Satisfiability Modulo Bit-precise Theories for Program Exploration

Nikolaj Bjørner, Leonardo de Moura, Nikolai Tillmann

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{nbjorner, leonardo, nikolait}@microsoft.com

Abstract. The Satisfiability Modulo Theories solver Z3 [10] is used in several program analysis and verification tools at Microsoft Research. Some of these tools require bit-precise reasoning for accurately modeling machine arithmetic instructions. But this alone is rarely sufficient, and an integration with other theories is required. The Pex tool [20] performs program exploration of .NET programs by generating and solving path conditions corresponding to paths that get explored during concrete execution. The path conditions reflect directly the executed instructions, including ones involving machine arithmetic supported by the CLR. The path conditions include also operations on heaps and structures. Pex relies on Z3's ability to produce models for satisfiable path conditions, the models must reflect the combination of the involved theories: bit-vectors, arrays, and tuples. This paper describes the features of Z3 that are used by Pex.

1 Introduction

The ability to solve logical assertions is essential in several tools that perform program analysis, program verification and program testing. Satisfiability Modulo Theories (SMT) solvers appear to match these tools particularly well, since they integrate a number of the domains that occur naturally in programs. For example, machine arithmetic is handled by the theory of bit-vectors, heaps are handled by a theory of arrays, and a theory of uninterpreted functions and integers can be used to abstract program state.

This paper illustrates the use of the solver Z3 in the context of one of its clients, Pex - a program exploration tool. Both tools are freely available for download from Microsoft Research.

2 Z3

Z3 [17] is an SMT solver from Microsoft Research. Z3 is targeted at solving problems that arise in software verification and software analysis. Consequently, it integrates support for a variety of theories. Z3 combines several algorithms, in particular for quantifier instantiation [7], superposition [11], theory combination [8], and for discriminating relevant atoms from don't cares [9].

Algorithm 3.1 Dynamic symbolic execution

Set $J := \mathit{false}$	*J summarizes the set of already analyzed program inputs*
loop	
Choose program input i such that $\neg J(i)$	*stop if no such i can be found*
Output i	
Execute $P(i)$; record path condition C	*in particular, $C(i)$ holds*
Set $J := J \vee C$	
end loop	

Z3 supports binary interfaces for C, OCaml, and .NET. It also allows supplying text inputs in the Simplify, SMT-LIB, and a native format. It has been integrated with several projects at Microsoft Research, including, Spec#/Boogie [2, 12], Pex [20], Yogi [15], Vigilante [6, 5], SLAM/SDV [1], HAVOC [18], and F_7 [3].

3 Pex

Pex [21, 19] is a dynamic symbolic execution platform Pex for .NET. It is developed at Microsoft Research. Pex contains a complete symbolic interpreter for safe programs that run in the .NET virtual machine, and it also supports reasoning about pointers and unsafe memory accesses. Pex uses Z3 as a constraint solver, using Z3's ability to compute models for satisfiable constraint systems. Pex has been used within Microsoft to test core .NET components developed at Microsoft. Pex is integrated with Microsoft Visual Studio and available from [19].

Dynamic symbolic execution [14, 4] is a variation of conventional static symbolic execution [16]. Dynamic symbolic execution consists in executing the program, starting with arbitrary inputs, while performing a symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. Then a constraint solver is used to compute variations of the previous inputs in order to steer future program executions along different execution paths. In this way, all execution paths will be exercised eventually. Algorithm 3.1 shows the general dynamic symbolic execution algorithm decides in which order the different execution paths of the program are enumerated.

The advantage of dynamic symbolic execution over static symbolic execution is that the abstraction of execution paths can leverage observations from concrete executions, and not all operations must be expressed and reasoned about symbolically. Using concrete observations for some values instead of fully symbolic representations leads to an under-approximation of the set of feasible execution paths, which is appropriate for testing. Such cases can be detected, e.g. when a function is called that is defined outside of the scope of the analysis. Our tool reports them to the user.

