Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 2575

A system for SMT based constraint programming in Java

Maurice Funk[1]

Abstract: This paper presents a system for constraint programming in Java using translation of JVM bytecode into SMT. This allows the constraints to be written in normal Java, to being interoperable with the rest of the program and lowers the entry barrier to using specialised solvers in applications. Due to the nature of the performed translation, variable assignments for other runtime properties than constraint satisfaction can be found. These include variable assignments that lead to runtime exceptions during normal code execution. We demonstrate that the implemented approach is able to find variable assignments for non-trivial constraints such as a Sudoku puzzle.

Keywords: Constraint Programming; SMT; Java; Program Analysis; Symbolic Execution

1 Introduction

Many problems in computer science can be expressed as a set of constraints over a number of variables. There are many languages and libraries designed to find satisfying variable assignments for constraints. These constraints are often formulated in domain specific languages, some of them are embedded in a commonly used language through a library. In both cases the semantics of the constraint language does not match the semantics of general programming languages. This fact represents an entry barrier to using specialised solvers for constraint satisfaction problems when a computer programmer encounters one while using a general purpose language.

If, however, the semantics (and syntax) of a constraint satisfaction language were to exactly match a general purpose language, this entry barrier would be lower and the solving of a constraint satisfaction problem in a general purpose language would require little additional work. This paper presents a library for the Java programming language that accomplishes exactly that. It accepts Java objects as problem representations, wherein (almost) arbitrary methods with boolean return value can be constraints. This is accomplished by translation of the JVM bytecode that a standard Java compiler produces into an Satisfiability Modulo Theories (SMT) formula [Ba09]. The system then uses an SMT solver such as Z3 [DB08] to find constraint fulfilling values for all variables. Figure 1 shows an example of a constraint as a Java method. It constrains |a|, |b| and |c| to be a Pythagorean triple.

[1] Universität Bremen, mfunk@cs.uni-bremen.de

doi:10.18420/in2017_260

```
@Constraint boolean constraint(int a, int b, int c) {
    return a * a + b * b == c * c
}
```

Fig. 1: Java code for a simple constraint over three variables

2 Related Work

Soeken et al. [SND14] implemented a similar constraint solving library using a different approach. Their system restricts the permitted Java methods to simple checks of objects. These checks are then symbolically applied to annotated collections of Java objects. That forces the problem descriptions to be in a specific shape and requires some conversion work. This limitation inspired the creation of the system described in this paper.

z3py is the Python API of the Z3 SMT solver [DB08]. It is mentioned here since it also aims to have a low entry barrier and allows using an SMT solver from a general purpose programming language. However, while it uses Python syntax, problems have to be formulated with SMT semantics and thus need to be translated from their Python representation by the user.

3 Approach

Java was chosen as an embedding language, since it is a relatively widespread, commonly taught high level language, which is compiled to the analysable JVM bytecode. Analysing the intermediate representation (IR) that is the JVM bytecode is significantly easier than analysing the program in its original syntax.

The approach implemented in the presented system translates the constraint satisfaction problems encoded in the constraint methods as JVM bytecode and the object state into a single SMT formula which, in turn, is solved by an SMT solver producing a variable assignment that satisfies the constraint methods. Figure 2 shows an overview of the process used to translate the problem into an SMT formula. In a first step the methods that represent the constraint (that are annotated with `@Constraint`) and all reachable methods are symbolically executed into an IR. The annotation mechanism to identify relevant methods is used since it has advantages over other mechanism, e.g. providing a Java interface for the user to implement. Annotations allow multiple constraint methods and arbitrary method signatures, instead of the fixed number and fixed signatures of the interface approach. Without any other annotations, only the parameters of the constraint methods are treated as variables of the constraint problem. If the system should treat any fields of the problem object (or of other reachable objects) as variables as well, they have to be annotated with `@Variable`.

In the IR of the constraints each method is a (potentially very complex) partial function from an old state to a new state that corresponds to the semantics of the translated Java method. The function is only partial since the execution of the corresponding constraint

method might not terminate for all inputs, which translates to an infinite recursion chain in the IR. Combined with the initial state of the problem object and other reachable objects, the IR is in multiple steps translated into an SMT formula.

