Integrating dReal into JavaSMT

Tomaz Mascarenhas

April 3, 2023

Contents

1 Introduction and Background

Context

Satisfiability Modulo Theories (SMT) is a generalization of the Boolean Satisfiability Problem (SAT), in which the underlying logic system is First Order Logic, instead of Propositional Logic. In addition to that, a set of theories is included, that is, a set of structures on which all the terms can be defined (as opposed to just the Booleans), and it is allowed to use operators of such structures. For instance, one could use as structure the real numbers and their comparison operators, making assertions about real variables. SMT solvers are systems specialized in solving this problem. Moreover, they do this automatically, with the only interaction with the user being the definition of the problem that they want to solve. It is possible to encode the problem of software verification as an instance of SMT, and that constitutes one of the main uses of SMT solvers.

Given the expressivity and efficiency of SMT solvers, there exists a wide variety of ways in which the user can communicate with them. In this project, we will be focusing on JavaSMT, which is an API for communicating with SMT solvers through the Java programming language. Using it, one can define variables over various kinds (boolean, functions, reals, etc.), assert properties about them, producing a SMT problem and dispatching it into one of the SMT solvers available in the API. The other common approach to query a solver with such a problem would be to write it using the SMT-LIB2 language, that is the standard format recognized by all the SMT solvers to describe those problems, and invoke a solver directly using the script as an input.

Using an API such as JavaSMT has the advantage of being more flexible than defining the problem directly in SMT-LIB2 syntax, in the sense that it is possible to incrementally construct the problem to be solved through interactions with the user and using features of the language, allowing a more algorithmic approach, whereas, using SMT-LIB2, the problem is completely static. In theory, one could write a Java program that uses the algorithmic approach to incrementally write a SMT-LIB2 script to be sent to the solver, but that is significanly more complicated than using an API such as JavaSMT.

Currently, JavaSMT is able to communicate with nine solvers: Boolector, CVC4, cvc5, MathSAT5, OptiMathSAT, Princess, SMTInterpol, Yices2 and Z3. One useful addition to this list would be dReal. dReal is a SMT solver focused on solving problems with real numbers efficiently. It is known that, using floating points to represent real numbers is a strategy susceptible to errors due to the limited precision that processors can operate on them. Despite that, they are significantly faster than the more used alternative, that is using a pair of integers to represent fractions and other mechanisms to represent irrational numbers. dReal's approach is to use floating point numbers and control the amount of error accumulated throughout the computations, providing values that are approximate to the correct answer, but allowing the user to set the maximum relative error present in those values. Once this solver is added to JavaSMT, users that want to solve problems in this realm are likely to obtain better solutions, as they would have access to a solver that was specially designed for solving this kind of problem.

2 Project Goals

The main deliverables of the project that I intend to provide are the following:

- A set of Java classes, implementing all SMT theories available in dReal using the JavaSMT interface, as well as all the options that the solver supports (such as flags for setting tolerance values on the functions).
- The shared libraries containing the tweaked API for dReal, to be linked into JavaSMT.
- A set of tests covering all the code produced, following the standards of the existing tests in JavaSMT.
- Clear and organized documentation, explaining all the decisions that were made during the implementation of the project and what is the final state of the integration.

3 Implementation

Some of the SMT solvers already have a Java API. In those cases, to integrate them into JavaSMT is enough to simply write a wrapper class into the existing API, matching the required methods in the interfaces of JavaSMT with the methods from the API and instantiating the type parameters from the classes in JavaSMT with the existing types in the API. This is the case for instance for Z3, which has such an API implemented in com.microsoft.z3. On the other hand, other solvers, such as Boolector, doesn't have such an API. In those cases, it is necessary to use the foreign function interface mechanism in Java, namely Java Native Interface (JNI). With it, it is possible to access the code of the API of the solver, even if it is in another language, different from Java.

Since dReal currently doesn't have a Java API, it will be necessary to use JNI. For that, we must produce prototypes for functions in Java corresponding to the existing C code, defining the signatures of functions and mapping the types correctly. While it is possible to do it by hand, the most suitable approach is to use Simplified Wrapper and Interface Generator (SWIG), which is a tool that can read some C/C++ code and automatically generate those prototypes for a number of languages, including Java.