3.1 Symbolic path conditions

In a concrete execution the program's state is given by a mapping from program variables to concrete values, such as 32 or 64-bit integers and heap-allocated pointers and objects managed by the .NET garbage collector.

the state is a program counter and a mapping from program variables to terms built over symbolic input values, together with a predicate over symbolic input values, the so-called path condition. The path condition summarizes the constraints on input variables that must be satisfied in order for the program execution to reach the program counter.

For example, for the function

```
void compare_and_multiply(int x) {
  if (x>0) {
     int y = x*x;
     if (y==0) {
        target:
     }
  }
}
```

the path condition for reaching *target:* consists of two conjuncts $(x > 0)$ and $(x * x == 0)$. Note that the symbolic state is always expressed in terms of the symbolic input values, that is why the value of the local variable `y` was expressed with the symbolic input x.

3.2 Solving symbolic path conditions

The path condition $x > 0 \wedge x * x = 0$ is clearly not solvable when x ranges over integers, but it is solvable for machine represented numerals. When x ranges over 32-bit integers, then one possible solution is $x = 2^{16}$. Pex requires the semantics of machine representable numerals for accurately generating test cases.

3.3 ResourceReader - an example from the .NET BCL

An illustrative example of the use for integrating bit-level reasoning and other theories is the exploration of the .NET Base Class Library `ResourceReader`. The `ResourceReader` provides a default implementation of a utility that reads *resource* files. Resource files are used by Windows applications to display text in culture-specific ways. For example, when setting the default language of Windows to German, applications will load strings from the German resource files. Abstractly, resource files consist of a set of key-value pairs, and the role of a resource reader is to parse a text file and create a dictionary of the encoded key-value pairs. Here, the values are serialized instances of .NET classes, and the `ResourceReader` implements type-specific de-serialization.

We can use Pex to test the `ResourceReader` utility without having any resource files available. The listing in Fig. 1 shows a parameterized unit test that

exercises the `ResourceReader`. It consists of a function `ReadEntries`, which takes as input a non-null buffer of bytes (data), creates a stream from the bytes, then creates an instance of the ResourceReader class, and finally consumes the resources in the stream. Note that the construct `fixed (byte* p = data)` connects objects that are managed by the .NET garbage collector with pointers whose bit-vector nature the program can, and in fact does, reason about. The attribute [PexMethod] is used by Pex to identify the method as a parameterized unit test. Thus, values for data will have to be synthesized by dynamic symbolic execution.

```
[PexClass, TestClass]
public partial class ResourceReaderTest {
  [PexMethod]
  public unsafe
  void ReadEntries([PexAssumeNotNull]byte[] data) {
    fixed (byte* p = data)
    using (UnmanagedMemoryStream stream =
      new UnmanagedMemoryStream(p, data.Length))  {
      ResourceReader reader = new ResourceReader(stream);
      foreach (var entry in reader) { /* just reading */ }
    }
  }
}
```

Fig. 1. A Resource Reader test class

In general, the `ResourceReader` cannot assume anything about the contents of the stream supplied to the constructor. As is common with binary file formats, a *cookie* is used to provide a partial filter to distinguish file streams that may possibly correspond to well-formed resource files and those that are malformed. A cookie match is of course no assurance that the remaining of the stream conforms to the definition of the `.resources` format, but it provides a first sanity check. Fig. 2 shows the implementation within the Microsoft Base Class Library of the cookie check. Fig. 3 further illustrates the how a 32-bit integer is reconstructed by bit manipulation. In contrast to the white-box dynamic symbolic execution approach, a random block-box test input generator would most likely never pass this hurdle.

To exercise the remaining of the `ResourceReader` class Pex needs to create test inputs that match the cookie. By performing dynamic symbolic execution, the branch where the cookie match is checked corresponds to a constraint. Fig. 4 shows the term graph for the path condition for a successful cookie match. The corresponding formula is:

$$\mathit{0xcecaefbe} = (a[0] \mid a[1]{<<}8 \mid a[2]{<<}16 \mid a[3]{<<}24) \text{ where } a = \$\mathit{Items}[\mathit{data}].$$

