Applications and Challenges in Satisfiability Modulo Theories

Leonardo de Moura
Microsoft Research
One Microsoft Way
Redmond, WA 98052
leonardo@microsoft.com

Nikolaj Bjørner
Microsoft Research
One Microsoft Way
Redmond, WA 98052
nbjorner@microsoft.com

Abstract

The area of software analysis, testing and verification is now undergoing a revolution thanks to the use of automated and scalable support for logical methods. A well-recognized premise is that at the core of software analysis engines is invariably a component using logical formulas for describing states and transformations between system states. One can thus say that symbolic logic is the calculus of computation. The process of using this information for discovering and checking program properties (including such important properties as safety and security) amounts to automatic theorem proving. In particular, theorem provers that directly support common software constructs offer a compelling basis. Such provers are commonly called satisfiability modulo theories (SMT) solvers. Z3 is the leading SMT solver. It is developed by the authors at Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories such as arithmetic, bit-vectors, lists, records and arrays.

This paper examines three applications of Z3 in the context of invariant generation. The first lets Z3 infer invariants as a constraint satisfaction problem, the second application illustrates the use of Z3 for bit-precise analysis and our third application exemplifies using Z3 for calculations.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers have been the focus of increased recent attention thanks to technological advances and an increasing number of applications. The Z3 solver from Microsoft Research has several applications. We describe several of them, some are shipped with Windows 7; others are used as part of internal security testing and some are in an earlier research stage. Z3 is the premier SMT solver. It is currently mainly targeted at solving problems that arise in software verification and software analysis. Consequently, it integrates support for a variety of theories that arise naturally in the context of program analysis. Z3 is released publicly for non-commercial use and is available from Microsoft Research for download at

`http://research.microsoft.com/projects/z3`.

You can try Z3 on-line at

`http://rise4fun.com/z3`.

This paper describes three uses of Z3. Section 2 takes as starting point the VS3 project that treats problems, such as invariant generation, ranking function synthesis, and program fragment synthesis as a constraint satisfaction problem. We illustrate how a recent feature in Z3 can be used to solve such satisfaction problems. Section 3 recalls a project using Z3 in the context of the scalable program analysis system PREfix. Many invariant generation techniques take as starting point solvers for the domains of mathematical integers and reals. This is at best imprecise in the context of software analysis. We use the PREfix experience as an illustration for the use for bit-precise static analysis techniques. Our last example in Section 4 is about using SMT solvers for generating test-cases from models. It is not directly related to invariant generation; it illustrates using Z3 for symbolic calculations. Section 5 summarizes several important applications of Z3.

2 From Models to Invariants

The VS3 [31] project uses Z3 to automatically discover inductive invariants for proving safety properties of systems. The project also explores techniques for using SMT solvers to synthesize program fragments. Other work involving Z3 aim to determine the precise asymptotic run-time bounds of programs [21]. The associated tools can extract asymptotic bounds for a majority of routines from the .NET base class library (of the form $O(n)$, $O(nlog(n))$, etc.). The perspective of program analysis (invariant generation and ranking function synthesis) as a constraint solving problem has been pursued in several contexts, including [12, 5, 20]. We will here sketch a technique that uses a newer feature available in Z3, called *Model-Based Quantifier Instantiation* [39]. We will adapt an example used previously in [14] to illustrate the feature. Considering the template while-loop to the left below. The invariant synthesis problem is to synthesize an intermediary assertion I that can be used to show that *post* holds in the end of the *while-loop*. Let $pre[s]$ be a formula encoding the set of states reachable before the beginning of the loop, $c[s]$ be the encoding of the entering condition, $T[s,s']$ be the transition relation for the loop body, and $post[s]$ be the encoding of the property we want to prove. The loop invariant exists if the formula φ_I is satisfiable. Any model that provides an interpretation for I can be used to extract the loop invariant.

```
pre
while (c) {
   T
}
post
```

$$\varphi_I : \begin{array}{l} \forall s.\ pre[s] \to I(s) \land \\ \forall s,s'.\ I(s) \land c[s] \land T[s,s'] \to I(s') \land \\ \forall s.\ I(s) \land \neg c[s] \to post[s] \end{array}$$

We use the following very simple sample program. The loop increments x and y by 1 and 2, respectively. The post-condition asserts that when the loop terminates, then $y = 2n$. The pre-condition of the loop comprises of the assertion and initializations to x and y, thus $pre[x,y,n]$ is $n \geq 0 \land x = 0 \land y = 0$. The loop condition $c[x,y,n]$ is $x < n$, and the transition relation of the loop body is $T[x,y,n,x',y',n']$ given by $x' = x+1 \land y' = y+2 \land n' = n$. Consequently, the formula corresponding to φ_I is:

```
assert(n >= 0);
x = 0; y = 0;
while (x < n) {
   x = x + 1;
   y = y + 2;
}
assert(y == 2*n);
```

$$\begin{array}{l} \forall x,y,n.\ n \geq 0 \land x = 0 \land y = 0 \to I(x,y,n) \land \\ \forall x,y,n,x',y',n'.\ I(x,y,n) \land x < n \land x' = x+1 \land y' = y+2 \land n' = n \to I(x',y',n') \land \\ \forall x,y,n.\ I(x,y,n) \land \neg(x < n) \to y = 2n \end{array}$$

