ScalaSMT: Satisfiability Modulo Theory in Scala (Tool Paper)

Franck Cassez and Anthony M. Sloane
Programming Languages and Verification Research Group
Department of Computing, Macquarie University, Sydney, Australia
Franck.Cassez@mq.edu.au,Anthony.Sloane@mq.edu.au

Abstract

A Satisfiability Modulo Theory (SMT) solver is a program that implements algorithms to automatically determine whether a logical formula is satisfiable. The performance of SMT solvers has dramatically increased in the last decade and for instance, many of the state-of-the-art software analysis tools heavily rely on SMT solving to analyse source code. We present ScalaSMT, a Scala library that leverages the power of SMT solvers and makes SMT solving directly usable in Scala. ScalaSMT provides seamless access to numerous popular SMT solvers like Z3, CVC4, Yices, MathSat or SMTInterpol. Our library comes with a domain-specific language to write terms and logical formulas for a wide range of logical theories, thereby isolating users from the details of particular solvers.

CCS Concepts • **Software and its engineering → Semantics**;

Keywords Scala, SMT solvers, logics

ACM Reference Format:
Franck Cassez and Anthony M. Sloane. 2017. ScalaSMT: Satisfiability Modulo Theory in Scala (Tool Paper). In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3136000.3136004

1 Introduction

Satisfiability modulo theory (SMT) consists of determining the satisfiability of logical formulas. It can reason in various formal theories, e.g., in linear integer or real arithmetic, first-order logic, or logics of arrays. An SMT solver is a program that implements the corresponding algorithms to automatically determine whether a logical formula is satisfiable. The SMTLIB initiative[1] provides a common input and output format based on S-expressions for interacting with SMT solvers. We present ScalaSMT, a Scala library that provides an abstraction over the SMTLIB format. The library brings consistency and type safety to the textual and dynamically typed world of SMTLIB solver interaction.

ScalaSMT relies on the SMTLIB input/output capabilities of the solvers and consequently provides access to numerous popular SMTLIB-compliant solvers such as Z3 [de Moura and Bjørner 2008], CVC4 [Deters et al. 2014], Yices [Dutertre 2014], MathSat [Cimatti et al. 2013] or SMTInterpol [Christ et al. 2012]. ScalaSMT is easily extendable (SMTLIB commands and theories can be added) and configurable (SMTLIB-compliant solvers can be added). ScalaSMT fills a gap in the Scala library landscape by providing support for powerful logical reasoning algorithms. ScalaSMT is open source and available from https://bitbucket.org/franck44/scalasmt.

[1]http://smtlib.cs.uiowa.edu

SCALA'17, October 22–23, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17)*, https://doi.org/10.1145/3136000.3136004.

2 ScalaSMT Example Usage

In this section we introduce ScalaSMT using simple examples and discuss the advantages over SMTLIB.

2.1 Integer and Real Arithmetic

Assume you want to determine whether the logical formula $x + 1 \leq y \ \wedge \ (y \mod 4 = 3 \ \wedge \ y \geq 2)$ is *satisfiable* (SAT in the sequel). I.e., we want to know if we can assign integer values to x and y, say $v(x), v(y)$ such that $v(x) + 1 \leq v(y)$ and $v(y) \mod 4 = 3$ and $v(y) \geq 2$.

```
(set-option :produce-models true)
;; Set logic to Quantifier-Free Integer Arithmetic
(set-logic QF_LIA)
;; Define the integer variables x and y
(declare-fun x () Int)
(declare-fun y () Int)
;; Assert the formula x+1 ≤ y
(assert (<= (+ x 1) y))
;; Assert the formulas y  mod  4=3∧y≥2
(assert (and (= (mod y 4) 3) (>= y 2)))
;; Check SAT
(check-sat)
;; returns sat
;; Get values for x and y
(get-value (x y))
;; returns ((x 2) (y 3))
```

Listing 1. Simple SAT query in SMTLIB

Checking whether such an assignment v exists is a typical SAT query that can be answered by an SMT solver. Listing 1 shows how this example can be specified in the standard SMTLIB format. Running an SMTLIB compliant solver on Listing 1 will produce the result `sat` followed by a witness assignment that gives an instance of how the formulas are satisfiable, in this case `((x 2) (y 3))`.