We notice that Pex maintains the input array `data` within a global array `$Items` of array inputs, and that the constraint refers to the elements stored at indices

```
private void ReadResources() {
  BinaryFormatter bf = new BinaryFormatter(null, ...);
  bf.AssemblyFormat = FormatterAssemblyStyle.Simple;
  _objFormatter = bf;

  try {
    int magicNum = _store.ReadInt32();
    if (magicNum != ResourceManager.MagicNumber)
      throw new ArgumentException();
...
```

Fig. 2. Checking the `.resources` cookie

```
public virtual int ReadInt32() {
  FillBuffer(4);
  return (int)(m_buffer[0] | m_buffer[1] << 8 |
               m_buffer[2] << 16 | m_buffer[3] << 24);
}
```

Fig. 3. Checking the `.resources` cookie

0, 1, 2, and 3; which should match a *beefca"k"e* cookie. These indices correspond to the run-time values of the indices used by `ReadInt32` for parsing the first four bytes.

Upon solving this path condition, which is easy with a solver that supports bit-vectors and arrays, Pex can use the solution to continue exploring the paths following the cookie check. Fig. 5 shows an example buffer that was generated by exploring branches following the cookie check in the resource reader.

```
byte[] a = new byte[14];
a[0] = 206;
a[1] = 202;
a[2] = 239;
a[3] = 190;
a[7] = 64;
a[11] = 128;
```

Fig. 5. Sample resource reader input

3.4 Heap constraints on garbage-collected objects

Addresses of garbage-collected objects are opaque: while they are pointers that can be represented as bit-vectors, they cannot be observed by the program. Pex takes advantage of the abstraction by representing pointers as abstract, distinct, values. To handle such object values in constraints, the solver needs to provide a theory that provides distinct elements. To this end, we use integers that are

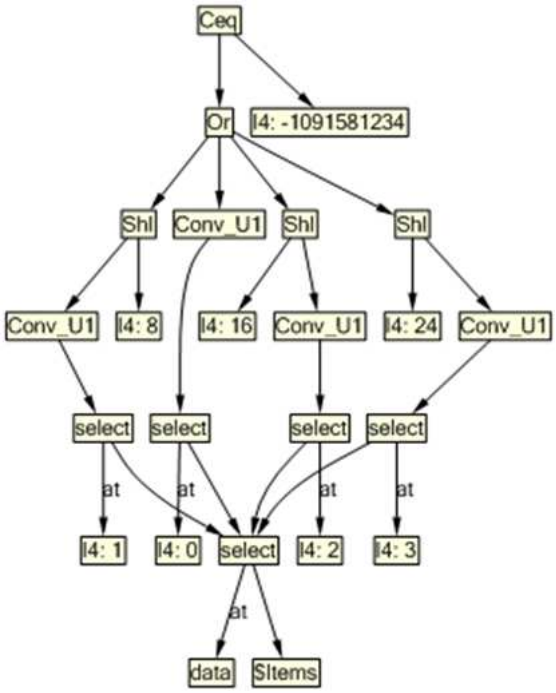

Fig. 4. A path condition for matching the `.resources` cookie

available through the theory of linear integer arithmetic. Similarly, we encode the types of a .NET program as integers.

We take an approach similar to how the heap is encoded in ESC/Java [13]. Each mutable field of a .NET class is modeled by an evolving mathematical mapping from objects to values. For example, for the class `Class`, with the declaration, `class Class { int X; int Y; }`, there will be a mapping for `X` and one for `Y`, which associates objects with their respective values. To this end, we use the theory of arrays with the usual read/write functions in Z3.

In the same spirit, an array can be seen as a class with a single mutable field, which itself is an evolving mapping from integers, representing array indices, to bytes. Intuitively, the class `byte[]` corresponds to a class with a single mutable field that holds the map `class byte[] { map<int,byte> $Items; }`.

As a result, .NET arrays are encoded as a map of (array) objects to a map of integers to values.

We omitted two conjuncts from the path condition that required that the `data` object is not null, and that it is object that has the byte-array type.

$$data \neq 0 \wedge \$Type(data) = 1$$

The `null` object has been assigned the value 0, and the type for byte arrays has been assigned the value 1. $\$Type$ is an uninterpreted function that maps integers (representing objects) to integers (representing types).