The formula is satisfiable, the following interpretation is a model for φ_I:

$$I(x,y,n) \mapsto 2x = y \land x \leq n\ .$$

Until now invariant and rank-synthesis tools that have used this reduction to constraints have relied on special purpose algorithms for synthesizing these predicates. One method uses Farkas' lemma. Other methods use *templates* that specify a space of possible interpretations. The templates contain parameters that a separate solving method has to instantiate. Model-based quantifier instantiation, within Z3, integrates all decision procedures directly and can be used in lieu of these external methods.

Model-based quantifier instantiation works in the following way. Suppose we are given a constraint $\psi \land \forall x.\ \varphi[x]$, where ψ is quantifier-free and suppose M is an interpretation that satisfies ψ. We wish to check whether M also satisfies $\forall x.\ \varphi[x]$. Then let us replace the free symbols in φ by their values according to M. We call this new specialized formula φ^{M} and check if $\neg\varphi^{\mathsf{M}}[x]$ is satisfiable. If it is, then there is an extension M' of M that assigns x to a value v_x and M' satisfies $\neg\varphi[x]$. Consequently, M is not a model for $\forall x.\ \varphi[x]$. The idea is to instantiate the quantifier with a term t such that $t^{\mathsf{M}} = v_x$, and conjoin the instantiation to ψ to rule out this model that did not satisfy the quantifier. On the other hand, if $\neg\varphi^{\mathsf{M}}[x]$ is unsatisfiable, then M does indeed satisfy the entire formula. Note that there may be many terms t such that $t^{\mathsf{M}} = v_x$. One simple heuristic it to use a ground term t from ψ if such term exists. In [39], a strategy for selecting the term t is described, and it is shown that Model-based Quantifier Instantiation equipped with this strategy is a decision procedure for many fragments of first-order logic.

It is also possible to supply enough guidance to Z3 to build an interpretation for a predicate or function symbol I. This is achieved by supplying extra (template) equalities that restrict the possible interpretations for I. For the purpose of this example, the relevant template equalities could be:

$$\forall x,y.\ I(x,y,n) \leftrightarrow ax+by+cn = d \wedge a'x+b'y+c'n \leq d'$$

It is furthermore helpful restricting the variables a,a', b,b', c,c' to range over a finite domain. Then the instantiation of the templates remain decidable integer linear problems even when x,y,n are instantiated by symbolic terms. One way to restrict these variables to be finite domain is to let them range over bit-vectors.

Z3 recognizes that the template is essentially a *macro* for the predicate I. It uses this information to produce an interpretation for I.The notion of macros is more general than just equations; they include combining several conditional equalities as well.

We provide an encoding of the constraint satisfaction problem in SMT-LIB2 below.

```
(set-option :produce-models true)
(set-option :mbqi true)
(set-option :bv-enable-int2bv-propagation true)

(define-sort Char () (_ BitVec 8))
(declare-fun I (Int Int Int) Bool)

(declare-fun a1 () Char)
(declare-fun b1 () Char)
(declare-fun c1 () Char)
(declare-fun d1 () Int)
(declare-fun a2 () Char)
(declare-fun b2 () Char)
(declare-fun c2 () Char)
(declare-fun d2 () Int)

(define-fun linearT ((x Int) (y Int) (n Int) (a Char) (b Char) (c Char) (d Int)) Int
      (- (+ (* (bv2int a) x) (* (bv2int b) y) (* (bv2int c) n) d)
         (+ (* 128 x) (* 128 y) (* 128 n))))

(assert (forall ((x Int) (y Int) (n Int))
                (=> (and (= x 0) (= y 0) (>= n 0)) (I x y n))))

(assert (forall ((x Int) (y Int) (n Int))
                (=> (and (< x n) (I x y n)) (I (+ x 1) (+ y 2) n))))

(assert (forall ((x Int) (y Int) (n Int))
                (=> (and (>= x n) (I x y n)) (= y (* 2 n)))))

(assert (forall ((x Int) (y Int) (n Int))
                (iff (I x y n)
                  (and
                   (=  (linearT x y n a1 b1 c1 d1) 0)
                   (<= (linearT x y n a2 b2 c2 d2) 0)))))
(check-sat)
(get-model)
```

