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

Richard Uhler[1] and 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

**Abstract.** Development of computer aided verification tools has greatly benefited from SMT technologies; instead of writing an ad-hoc reasoning engine, designers translate their problem into SMT queries which solvers can efficiently solve. Translating a problem into effective SMT queries, however, is itself a tedious, error-prone, and non-trivial task. This paper introduces Smten, a tool for automatically translating high-level symbolic computations into SMT queries. We demonstrate the use of Smten in the development of an SMT-based string constraint solver.

## 1 Introduction

As Satisfiability Modulo Theories (SMT) solvers mature, their use continues to grow across many domains, including model checking, program synthesis, automated theorem proving, automatic test generation, and software verification. A primary reason for the popularity of SMT is it removes the need for ad-hoc reasoning engines in each application in favor of a simpler translation to a well understood domain with high-performance solvers.

Translating a problem into effective SMT queries, however, is itself a tedious, error-prone, and non-trivial task required for each new SMT-based tool. To better understand the effort involved in translating a problem into SMT queries, we will discuss issues that arise in the translation of Hampi [1], a string constraint solver originally implemented using the STP [2] SMT solver.

The primary form of string constraint supported by the Hampi solver is regular expression match. Figure 1 presents a high-level description in a Haskell-like pseudo-code of what it means to perform regular expression matches in Hampi. This description is polymorphic in the character type. At a high level, the goal is to compute the `match` function symbolically, given a regular expression and symbolic string, to obtain a boolean formula representing membership of the

* 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.

```
data RegEx = Epsilon | Empty | Atom Char | Range Char Char
           | Star RegEx | Concat RegEx RegEx | Or RegEx RegEx

match :: (SChar c) ⇒ RegEx → [c] → Bool
match Epsilon str       = null str
match Empty   _         = False
match (Atom x) [c]      = toSChar x == c
match (Range lo hi) [c] = toSChar lo ≤ c && c ≤ toSChar hi
match r@(Star x) str    = null str || any (match2 x r) (splits [1..length str] str)
match (Concat a b) str  = any (match2 a b) (splits [0..length str] str)
match (Or a b) str      = match a str || match b str

match2 a b (sa, sb) = match a sa && match b sb
splits ns x = map (λn → splitAt n x) ns
```

**Fig. 1.** High-level regular expression match

string in the language of the regular expression. This boolean formula is used in the SMT query.

The translation of regular expression match into a formula is complicated by the fact that the STP SMT solver does not support strings or characters. The symbolic string must first be translated into something understood by the SMT solver, *e.g.,* as a collection of free bit-vectors. High-level string operations must also be translated to SMT-understandable operations. Some operations, such as comparing two characters, translate naturally to bit-vector comparison in the SMT formula. Others, like string length, depend entirely on how strings are represented in the SMT query. Choosing how to represent high-level data types and operations in a lower-level SMT formula is often tedious.

In practice, directly translating high-level data types and operations into SMT formulas and querying the solver is not necessarily efficient. For example, if we know the length of a substring being matched against part of a regular expression, we can restrict the number of alternatives considered, drastically reducing the SMT query size and consequently the solver runtime. Similar improvements can be achieved by exploiting the knowledge of known character values in the string during the translation. Adding special code to exploit these cases is non-trivial, and it is not always obvious when it will lead to a worthwhile improvement in the translation process.

We have implemented Smten, a tool for automatically translating high-level symbolic computations into efficient SMT queries. Smten minimizes the manual effort of implementing and optimizing ad-hoc translations into SMT queries, leading to simpler, more readable code, and increasing developer productivity.

We demonstrate Smten via a new implementation of the Hampi string constraint solver. Smten allowed us to easily identify and implement optimizations in the SMT query generation, resulting in performance comparable to the original implementation in 5% of the code size.

```
data Symbolic a
instance Monad Symbolic

assert :: Bool → Symbolic ()

runSymbolic :: Symbolic a → IO (Maybe a)
```

```
class Free a where
    free :: Symbolic a

instance Free Bool
instance Free Integer
instance Free (Bit #n)
```

**Fig. 2.** The Smten `Symbolic` monad

### Related Work

The value of augmenting SMT with general-purpose programming abstractions is well recognized [3–5] and can be found in multiple guises in the literature. SMT solvers, *e.g.,* Z3 [6] and Yices [7] add theories to improve the solver runtime, *e.g.,* record types and lambda terms; features which also raise the level of abstraction. For practical reasons, however, SMT developers have not devoted significant effort to abstractions aimed solely at improving the user's representation task, *e.g.,* modules, parametric and ad-hoc polymorphism, and metaprogramming.