With the domain-specific language (DSL) exposed by the ScalaSMT library, the query above can be written as shown in Listing 2 (we omit the `import` statements). Lines 1 and 2 define two objects of type `Ints` that corresponds to the SMTLIB type `Int` of the theory of Quantifier-Free Linear Integer Arithmetic `QF_LIA`. Lines 5 to 12 are in the scope of the `using` method which is provided by ScalaSMT (line 4). This method takes two arguments: a solver description (in this example a process running Z3) and initialisation I (theory set to `QF_LIA` and `MODELS` option to produce solution models); and a function literal f that maps a solver to a computation (`isSat`). The `using` method creates a solver s configured with I and calls $f(s)$ to compute the result. The solver only exists during the `using` call. When this method terminates (line 13) and we exit its scope, the process running Z3 is killed.

The SAT test is at lines 6 to 9 and the result is matched in lines 9 to 12 to decide if we can get a model (i.e., some values for x and y). The values of the variables can be extracted from a model m by using the `valueOf` method (e.g., `m.valueOf(x)`).

Listing 3 shows another example in the theory of Non-linear Real Arithmetic that uses `flatMap` to chain computations in a monadic manner. The method `|=` at line 7 asserts a term on the solver stack, and the result is a `Try` that can be forwarded to new computations (e.g., `checkSat`). Notice that if a command (lines 7 to 9) returns a `Failure`, the next command is not executed but the first `Failure` is the result of the full chain of commands.

2.2 Interpolants

The previous example showed how to retrieve values when an SMT query is SAT. It is also possible to retrieve predicates from an SMT query when a logical formula is unsatisfiable (UNSAT): they correspond to explanations for why the formula is UNSAT.

The example of Listing 4 shows how to do this using the `getInterpolants` method. Let $P_1 := x = z + 1 \wedge z \geq 0$ and $P_2 := y \geq x \wedge y < 1$. Then $P_1 \wedge P_2$ is clearly UNSAT as P_1 implies $x \geq 1$ but P_2 implies $x < 1$. For some theories, including Linear Integer Arithmetic, there is a predicate I, called an *interpolant*, such that the following conditions C_1 and C_2 are satisfied:[2]

C$_1$: $P_1 \implies I$, i.e., I is weaker (more general) then P_1, and
C$_2$: $I \wedge P_2$ is still UNSAT, i.e., I is good enough to be inconsistent with P_2.

The use of interpolants is instrumental in many software verification tools and we use it in our own static analyser Skink [Cassez et al. 2017]. The example in Listing 4 shows how to compute an interpolant, which in this case is $I = x \geq 1$, for $P_1 \wedge P_2$ and to check that indeed it is an interpolant (i.e., satisfies **C$_1$** and **C$_2$**). The interpolant is computed with an *interpolating* solver (SMTInterpol) and checked with a *non-interpolating* solver (CVC4) so this example illustrates how easy it is to combine the powers of different solvers.

[2]Another condition is that I has only free variables from P_1 and P_2.

2.3 Supported Theories

ScalaSMT supports a large number of SMTLIB theories including the common quantifier-free theories (`QF_LIA`, `QF_-LRA`, `QF_NRA`, `QF_UF`), the theory of Arrays (`QF_AUFLIA`, `QF_-AUFLIRA`), fixed-size (integer) bit-vectors (`QF_BV`), floating-point bit-vectors (`QF_FPBV`) as well as theories with quantifiers (e.g., `AUFNIRA`). An ecxample of usage of quantifiers is available in the `test` directory of the ScalaSMT repository https://bitbucket.org/franck44/scalasmt. Note that some theories are not supported by all the solvers. This is reflected in a configuration file that describes the capabilities of each solver to the library.

