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 [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.

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

\[ x + 1 \leq y \land (y \ mod \ 4 = 3 \land y \geq 2) \]

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 \( v(y) \geq 2 \).

Listing 1. Simple SAT query in SMTLIB

1. `\( \text{set-option :produce-models true} \)`
2. `\( ; ; \quad \text{Set logic to Quantifier-Free Integer Arithmetic} \)`
3. `\( \text{set-logic QF_LIA} \)`
4. `\( ; ; \quad \text{Define the integer variables } x \text{ and } y \)`
5. `\( \text{declare-fun x () Int} \)`
6. `\( \text{declare-fun y () Int} \)`
7. `\( ; ; \quad \text{Assert the formula } x+1=y \)`
8. `\( \text{assert (<= (+ \( x \) \( 1 \) \) \( y \))} \)`
9. `\( ; ; \quad \text{Assert the formulas } y \mod 4 = 3 \land y \geq 2 \)`
10. `\( \text{assert (and (eq (mod \( y \) 4) 3) (\( \geq \) \( y \) 2)))} \)`
11. `\( ; ; \quad \text{Check SAT} \)`
12. `\( \text{check-sat} \)`
13. `\( ; ; \quad \text{returns sat} \)`
14. `\( ; ; \quad \text{Get values for } x \text{ and } y \)`
15. `\( \text{get-value (x y))} \)`
16. `\( ; ; \quad \text{returns ((x 2) (y 3))} \)`
Checking whether such an assignment υ 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 (the 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 Nonlinear 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 \(P_1 := x = z + 1 \land z \geq 0\) and \(P_2 := y \geq x \land y < 1\). Then \(P_1 \land P_2\) 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:

\[
C_1: P_1 \implies I, \text{ i.e., } I \text{ is weaker (more general) then } P_1, \text{ and } \\
C_2: I \land P_2 \text{ is still UNSAT, i.e., } I \text{ is good enough to be } \\
\text{inconsistent with } P_2.
\]

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 \(P_1 \land P_2\) 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.

### 2.3 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\_AUF\_LIA, QF\_AUF\_LRA), fixed-size (integer) bit-vectors (QF\_BV), floating-point bit-vectors (QF\_FPBV) as well as theories with quantifiers (e.g., AUF\_NIRA). An example 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:

1 name = "MathSat"
2 executable = "mathsat"
3 version = "MathSAT5 v. 5.4.1"
4 args = []
5 timeout = 10 seconds
6 prompt1 = "(\ s) *"
7 prompt2 = "(" (success)"
8 supportedLogics = [QF\_UF, QF\_LIA, QF\_LRA, QF\_AUF\_LIA, QF\_BV, QF\_ABV, QF\_AUF\_LRA]
9 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.
1 val x = Ints("x") // Variables of type Int in SMTLIB, theory QF_LIA
2 val y = Ints("y")
3 val result : Try[ Model ] =
4 using( new SMTSolver("Z3", new SMTInit( QF_LIA, List( MODELS ) )) ) {
5 implicit withSolver =>
6 isSat( // isSat returns a Try[ SatStatus ]
7 x + 1 <= y, // use overloaded operators +, <=
8 y % 4 === 3 & y >= 2
9 ) match {
10 case Success( Sat() ) => getModel()
11 case _ => Failure( new Exception( "failed" ) )
12 }
13 }

Listing 2. Example of Listing 1 using ScalaSMT

1 val b = Reals("b") // Variables of type Real in SMTLIB, theory NRA
2 val c = Reals("c")
3 val model : Try[ Model ] =
4 using( new SMTSolver("Z3", new SMTInit( QF_NRA, List( MODELS ) )) ) {
5 implicit solver =>
6 // assert b^3 + b * c = 3, for b, c \in \mathbb{R}, then checkSat and getModel
7 |= ( ( b * b * b ) + ( b * c ) === 3.0 ) flatMap
8 { _ => checkSat() } flatMap // If |= is Success then checkSat
9 { _ => getModel() } // If checkSat is Success then getModel
10 }

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

1 // Predicates must be named in SMTLIB to refer to them and compute interpolant
2 val P1 = ( x === z + 1 & z >= 0 ).named( "P1" )
3 val P2 = ( y >= x & y < 1 ).named( "P2" )
4 val interpolants =
5 using( new SMTSolver("SMTInterpol", new SMTInit( QF_LIA, List( INTERPOLANTS ))) ) {
6 implicit solver =>
7 |= ( P1 & P2 ) flatMap // assert P_1 and P_2
8 { _ => checkSat() } flatMap // If previous asserts are Success then checkSat
9 { _ => getInterpolants( P1, P2 ) } // If checkSat is Success then getInterpolants
10 }
11
12 // There should be only one interpolant
13 interpolants.isSuccess shouldBe true
14 interpolants.getSize shouldBe 1
15 val i = interpolants.getHead
16
17 // Check that i is an interpolant
18 using( new SMTSolver("CVC4", new SMTInit( QF_LIA, List() )) ) {
19 implicit solver =>
20 // Create a context on the solver's stack and assert first term
21 push() // check P_1 \implies I \iff, P_1 \land \neg I \text{ is UNSAT}
22 isSat( P1 & !i ) shouldBe Success( UnSat() )
23 // Pop previous term from the stack
24 pop() // check I \land P_2 \text{ is UNSAT}
25 isSat( i & P2 ) shouldBe Success( UnSat() )
26 }

Listing 4. Computing and checking interpolants
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.

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-parenthesized) S-expressions.

Finally, when using SMTLIB, type checking occurs at run-time. In contrast, ScalaSMT enforces consistent typing: a Scala program with x :: 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).

![ScalaSMT Architecture](image)

**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.

3.1 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:

```scala
// 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.

\(^3\)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/

\(^8\)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\textsuperscript{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**\textsuperscript{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.
- **SMTLIB INTERFACE**\textsuperscript{15} also spawns an interactive solver process. It has a hand-coded parser/prettyprinter. The DSL offers limited support for some theories like Reals, Ints.
- **ScalaZ**\textsuperscript{2,3}\textsuperscript{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/prettyprinter 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. This will extend allow users to interact with solvers with lower communication overheads.

References