In this context, there are two main challenges to overcome:

- The code present in the native solver API doesn't always contain all the functionality required by the JavaSMT library. For instance, many solvers' APIs doesn't have a function that take a string directly representing a problem encoded in SMT-LIB2 and try to solve it using the solver, which is an essential part of the functionality of JavaSMT. In those cases, there is just a function that reads the problem to be solved from a file and invoke the solver on it. Therefore, it is necessary to implement another C function, that would receive the string containing the problem, dump it into a temporary file and call the existing function from the API. This exactly tweak is done, for instance, in the function *Java_org_sosy_1lab_java_1smt_solvers_boolector_-BtorJNI_boolector_1help_1parse*, present in the file *java-smt/lib/native/source-/libboolector/interface_wrap.c*, which adjusts the interface for boolector. Besides that, it is often the case that the types present in the API do not match the required types by JavaSMT. In those cases, it is also necessary to write wrapping functions that cast the types. One example of such function present in JavaSMT is the following, which converts the long type into an instance of Btor, which is a class defined in the project. The name of the function was changed for brevity.

```
SWIGEXPORT jint JNICALL boolector_1sat
  (JNIEnv *jenv, jclass jcls, jlong jarg1) {
    Btor *arg1 = (Btor *) 0 ;

    (void)jenv;
    (void)jcls;
    arg1 = *(Btor **)&jarg1;
    return boolector_sat(arg1);
}
```

- After adding the necessary required functionality, we must compile it, together with the original API for dReal to build a shared object that will be linked to the JavaSMT library. This will be done locally on my machine, that runs Linux, which have tools to facilitate this process.

4 Timeline

Currently, I'm finishing my master's degree. I have finished almost all the code of it, the only tasks left are to benchmark it and write my master thesis, and I have six months left for it. Considering this, I can work between 25 and 30 hours a week for this project.

The following is a proposal of timeline for execution of the tasks, based on an estimation of the time taken to conclude each one of them:

- Week 1: Understand the structure of the code of JavaSMT, as well as the project patterns that are used in detail
- Weeks 2 and 3: Study the documentation available for dReal, outlining the theories supported by the solver and matching them with the corresponding interfaces in JavaSMT. Understand dReal's C++ API.
- Weeks 4 to 6: Use SWIG to generate a JNI wrapper for the C++ API of dReal making the necessary tweaks. Document the code and the design decisions made.
- Weeks 7 and 8: Implement the interfaces decided in weeks 2 and 3, using the native code generated by SWIG. Document the code and the design decisions made.
- Weeks 9 and 10: Write a set of tests following the pattern of tests that already exist in JavaSMT. Document the tests.
- Weeks 11 and 12: Reserved to investigate possible bugs and fix them. Document the finishing state of the integration, describing possible next steps.

5 About Me

I'm a computer scientist currently pursuing a master's degree. I'm passionate about computing and mathematics, and, specially, the intersection between those two areas. Specifically, my main interests are functional programming, formal methods and type theory, but I'm constantly looking for new and exciting challenges.

I discovered the field in 2016, when I started my bachelor's at Universidade Federal de Minas Gerais. During my bachelor's, I produced several projects in Java, for class assignments, which made me comfortable with the language. In 2018 and 2019 I did a project with a professor where we studied in depth type theory and the proof assistant Agda. In 2020 I had my first experience in the industry, as an optimization intern at the company Accenture, where I've written some metaheuristics and tested the efficiency of their software. In 2021 I worked in the company Imagine A.I. (now Plank). We developed a compiler for a DSL in Haskell and wrote smart contracts using Plutus. More recently, in the end of 2021 I joined the MSc. program at Universidade Federal de Minas Gerais, and I'm working to integrate the proof assistant Lean with the SMT solvers, so that the user could discharge the responsibility of some proofs to the solver. More information about this project can be found at `https://github.com/ufmg-smite/lean-smt`.

I also participated in the International Collegiate Programming Contest, where I won a silver medal at the regional level (Maratona Mineira) in 2019 and a silver medal at the national level (Maratona Brasileira) in 2022. Finally, my contributions to open source are mainly represented by personal projects that I develop in my free time. The most relevant one of them is an interpreter for Lambda Calculus, written in Haskell and formalized in Agda: `https://github.com/tomaz1502/Lam`. Besides that, I have an open PR in the mathlib repository (the de facto mathematical library of the Lean proof assistant) that corresponds to the work that I did during the final project of my BSc., that is, the formalization of the runtime complexity of sorting algorithms, which can be found here: `https://github.com/leanprover-community/mathlib/pull/14494`.

Overall, I believe that I am a good fit for the project for my previous knowledge of SMT solvers, as well as experience integrating SMT solvers into other systems, which is confirmed by the development of the previously mentioned project, LeanSMT. Besides that, being previously exposed to Java makes me confident that there will be no big surprises coming from the language that would create obstacles in the development of the integration proposed in this document.