2.4 Solver Configurations

Addition of new solvers in ScalaSMT is through a configuration file and does not require modifying the Scala source code. A typical SMTLIB2-compliant solver (e.g., MathSat) is described as follows:

```
name = "MathSat"
executable = "mathsat"
version = "MathSAT5 v. 5.4.1"
args = []
timeout = 10 seconds
prompt1 = "(\\s)*\""
prompt2 = "\"(\\s)*(success)"
supportedLogics = [QF_UF, QF_LIA, QF_LRA,
    QF_AUFLIA, QF_BV, QF_ABV, QF_AUFLIRA]
supportedOptions = [PROOFS, MODELS]
```

The configuration comprises of the name of the executable (can be a full path), the arguments to be passed to use the solver in interactive mode, a default timeout for SMT-queries, the sets of supported logics and options. The `prompt*` fields are used to detect the end of a computation (see Section 3.2). Note that the timeout is enforced i.e., if the solver does not reply within the allocated time, the solver is killed. This is not a feature of ScalaSMT but of the Expect package we use to interact with the solvers (see Section 3.2).

2.5 Benefits of using ScalaSMT

Using a DSL for writing logical formulas has several advantages over writing them as *S-expressions* using standard SMTLIB input/output:

- Operators can be overloaded, which provides a natural way of expressing logical and formulas.
- A common error when writing SMTLIB specifications is to omit the `declare-fun` command for some variables. The use of ScalaSMT naturally enforces declarations prior to usage.

```
val x = Ints("x")        // Variables of type Int in SMTLIB, theory QF_LIA
val y = Ints("y")
val result : Try[ Model ] =
  using( new SMTSolver( "Z3", new SMTInit( QF_LIA, List( MODELS ) ) ) ) {
    implicit withSolver =>
        isSat(                      //  isSat returns a Try[ SatStatus ]
            x + 1 <= y,             //  use overloaded operators +, ≤, ≥
            y % 4 === 3 & y >= 2
        ) match {
            case Success( Sat() ) => getModel()
            case _                => Failure( new Exception( "failed" ) )
        }
  }
```

Listing 2. Example of Listing 1 using ScalaSMT

```
val b = Reals( "b" )      // Variables of type Real in SMTLIB, theory NRA
val c = Reals( "c" )
val model : Try[ Model ] =
  using( new SMTSolver( "Z3", new SMTInit( QF_NRA, List( MODELS ) ) ) ) {
    implicit solver =>
        //  assert  b^3 + b × c = 3, for b, c ∈ ℝ, then checkSat and getModel
        |= ( ( b * b * b ) + ( b * c ) === 3.0 ) flatMap
        { _ => checkSat() } flatMap      //  If |= is Success then checkSat
        { _ => getModel() }              //  If checkSat is Success then getModel
  }
```

Listing 3. Non Linear Real Arithmetic and "monadic" usage

```
// Predicates must be named in SMTLIB to refer to them and compute interpolant
val P1 = ( x === z + 1 & z >= 0 ).named( "P1" )
val P2 = ( y >= x & y < 1 ).named( "P2" )

val interpolants =
  using( new SMTSolver( "SMTInterpol", new SMTInit( QF_LIA, List( INTERPOLANTS )))) {
   implicit solver =>
     |= ( P1 & P2 ) flatMap              //  assert P1 and P2
     { _ => checkSat() } flatMap         //  If previous asserts are Success then checkSat
     { _ => getInterpolants( P1, P2 ) } //  If checkSat is Success then getInterpolants
}

// There should be only one interpolant
interpolants.isSuccess shouldBe true
interpolants.get.size shouldBe 1
val i = interpolants.get.head

// Check that i is an interpolant
using( new SMTSolver( "CVC4", new SMTInit( QF_LIA, List() ) ) ) {
  implicit solver =>
    //  Create a context on the solver's stack and assert first term
    push()
    // check P1 ⟹ I i.e., P1 ∧ ¬I is UNSAT
    isSat ( P1 & !i ) shouldBe Success( UnSat() )
    // Pop previous term from the stack
    pop()
    // check  I ∧ P2 is UNSAT
    isSat ( i & P2 ) shouldBe Success( UnSat() )
}
```

Listing 4. Computing and checking interpolants

- Another common problem with SMTLIB S-expressions is syntax errors (e.g., due to missing parentheses). In such a case, the error messages issued by the solver are sometimes hard to decrypt especially when writing long nested formulas as S-expressions. ScalaSMT constructs correct (well-parenthesised) S-expressions.
- Finally, when using SMTLIB, type checking occurs at run-time. In contrast, ScalaSMT enforces consistent typing: a Scala program with `x === true` and `x` of type `Ints` will not type-check. In the latter case a useful error message is issued by the Scala compiler which greatly helps in fixing the problem.