3.5 Minimizing values

We omitted another conjunct from the path condition that was required for generating the input buffer. The condition imposes that the length of the buffer be at least 4, to ensure that an array of appropriate size gets allocated to hold the cookie. Thus, the constraint

$$\$Length(data) \geq 4$$

is added, where $\$Length$ is an uninterpreted function that maps integers (representing objects) to 32-bit bit-vectors (representing array lengths). We use universal quantifiers to state axioms which restrict possible interpretations of the $\$Length$ function. In particular, to ensure that array lengths are not negative:

$$\forall x.\$Length(x) \geq 0$$

Clearly, $\$Length(data) = 4$ is a satisfying assignment to $\$Length(data)$, but so is 5, or 6, or $2^{31}-1$. However, the last assignment is not going to be helpful because running the resource reader with this input will cause attempting to allocate a prohibitive amount of memory. Pex interacts with Z3 to guide the generation of test inputs by minimizing some values. We do not attempt to solve the optimization problem in its full generality, but instead use a partial greedy algorithm for minimizing selected values. The algorithm is shown in Algorithm 3.2. It starts with a constraint $C(x)$ with variable x. The constraint is tested for feasibility. The first model produces an upper bound on x. It then seeks to reduce the upper bound until the lower and upper bounds are the same.

Z3 exposes methods Push and Pop to enable asserting and retracting bounds. So if a bound $x < mid$ is infeasible, it is undone by calling Pop. On the other hand, if the bound is feasible, then the context is not popped. This enables subsequent search to re-use conflict clauses that were learned.

Note that the greedy minimization algorithm is complete when only one variable is minimized. When used for several variables, it will find a local minimum according to the order in which variables are minimized. It cannot be used to minimize, say, the sum of the variables.

4 Conclusion

We illustrated the use of the SMT solver Z3 in the context of the program exploration tool Pex. Pex used a combination of several of the theories supported in Z3 to model different aspects related to symbolic execution. Other program analysis and verification tools also use Z3 for solving logical formulas. These tools exercise yet other features of Z3. For example, the program verification tool chains VCC, Spec#, and HAVOC all prolifically use quantified formulas to summarize loop invariants and frame conditions on heaps.

Algorithm 3.2 Greedy minimization algorithm

Assert $C(x)$	*Assert the path condition*
Let m be a model for $C(x)$	
If there is no model, return Unsat	
Set $lo := 0, hi := m(x)$	*Initialize bounds for x*
while $lo < hi$:	
Push	*Push a local context*
Let $mid = (lo + hi)/2$	
Assert $x < mid$	
If context has a model m Then	
Set $hi := m(x)$	
Else	
Set $lo := mid$	
Pop	*Pop the local context*
Pop remaining pushed contexts	
Return the last model m	

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.
2. 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.
3. Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Refinement types for secure implementations. In *CSF*, 2008.
4. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.
5. Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *ASPLOS*, 2008.
6. M. Costa, J. Crowcroft, M. Castro, A. I. T. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In A. Herbert and K. P. Birman, editors, *SOSP*, pages 133–147. ACM, 2005.
7. L. de Moura and N. Bjørner. Efficient E-matching for SMT Solvers. In *CADE'07*. Springer-Verlag, 2007.
8. L. de Moura and N. Bjørner. Model-based Theory Combination. In *SMT'07*, 2007.
9. L. de Moura and N. Bjørner. Relevancy Propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.
10. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS 08*, 2008.
11. L. de Moura and N. Bjørner. Engineering DPLL(T) + Saturation. In *IJCAR'08*, 2008.
12. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research, 2005.
13. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

14. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. *SIGPLAN Notices*, 40(6):213–223, 2005.
15. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In Michal Young and Premkumar T. Devanbu, editors, *SIGSOFT FSE*, pages 117–127. ACM, 2006.
16. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
17. L. de Moura and N. Bjørner. Z3. http://research.microsoft.com/projects/z3, 2008.
18. S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *POPL'2008*, 2008.
19. Pex development team. Pex. http://research.microsoft.com/Pex, 2007.
20. N. Tillmann and W. Schulte. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE software*, 23:38–47, 2006.
21. Nikolai Tillmann and Jonathan de Halleux. Pex – white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.