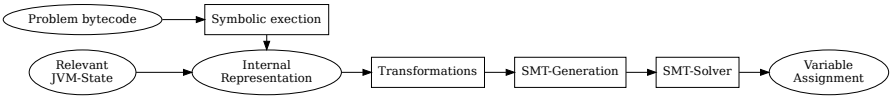

Fig. 2: Overview of the approach implemented in the software

The translation is required in order to accomplish several things. Complex control flow (or recursion) in the constraint methods results in recursive state function definitions which cannot be correctly expressed in SMT, thus the system needs to "unroll" the definitions with an upper depth boundary, similar to bounded model checking techniques (first introduced by Biere et al. [Bi99]). Concepts like objects, arrays and heap accesses need to be lowered to SMT as well, since these constructs are not available in SMT or their semantics do not match their Java counterparts. The transformed IR is in a final step used to generate an equivalent SMT formula, which is satisfied by a variable assignment if and only if the all constraint methods return the value `true` for the assignment. The generated SMT formula corresponding to the small example from Figure 1 is shown in Figure 3.

```
(let ((a!1 (ite (= (+ (* x1 x1) (* x2 x2)) (* x3 x3)) 1 0)))
  (= x0 a!1))
(assert (= x0 1))
```

Fig. 3: SMT formula generated for the simple example constraint in Figure 1

4 Implementation

We implemented the approach described in the last section using the Java interface of the SMT solver Z3. Variable assignments that satisfy the generated SMT formula also satisfy the constraints of the Java formulation of the problem and are returned to the user.

Since other properties of the program state are modeled in the produced SMT formula, the current system is also capable of finding variable assignments that lead to other outcomes than constraint satisfaction. For example the system can be configured to generate variable assignments that lead to nullpointer exceptions. Thus this approach may also be applicable to program testing or testcase generation. These features are, however, currently not well integrated into the system and just point to a possible future direction.

4.1 Translation of a method

As mentioned above, each method in the problem description will be symbolically executed separately[2]. The result for each method is a symbolic partial function from an initial

[2] Note that symbolic execution may be the wrong term for the implemented approach. "symbolic compilation" seems more fitting since the entire program is translated into a single SMT formula

state to a new state. It represents all state modifications the method may perform. This representation is constructed from smaller pieces by splitting the JVM bytecode of the method into basic blocks[3], which in turn are translated into symbolic functions from state to state. The (conditional) jumps from basic block to basic block in a method are constructed using function composition of the functions that represent basic blocks.

The symbolic function f of a basic block that ends in an unconditional jump to a block with symbolic function g can in that manner be constructed like $f(s) = g(f'(s))$, where f' symbolically represents the unconditional state modification of the instructions in the basic block previous to the final jump. It is easy to see that f returns the state after the execution of the basic block corresponding to g. If g is constructed recursively in a similar way, f returns the new state after execution of the method finished. If the block ends in a conditional jump, the functions are composed using a conditional expression as follows:

$$f(s) = \begin{cases} g_1(f'(s)) & f_{condition}(f'(s)) \\ g_2(f'(s)) & \neg f_{condition}(f'(s)) \end{cases}$$

with g_1 and g_2 being the symbolic functions of the two possible jump destinations, $f_{condition}$ being the symbolic condition that the jump destination depends on and f' begin the unconditional state modifications of the instructions of the block up until the conditional jump. Similar to the unconditional case f returns the new state after execution of all successor blocks finishes. It can be noted that these function definitions are recursive if and only if the control flow graph of the original program contains circles, that is if and only if the original program contains loops.

```
if (x > 0) {
    x += 1;
}
```

```
BB0: ifle BB1 else BB2
BB1: iinc x
     jmp BB2
BB2: ...
```

Fig. 4: Simple conditional branch example

Consider the small example in Figure 4 (left) with three basic blocks BB0, BB1, and BB2 (right). The translation results in the following symbolic functions

$$\begin{aligned} \mathrm{BB2}(s) &= \ldots \\ \mathrm{BB1}(s) &= \mathrm{BB2}(s[x \mapsto x + 1]) \\ \mathrm{BB0}(s) &= \begin{cases} \mathrm{BB1}(id(s)) & id(s)(x) > 0 \\ \mathrm{BB2}(id(s)) & id(s)(x) \leq 0 \end{cases} \end{aligned}$$

[3] Basic blocks are sequences of instructions that are always executed in order, i.e. contain no jumps and are not jumped into.

4.2 Limitations

The current implementation supports almost arbitrary Java code in the constraints, including loops, method calls, recursion and modification of the heap as well as static variables. Support for strings and characters is currently quite limited and will be expanded in the future depending on the capabilities of the used SMT solver. Java's exception mechanisms are currently not modeled by the symbolic execution and cannot be used in problem descriptions. Advanced challenges of symbolic execution like native methods, concurrency and IO-operations [Ba16] are currently not handled in the presented system and would require extensions to the current approach and implementation.

5 Evaluation

Since the system attempts to be a constraint solver, it is briefly compared against other easily usable constraint solvers and the system implemented by Soeken et al. [SND14]. We implemented constraints for a 9×9 Sudoku puzzle in the following systems:

- *Choco*, a conventional constraint programming library for Java [PFL16]
- *DIAB*, the system implemented by Soeken et al.
- *z3py*, the python interface of the Z3 SMT solver

Although *DIAB* and the presented system share the idea of using Java code as their input, they do not use the same Sudoku implementation in this evaluation, since *DIAB* does not accept general constraints, but constraints on sets of objects. The version for the presented system directly constrains values of array elements. The implementation for *Choco* and *z3py* both directly produce inequality constraints according to the Sudoku rules in their respective domain language.

All tested systems are able to find a satisfying assignment for 9×9 Sudoku constraints given a constraint formulation in their input language. The average of the required runtimes of 20 runs are listed in Table 1.

System	Average time
Choco	4 ms
presented system	1218 ms
DIAB	39127 ms
z3py	440 ms

Tab. 1: Runtime for finding a satisfying variable assignment for a single 9×9 Sudoku puzzle

It can be noted that the presented system does not reach the speed of specialised constraint programming libraries or directly working with SMT solvers. That can be explained by the additional work of program analysis and translation this approach imposes before the actual constraint solving begins. Compared to *DIAB*, the presented system is able to find a variable

assignment faster. If that is due to differences in implementation speed or due to a different approach is unknown. Additionally this system does not require the Java description to be in a specific shape, while *DIAB* does.

6 Conclusion and Future work

This paper presented a system for constraint programming in Java. The current technical limitations are listed in subsection 4.2. Although the current implementation cannot be applied to complex problems interacting with the environment due to the mentioned limitations, the presented system is able to find satisfying variable assignments for a lot of constraint problems given just a problem description in Java. Due to the program analysis required to make this possible, the system is slower than directly working with the underlying SMT solver and a lot slower than using a specialized library for constraint programming, but a lot faster than an older approach.

Future work includes removing some of the technical limitations and decreasing the performance gap between the presented system and direct usage of a SMT solver interface. Furthermore the current system needs to be evaluated in more aspects than performance, especially if it aims to be a low barrier system for constraint programming. Additionally the applicability to testcase generation and how this approach compares to conventional symbolic execution should be investigated.

References

[Ba09] Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; Tinelli, C.: Satisfiability Modulo Theories. Handbook of satisfiability 185/, pp. 825–885, 2009.

[Ba16] Baldoni, R.; Coppa, E.; D'Elia, D. C.; Demetrescu, C.; Finocchi, I.: A Survey of Symbolic Execution Techniques. CoRR abs/1610.00502/, 2016.

[Bi99] Biere, A.; Cimatti, A.; Clarke, E.; Zhu, Y.: Symbolic model checking without BDDs. Tools and Algorithms for the Construction and Analysis of Systems/, pp. 193–207, 1999.

[DB08] De Moura, L.; Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 337–340, 2008.

[PFL16] Prud'homme, C.; Fages, J.-G.; Lorca, X.: Choco Documentation, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016, URL: `http://www.choco-solver.org`, visited on: 06/13/2017.

[SND14] Soeken, M.; Nitze, M.; Drechsler, R.: Formale Methoden für Alle. In: ITG/GMM/-GI-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen - Proceedings. Cuvillier Verlag Göttingen, pp. 213–217, 2014.