ScalaSMT provides an interface to SMTLIB compliant solvers together with a DSL and SMT-solvers can be used as back ends inside other tools: An example is our static analysis tool, Skink [Cassez et al. 2017], which uses ScalaSMT and solvers with different capabilities (e.g., Z3, SMTInterpol)

3 Architecture of ScalaSMT

An overview of ScalaSMT's architecture is provided in Figure 1. The current version supports five SMTLIB-compliant solvers: Z3[3], SMTInterpol[4], Yices[5], MathSat[6] and CVC4[7]. It comprises 99 source files and 10402 lines of code (LOC).

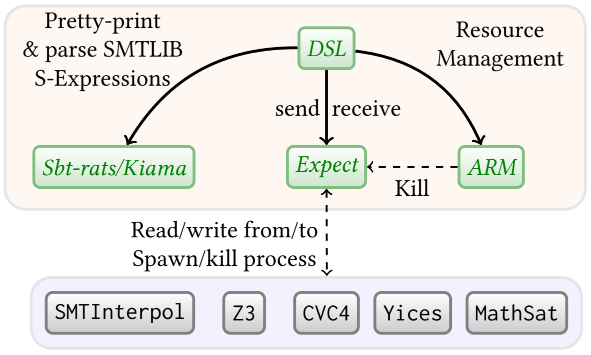

Figure 1. ScalaSMT Architecture

The main challenges in designing a robust, configurable and maintainable ScalaSMT library are:

1. reading/writing S-expressions from/to the solver;
2. interacting with an asynchronous external solver process from Scala code.

To address these challenges, ScalaSMT uses some Scala libraries and an SBT plugin that we described hereafter.

[3]https://github.com/Z3Prover/z3
[4]https://ultimate.informatik.uni-freiburg.de/smtinterpol/
[5]http://yices.csl.sri.com
[6]http://mathsat.fbk.eu
[7]http://cvc4.stanford.edu/

3.1 Sbt-rats and Kiama

Our parser and pretty-printer for SMTLIB S-expressions is written using our sbt-rats[8] parser generator [Sloane et al. 2016] and our Kiama[9] Scala library for language processing [Sloane 2009]. The sbt-rats SMTLIB syntax definitions in ScalaSMT has 500 LOC, with much simpler and maintainable definitions than the equivalent hand-coded counterparts in another similar project Scala-SMTLIB.[10] The generated parser implementation in Java has 11868 LOC and the Scala pretty-printer 408 LOC. The generated syntax-tree classes comprise of 269 LOC.

Our SMTLIB parser builds an abstract syntax tree (AST) that can be processed using Kiama rewriting strategies and annotated using Kiama attributes. This enables us to perform analysis and transformation on the AST, e.g., to construct S-expressions, or to compute properties such as variable names, or whether an AST term uses just linear arithmetic, or whether it contains quantifiers.

3.2 Expect

The main problem to address when programmatically interacting with an external asynchronous process (the solver) is to detect that the output is ready to collect. This means we need to be able to determine when the solver has finished its computation for a given input. This problem of interaction with an asynchronous process has long been identified and general solutions like *Expect*[11] have been devised. The idea of *Expect* is to send some input to an interactive program and detect the end of a computation (output ready) by detecting the *prompt* of the program on the output channel. A typical interaction is *Expect* runs as follows:

```
// send some input msg to the interactive program
send(input)
// expect to see a string `output', followed by a
// "prompt", within 10 seconds
output = expect(prompt, 10.seconds)
```

We have developed a Scala package Expect[12] which provides a simple Scala implementation of *Expect* in 76 LOC. Our implementation provides the `send` and `expect` methods and uses Java's `ProcessBuilder`, `Future` and `Scanner`.