In the context of using SMT solvers, multiple Domain Specific Embedded Languages (DSEL) [8] have been developed to hide the complexity of interacting with the SMT solver and provide a straightforward metaprogramming layer for SMT, *e.g.,* Haskell embeddings of Yices [9] and Z3 [10]. These allow a programmer to describe complicated SMT queries metaprogramatically using the host language. However, as these tend to focus on providing a syntactic bridge to the host language, they have issues naturally exposing SMT features which overlap with the host language abstractions, *e.g.,* user-defined data types.

Smten combines the metaprogramming of DSELs with the raised abstractions of SMT solvers to provide a more coherent user experience, allowing rich interactions between the two approaches.

## 2 Smten: Language and Implementation

The Smten input language is used to describe high-level symbolic computations for translation into SMT queries. Smten's input language is a strongly typed, purely functional language borrowing its syntax and many features from Haskell [11], including support for algebraic data types, pattern matching, polymorphism, and type classes. We chose to base the Smten language on Haskell because of its ability to concisely describe side-effect free computations. For a complete description of type classes, pattern matching, functions, and other Smten language features, we refer interested readers to the Haskell reference [11]. The remainder of this section is devoted to the `Symbolic` monad, the mechanism in Smten for managing symbolic computations.

Figure 2 summarizes the `Symbolic` monad in Smten. Computations in the `Symbolic` monad take place in the context of free variables and assertions. The primitives for using the `Symbolic` monad are described as follows:

`free` introduces a new free variable into the `Symbolic` context and returns an expression representing it. The expression returned can be used as a normal concrete expression in the Smten language. Smten provides primitive instances of `free` for types supported directly by the SMT solver and can automatically derive sensible, user-overloadable, instances of `free` for any bounded algebraic data types. Currently Smten provides primitive support for Bool, Integer, and Bit.

`assert` introduces a boolean assertion into the `Symbolic` context.

`runSymbolic` queries the solver to determine whether there exists an assignment to the free variables in the given symbolic object satisfying all assertions. If there is such an assignment, `runSymbolic` returns the value of its argument under that assignment, otherwise it returns `Nothing`.

Smten also provides primitives to enable incremental queries supported by many SMT solvers. These primitives are not discussed in this paper.

The high-level pseudocode for matching shown in Fig. 1 is valid Smten code. Given `match`, the `Symbolic` monad can be used to easily describe an SMT-based tool which accepts a length $i$ and regular expression as input, and uses an SMT solver to find a concrete string of length $i$ matching the regular expression:

```
main :: IO ()
main = do
  (len, regex) ← parseArgs
  result ← runSymbolic (qmatch len regex)
  case result of
    Just v → putStrLn v
    Nothing → putStrLn "no solution"

qmatch :: Integer → RegEx
                  → Symbolic String
qmatch len regex = do
  str ← sequence (replicate len free)
  assert (match regex str)
  return str
```

Note that the usage of the `match` function is the same whether the string argument is symbolic or concrete. Smten evaluates concrete inputs directly and generates SMT expressions for symbolic inputs.

The Smten tool compiles high-level descriptions of symbolic computations to Haskell using standard compilation techniques. Case expressions and primitive operations in the kernel language recognize concrete arguments and perform concrete evaluation wherever possible. This concrete evaluation removes objects with no primitive SMT support, such as lists and complex data structures, from the generated SMT query. Smten explicitly preserves dynamic sharing of expressions in the generated SMT queries.

As Smten generates Haskell, one can easily mix Haskell and Smten code, and, via Haskell's foreign function interface, other languages, *e.g.,* C or Java.

## 3 Implementing Hampi with Smten

Hampi is a string constraint solver whose constraints express membership of strings in both regular languages and fixed-size context-free languages. A Hampi input consists of regular expression and context free grammar (CFGs) definitions, bounded-size string variables, and predicates on these strings referencing

