Optimized translation is crucial to a high-performance SMT-based tool:

- Generating large formulas is costly
- Transmitting large formulas to an SMT solver is costly

The Smten Language

Unified language for orchestration and symbolic computation

High-level, purely functional

Syntax and features borrowed heavily from Haskell

- Algebraic data types
- polymorphism
- general purpose input/output
- pattern matching
- type classes

Provides a primitive API (based on monads) for describing symbolic computations

Example: Translating String Constraints

High Level Problem:

```
var v : 4;
val v' := concat("f", v);
assert v' contains "foo";
assert v' contains "odd";
assert v' not contains "weird";
```

Translation concerns:

How to represent

- strings?
- chars?

Generated SMT Formula:

Represent char as bit vector

Variables: v1, v2, v3, v4 of type Bit #8

Use naming variable naming convention

Formula:

```
( (102 = 102 ∧ v1 = 111 ∧ v2 = 111)
∨ (v1 = 102 ∧ v2= 111 ∧ v3 = 111)
∨ (v2 = 102 ∧ v3 = 111 ∧ v4 = 111) )
```

Map char equality to bitvector equality

```
∧ ( (102 = 111 ∧ v1 = 100 ∧ v2 = 100)
∨ (v1 = 111 ∧ v2 = 100 ∧ v3 = 100)
∨ (v2 = 111 ∧ v3 = 100 ∧ v4 = 100) )
```

Should optimize away in translation!

```
∧ (¬ (102 = 119 ∧ v1 = 105 ∧ v2 = 101 ∧ v3 = 114 ∧ v4 = 100))
```

Result: SAT with v1 = 111, v2 = 111, v3 = 100, v4 = 100

Corresponds to v' = "foodd"

High-level Problem in Smten:

Specify SMT solver

Char representation

```
main = do
  result <- runSymbolic STP $ do
     v <- sequence (replicate 4 (free :: Symbolic (Bit #8)))
     let v' = fromStr "f" ++ v
     assert (contains v' (fromStr "foo"))
     assert (contains v' (fromStr "odd"))
     assert (not (contains v' (fromStr "weird"))
     return v'
  case result of
     Just x -> putStr $ "SAT: " ++ show (toStr v')
     Nothing -> putStr $ "UNSAT"
```

Case Study: Hampi

An existing SMT-based tool for solving string constraints

Has been successfully applied to testing and analysis of real programs:

- Analyses for SQL injections in web applications
- Automated bug finding in C programs

Implemented in Java, using STP SMT solver

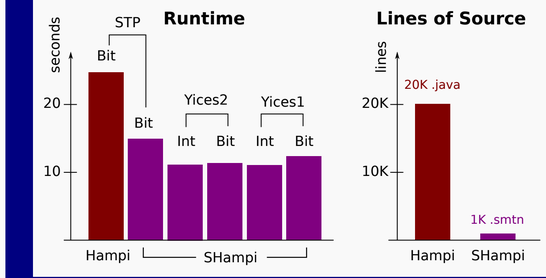
SHampi

A re-implementation of the Hampi tool using Smten

Easily expressed high level optimizations used in Hampi:

- Fixed sizing of CFGs
- Caching of submatch results

We explored 3 SMT solvers and 2 representations for characters



References

Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: Proceedings of the 18th international symposium on Software testing and analysis. ISSTA '09, New York, NY, ACM (2009) 105-116

Uhler, R., Dave, N.: Smten: Automatic Translation of High-level Symbolic Computation into SMT Queries. In: Proceedings of 25th International Conference on Computer Aided Verification. CAV '13, Saint Petersburg, Russia. (2013)

Acknowledgments

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 and supported by National Science Foundation under Grant No. CCF-1217498.

The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense