Unleashing the Verification Genie in the Cloud

Nikolaj Bjørner
Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
nbjorner@microsoft.com

Abstract

Z3 is a state-of-the-art SMT (Satisfiability Modulo Theories) solver available from Microsoft Research. It is used as a logic engine in several program analysis, test-generation and verification systems. Z3 exposes a number of APIs for these applications, but is in itself a low-level tool. We discuss the design trade-offs we have faced and different ways we lower the barrier of entry to use Z3. This includes using programming language abstractions, such as quotations and LINQ (Language Integrated Query features) and exposing Z3 as a web service.

1 Introduction

Most program analysis, test and verification tools rely on a reduction to logic queries. At their core, most logic queries can be formulated as first-order logic formulas, and answers to these logic queries can be categorized as a yes/no answer (is this formula satisfiable or dually valid), a model (a counter-example to validity), a proof, or less ambitiously, an unsatisfiable core, which is a subset of the assertions that together suffice to produce a proof. What makes the program analysis, test and verification tools usable is a combination of ease of use and appropriateness for their domain together with scale, performance and precision of their analysis engine. Much of the burden for the latter is on the logic engine and how it is used.

The Z3 [3] solver available from Microsoft Research is a state-of-the-art SMT (Satisfiability Modulo Theories) solver. It serves the purpose as the logic engine under several higher-level tools. This position paper discusses some of the recent efforts in making Z3 even more usable, and some efforts around Z3 to lower the barrier of entry to using it.

Section 2 summarizes some of the (more recent) core enhancements that seek to improve usability. Section 3 discusses select interaction models for Z3. The emphasis is on using Z3 using modern programming language features and from the cloud.

2 Core Technologies

2.1 What makes an SMT solver good for usable verification?

First and foremost, our experience has been a that automatic and scalable support for a rich set of base theories is the most important part. Different tools use different theory features. For example, Boogie/Spec# [1] uses the theory of integers and formulas using quantifiers. On the other hand, Pex [4] uses arrays, bit-vectors and to a lesser extent integers and quantifiers even though they are good to have. FORMULA [6] uses quantifiers, bit-vectors, algebraic data-types.

2.2 Special features

Base theories, however are not enough. The SLAyer [5] system builds first-order formulas from symbolic execution based on separation logic. This introduces quantified formulas over integers and arrays and good quantifier elimination support is required to handle such formulas [2]. Furthermore, SLAyer also benefits with access to all the equalities that are implied from a given formula. Consequently, Z3 exposes a function to retrieve label a set of terms by their equivalence class representatives.

Example: The formula

$$\varphi : x \leq y+1 \wedge y \leq z-1 \wedge z \leq x$$

is satisfiable, but it constrains the interpretations for x, y, and z such that $x = y+1 = z$. From SMT-LIB2, you can learn the implied equalities using a query of the form:

```
  (declare-funs ((x Int) (y Int) (z Int)))
  (assert (and (<= x (+ y 1)) (<= y (- z 1)) (<= z x)))
  (get-implied-equalities x z (+ y 1) y (- z 1))
; (18 18 18 19 19)
```

The `get-implied-equalities` function takes a list of terms as arguments. It produces a list of integers. Each integer identifies a partition, so that two terms in the same equivalence class receive the same partition identifier. In the example the terms `x`, `z` and `(+ y 1)` are equal, so are `y` and `(- z 1)`. □

2.3 Extensibility

There comes a point where Z3 cannot be a host of arbitrary theories or features. Consider for example, the theory of *object graphs* with read-only fields. It comprises of the signature

$$\langle O, null : O, value : O \rightarrow Int, left : O \rightarrow O \ \mathsf{ro}, right : O \rightarrow O \ \mathsf{rw} \rangle$$

where O is the sort of objects, *null* is a special constant of object sort. The *value* field associates with an object an integer value and *left* and *right* retrieve the left and right children of an object. Furthermore the *left* field is *read-only*. The intent here is that this field cannot be updated, and therefore objects can only have well-founded chain of left-pointers (eventually ending with *null*). The *right* field is writable, a program can update this field and create a cyclic reference. Z3 does not support the theory of object graphs. The theory of algebraic data-types supports extensional equalities, equality over object graphs is not extensional, and algebraic data-types are well-founded. On the other hand, co-inductive types don't capture the acyclic read-only fields. We can soft-code object graphs as uninterpreted sorts and uninterpreted functions. A trick to enforce well-founded left chains is to associate an ordinal with each node and require that the ordinal decreases with each left deference:

$$\forall x : O \ . \ null \neq x \rightarrow ord(x) > ord(left(x))$$

Soft-coding does not allow the user to control preferences for some models over others. A specialized theory solver allows for full control and Z3 exposes methods for implementing custom theory solves. The custom theory solver for the theory of object graphs works essentially the same as the solver for algebraic data-types, except that it allows to disregard extensionality and it only requires acyclicity along read-only fields:

1. It maintains a set of objects that are asserted as non-null. The set is updated when the SMT core asserts a dis-equality between a term of sort O and *null*.
2. It maintains three queues for building interpretations of object terms. Producing an interpretation is the main feature of the decision procedure.
   (a) The first queue contains terms that the procedure will attempt assigning to *null*. Terms that are already in the set of non-nulls are inserted directly into the second queue. For term that are not already non-null and not already equal to *null*, the procedure creates an equality literal $t \simeq null$ and asks the solver core to assign the equality to *true*. If the solver core is unable to assign the atom to *true*, it backtracks and assigns the atom to *false*.

(b) The second queue contains terms that cannot be *null*, and needs to be assigned to one of the legal object sub-types.

(c) Once the proper object type has been fixed with an object term, terms from the last queue have their object fields assigned. The fields may recursively be objects.

3. To enforce acyclicity of read-only fields it checks partial assignments for occurs check violations.

The resulting theory solver can be programmed using Z3's managed API in less than 250 lines of F#.

3 Exposing Z3

3.1 Using Z3 from LINQ

LINQ, Language Integrated Queries, has been a part of Microsoft's C# language since C# v3.5 (November 2007). Its main use is making it easier to write SQL queries. However, it is easily re-targeted as the underlying mechanism is to build expression trees and rely on a back-end to process the expression trees. This was exploited more than a year ago by Bart De Smet during a plane trip and put up on his personal blog under a *Crazy Sunday* tag. Indeed, writing the LINQ provider is fairly straight-forward. The payoff with ease of use in the context of the host language C#, however, is significant. The comment left by `aL`, inspired the title of this paper: writing expression manipulating programs is sufficiently tedious that it poses a barrier of entry.

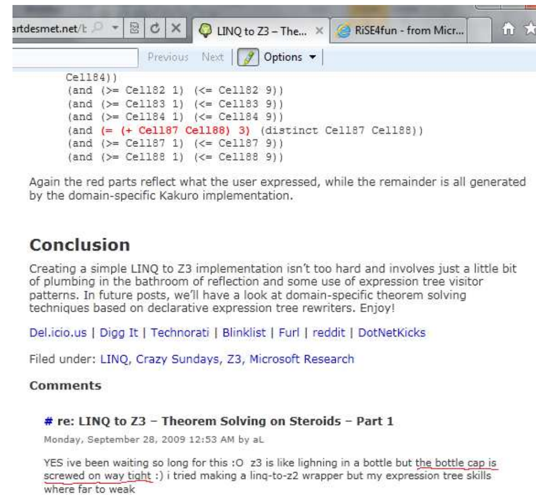

Figure 1: An excerpt from Bart De Smet's blog on using Z3 from LINQ

3.2 Using Z3 with F# Quotations

The Z3 distribution comes with power utilities for the F# programming language. A prolific feature of F# is the availability of *quotations*. Quotations have their origins in LISP: you can quote a piece of code and treat it as data. You can also quote code in F#, and access the abstract syntax tree for it. This feature is used for encoding formulas as F# expressions. It makes for quite legible syntax. The scheduling constraints using the quotation support from the Z3 distribution can be formulated as follows:

```
open Microsoft.Z3
open Microsoft.Z3.Quotations

do Solver.prove <@ Logic.declare
      (fun t11 t12 t21 t22 t31 t32 ->
         not
            ((t11 >= 0I) && (t12 >= t11 + 2I) && (t12 + 1I <= 8I) &&
             (t21 >= 0I) && (t22 >= t21 + 3I) && (t32 + 1I <= 8I) &&
             (t31 >= 0I) && (t32 >= t31 + 2I) && (t32 + 3I <= 8I) &&
             (t11 >= t21 + 3I || t21 >= t11 + 2I) &&
             (t11 >= t31 + 2I || t31 >= t11 + 2I) &&
             (t21 >= t31 + 2I || t31 >= t21 + 3I) &&
```

```
            (t12 >= t22 + 1I || t22 >= t12 + 1I) &&
            (t12 >= t32 + 3I || t32 >= t12 + 1I) &&
            (t22 >= t32 + 3I || t32 >= t22 + 1I)
           )
      )
   @>
```

Let us explain some of the features used in the example:

- `Solver.prove` consumes an expression of type `Expr<bool>`, a quoted expression of type `bool`. It checks for validity of the formula that results from compiling the expression. Since, we are interested in *satisfiability* of the scheduling constraints we check for validity of their negation.
- `Logic.declare` is a function that takes an arbitrary curried lambda expression and creates fresh constants for the variables that are bound by the lambda expression. In this case, it creates constants for `t11 t12 t21 t22 t31 t32`. The F# type inference will infer that these variables have type `BigInteger`. Z3 represents these as plain integers.
- The notation *big integer* literals in F# is to suffix numbers with an `I`, for example `1I` and `8I`. For "normal" integers, such as `1` and `8`, Z3's quotation compiler uses fixed-size bit-vectors.

3.3 Discover RiSE4fun.com

An important component of usability is also discover-ability. What are the capabilities of a given tool? It is much easier to make such discoveries interactively by using the tools directly. Tool installation poses an additional obstacle that can be quite inhibiting. Tools exposed over the web removes this barrier of entry.

The `http://rise4fun.com` web site sports a number of tools developed at Microsoft's RiSE (Research in Software Engineering) group, including Z3 and several tools using Z3 under the hood; Bek, Boogie, Dafny, Pex, Rex, Spec#, and Vcc. The interaction with Z3 is so far quite simple: a user poses a formula in the SMT-LIB2 format and gets back and answer, which can be a simple `sat/unsat` answer or it can be a model, or unsatisfiable core, or it can be a proof object.

The interaction with Pex is significantly richer. Here a user can enter a C#, F# or Visual Basic program, even aided by intellisense (for C#), and obtain the unit test results of running dynamic symbolic execution. A different interaction model, known as *code duels*, allows the user to learn a hidden specification by having Pex provide counter-example input/output pairs. This seems to be by far the most popular feature. The Rex tool likewise exposes a duel feature.

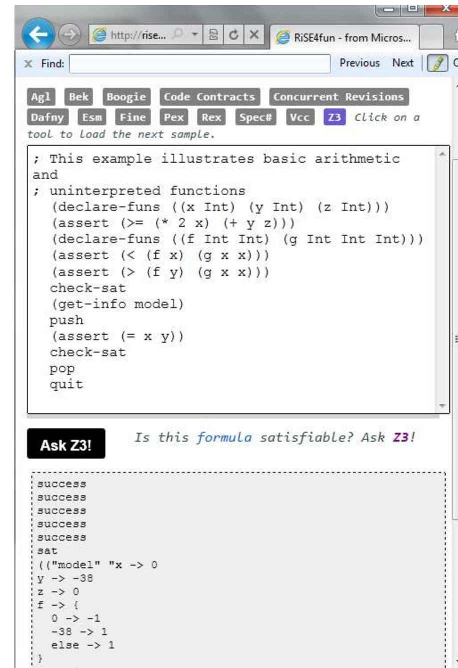

Figure 2: http://rise4fun.com

4 Conclusion

There are two parts to usability of SMT solvers: (1) the power and adaptability of the core set of features exposed by the solver; and (2) the ease of use. This position paper discussed what is done in the context of Z3 to address these two challenges.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*, LNCS 3362, pages 49–69. Springer, 2005.

[2] Nikolaj Bjørner. Linear Quantifier-Elimination as an Abstract Decision Procedure. In *IJCAR*, 2010.

[3] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 08*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[4] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.

[5] http://research.microsoft.com/en us/um/cambridge/projects/slayer/.

[6] Ethan K. Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjørner, and Wolfram Schulte. Specifying and composing non-functional requirements in model-based development. In Alexandre Bergel and Johan Fabry, editors, *Software Composition*, volume 5634 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2009.