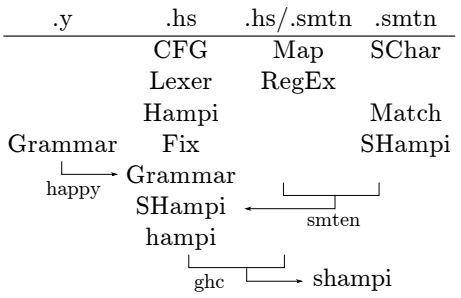


**Fig. 3.** SHAMPI source organization

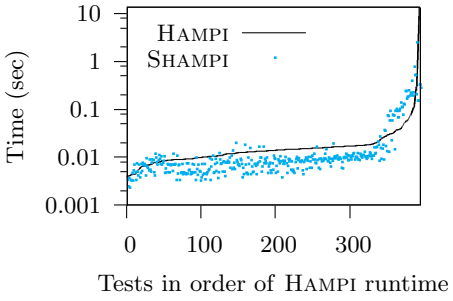


**Fig. 4.** HAMPI vs. SHAMPI runtime

the regular expressions and grammars. The output from HAMPI is a string which satisfies the constraints or a report that the constraints are unsatisfiable. The HAMPI tool has already been applied successfully to testing and analysis of real programs, most notably in static and dynamic analyses for SQL injections in web applications and automated bug finding in C programs using systematic testing. The original implementation of HAMPI was developed in about 20K lines of Java and uses the STP [2] SMT solver.

SHAMPI is our implementation of the HAMPI tool developed using Smten[3]. Figure 3 shows the organization of the source code for SHAMPI. We used the Happy parser generator to implement the HAMPI input parser. Much of the of the tool we left in Haskell, including the rest of the parser and fix-sizing of CFGs. The definition of RegEx is shared by both Haskell and Smten code. The match algorithm is implemented entirely in the Smten language. In total, our implementation of SHAMPI has a code base of 1030 lines.

Initially we used the naïve match algorithm from Fig. 1 for SHAMPI. To improve performance, we simplified CFGs by restricting them to match fixed-length strings. We further improved performance by caching boolean sub-match results in our match implementation. These optimizations, once understood, were implemented in Smten in a modest number of lines and in a matter of hours; replicating the same optimizations without Smten would have taken significantly more code and designer effort.

Figure 4 shows the performance of our implementation compared to the original implementation of HAMPI on all tests from the HAMPI distribution. For both SHAMPI and HAMPI, we took the best of 10 runs. SHAMPI was compiled with GHC-7.4.1 [12] and uses STP for solving SMT queries. We ran revision 46 of a single HAMPI server instance for all runs of all tests on HAMPI to ammortize startup cost. Even so, SHAMPI outperforms HAMPI on most tests, and is within a factor of 8 in the worst case. Smten also allowed us to easily experiment with using other solvers and representations for symbolic characters. Our best vari-

[3] Smten & SHAMPI source is at `http://people.csail.mit.edu/ruhler/shampi.tar.gz`

ant represented characters as integers and called the Yices-2.1.0 [13] solver; it slightly improves runtime overall and reduces the worst case overhead to a factor of 4.

## 4 Conclusion

SMT technologies greatly benefit developers, allowing them to share a small set of high-performance solvers rather than develop their own ad-hoc reasoning engine. Smten further extends this sharing from actual computations to the translation of problems into SMT queries. Smten allows developers to leverage existing effort and expertise in the difficult task of translating problems to effective SMT queries.

Our Shampi example clearly illustrates the value of Smten; Shampi closely matches the performance of the Hampi tool, while being only 5% of the code.

We believe Smten has great potential in further improving SMT-based tool construction. In the future, we plan to explore naturally describing counter-example guided queries and extend the portfolio approach of SMT solving across multiple sets of theories and solvers.

## References


1. 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
2. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: 19th International Conference on Computer Aided Verification (CAV-07). Volume 4590. (2007) 519–531
3. Kksal, A., Kuncak, V., Suter, P.: Scala to the power of z3: Integrating smt and programming. In Bjrner, N., Sofronie-Stokkermans, V., eds.: Automated Deduction CADE-23. Volume 6803 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2011) 400–406
4. Agarwal, S.: Functional SMT solving: A new interface for programmers. Master's thesis, Indian Institute of Technology Kanpur (June 2012)
5. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (December 2010)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-08). Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337–340
7. Dutertre, B., Moura, L.D.: The yices smt solver. (2006)
8. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: In Proceedings of the 2nd Conference on Domain-Specific Languages, ACM Press (1999) 109–122
9. Stewart, D. http://hackage.haskell.org/package/yices-painless-0.1.2 (2011)
10. Erkok, L. http://hackage.haskell.org/package/sbv-2.3 (July 2012)
11. Peyton Jones, S.: Haskell 98 Language and Libraries: the Revised Report. (2003)
12. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`
13. http://yices.csl.sri.com/index.shtml (August 2012)