3.3 Automatic Resource Management

An issue that is sometimes overlooked when spawning processes is the proper release of acquired resources. For instance, ScalaSMT spawns a solver process and attaches to it via *pipes*. Both the process and the pipes are resources (e.g., file descriptors) of the host OS. As per the documentation of the Java's `ProcessBuilder` package, it is not guaranteed that resources are released when the Java program terminates. The user is supposed to kill the processes spawned.

[8]https://bitbucket.org/inkytonik/sbt-rats
[9]https://bitbucket.org/inkytonik/kiama
[10]https://github.com/regb/scala-smtlib
[11]http://expect.sourceforge.net
[12]https://bitbucket.org/franck44/expect-for-scala

To address this issue, ScalaSMT uses the Scala-ARM[13] (Automatic Resource Management) library. This provides safe resource management and enforces that the resources (solvers and pipes) that are spawned by the library are properly released after usage.

4 Discussion

There are other approaches and implementation of SMTLIB in other languages like Haskell, Z3py. The Scala implementations similar to ours are:

- Scala-SMTLIB[14] operates similar to ScalaSMT via interactive solvers processes. It does not offer a DSL, nor safe resource management. The pretty-printer/parser is hand-coded, which makes it harder to extend and add new features or commands.
- Scala SMTLIB Interface[15] also spawns an interactive solver process. It has a hand-coded parser/pretty-printer. The DSL offers limited support for some theories like Reals, Ints.
- ScalaZ3[16] offers Scala bindings to Z3 only. It does not support other solvers.

Our experience in using ScalaSMT in our static analysis tool Skink [Cassez et al. 2017] shows that it greatly simplifies the code to write to generate and solve SMT queries. ScalaSMT also allows us to easily select the concrete solver depending on the features of an SMT query.

The use of sbt-rats to generate the SMTLIB parser/pretty-printer makes it easy to extend ScalaSMT and add new theories or commands.

We have not had space to describe some other interesting features of ScalaSMT. For instance, ScalaSMT maintains a stack of symbol declarations that is not available as an SMTLIB command. This enables us to simplify the DSL: e.g., the SAT query at lines 6 to 9 in Listing 2 does not require the user to *declare* the variables x, y to the solver before using `isSat`; the necessary SMTLIB declarations (`declare-fun` as per Listing 1 lines 5 and 6) are computed automatically by `isSat` and pushed to the solver prior to asserting the predicates.

In its current state, ScalaSMT interacts with the solvers via standard input and output. However, our implementation is ready to be interfaced with *in-memory* solvers: for instance, Z3, SMTInterpol offer Java bindings to use the solvers library directly. There is also a unified Java interface[17] [Karpenkov et al. 2016] for these solvers that provides a good basis for adding in-memory capabilities to our implemetation. This will improve the responsiveness of our abstract solver by decreasing communication overheads.

[13]https://github.com/jsuereth/scala-arm
[14]https://github.com/regb/scala-smtlib
[15]https://github.com/dzufferey/scala-smtlib-interface
[16]http://lara.epfl.ch/w/ScalaZ3
[17]https://github.com/sosy-lab/java-smt

References

Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo González de Aledo Marugán. 2017. Skink: Static Analysis of Programs in LLVM Intermediate Representation - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10206. Springer, 380–384. https://doi.org/10.1007/978-3-662-54580-5_27

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings (Lecture Notes in Computer Science)*, Alastair F. Donaldson and David Parker (Eds.), Vol. 7385. Springer, 248–254. https://doi.org/10.1007/978-3-642-31759-0_19

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. 2014. A tour of CVC4: How it works, and how to use it. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014.* IEEE, 7. https://doi.org/10.1109/FMCAD.2014.6987586

Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. 2016. JavaSMT: A Unified Interface for SMT Solvers in Java. In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 139–148. https://doi.org/10.1007/978-3-319-48869-1_11

Anthony M. Sloane. 2009. Lightweight Language Processing in Kiama. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (Lecture Notes in Computer Science)*, João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva (Eds.), Vol. 6491. Springer, 408–425. https://doi.org/10.1007/978-3-642-18023-1_12

Anthony M. Sloane, Franck Cassez, and Scott Buckley. 2016. The Sbt-rats Parser Generator Plugin for Scala (Tool Paper). In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016)*. ACM, New York, NY, USA, 110–113. https://doi.org/10.1145/2998392.3001580