In this case Z3 produces the answer:

```
sat
(model
  (define-fun d1 () Int
    0)
  (define-fun c1 () (_ BitVec 8)
    #x80)
  (define-fun d2 () Int
    0)
  (define-fun c2 () (_ BitVec 8)
    #x1a)
  (define-fun b1 () (_ BitVec 8)
    #x60)
  (define-fun a1 () (_ BitVec 8)
    #xc0)
  (define-fun b2 () (_ BitVec 8)
    #xc0)
  (define-fun a2 () (_ BitVec 8)
    #x66)
  (define-fun I ((x!1 Int) (x!2 Int) (x!3 Int)) Bool
    (ite (and (= x!1 0) (= x!2 0) (= x!3 1)) true
    (ite (and (= x!1 (- 1)) (= x!2 1) (= x!3 0)) false
    (ite (and (= x!1 (- 1)) (= x!2 2) (= x!3 (- 1))) false
    (ite (and (= x!1 1) (= x!2 2) (= x!3 1)) true
    (ite (and (= x!1 0) (= x!2 0) (= x!3 0)) true
    (ite (and (= x!1 (- 3198)) (= x!2 (- 1704)) (= x!3 (- 254))) false
    (ite (and (= x!1 (- 192)) (= x!2 (- 80)) (= x!3 (- 192))) false
      (let ((a!1 (not (= (+ (* 2 x!1) (* (- 1) x!2)) 0)))
            (a!2 (not (>= (+ (* 13 x!1) (* 51 x!3) (* (- 32) x!2)) 0))))
        (not (or a!1 a!2))))))))))
)
```

This is close to the expected model. The first conjunct is the expected $2x = y$, the second reads $13x - 32y + 51n \geq 0$, but if we replace y by $2x$ we get $51n \geq 51x$, which is $n \geq x$ in disguise.

Model-based Quantifier Instantiation is a relatively new feature in Z3. The combination with complete quantifier instantiation [39] and macro detection facilities allows it to subsume several known decision classes, including EPR, the Array Property Fragment [9] and pointer data-structures [27]. An interesting challenge is to develop efficient incremental techniques for applying model-based quantifier instantiation.

3 Bit-precise, scalable analysis with PREfix

Many invariant generation techniques take as starting point solvers for the domains of mathematical integers and reals. This is at best imprecise in the context of software analysis. This section highlights an experience with integrating Z3 with the static analysis tool PREfix [10] for bit-precise static analysis [10].

Since 1999, PREfix has been used at Microsoft to analyze C/C++ production code. It relies on an efficient custom constraint solver, but addresses bit-level semantics only partially. On the other hand, Z3 supports precise machine-level semantics for integer arithmetic operations. The integration of PREfix with Z3 allows uncovering software bugs that could not previously be identified, in particular integer overflows. These typically arise when the programmer wrongly assumes mathematical integer semantics, and they are notorious causes of buffer overflow vulnerabilities in C/C++ programs. We ran our integration during the spring of 2009 over several projects from the Windows 7 code base and we uncovered a number of bugs related to integer overflows.

Let us give a simple example of a buffer overflow that was discovered using Z3/PREfix.

```
ULONG AllocationSize;
while (CurrentBuffer != NULL) {
    if (NumberOfBuffers > MAX_ULONG / sizeof(MyBuffer)) {
        return NULL;
    }
    NumberOfBuffers++;
    CurrentBuffer = CurrentBuffer->NextBuffer;
}
AllocationSize = sizeof(MyBuffer) * NumberOfBuffers;
UserBuffersHead = malloc(AllocationSize);
```

Program 3.1: Allocating a vector of buffers

The semantics of multiplication in C is modulo 2^{32} (on DWORDs). Therefore, multiplication can overflow. The if statement does protect from an integer overflow in the multiplication

$$\texttt{sizeof}(\texttt{MyBuffer}) * \texttt{NumberOfBuffers},$$

but as `NumberOfBuffers` is incremented just before the loop exits, the test is ineffective. The resulting buffer can therefore be allocated with fewer bytes than anticipated. A buffer overflow will happen when the buffer is later accessed at positions beyond the allocation boundary.

4 Test-case generation using Spec Explorer

The Spec Explorer tool grew out of efforts at Microsoft Research for developing model-based design and test tools. The development of Spec Explorer moved to the Protocol Test Team in 2007 to provide testing support for the 250+ protocol documents that Microsoft furnished to the European Commission and the Department of Justice. The protocol test challenge has provided a flourishing environment for the development of Spec Explorer. One use of Z3 in the context of Spec Explorer is for generating pair-wise independent test inputs for input/output specifications with complex constraints [18]. It can also use Z3 as part of its state-space exploration engine of model-based test programs.

Let us illustrate how Z3 can be used for generating pair-wise independent tests with arbitrary constraints. When there are no side-constraints there are explicit enumeration methods for pairwise independent testing. The paper [8] examines several of these methods.

We will use a small artificial example. Suppose we have a helicopter that can fly in any direction, horizontally (with angle h) or vertically (with angle v), and it can fly backwards or forwards with a maximal velocity w. We are interested in testing flight-paths of the helicopter over the sphere with radius w and we are interested in testing different combinations of horizontal and vertical directions. The helicopter is allowed to fix a horizontal and vertical direction and then take two legs along the chosen direction. We can fix two ranges for each choice. We can constrain these choices as follows:

$$\begin{array}{ll}
up \rightarrow 0 \leq v \leq 90 & \neg up \rightarrow -90 \leq v \leq 0 \\
right \rightarrow 0 \leq h \leq 90 & \neg right \rightarrow -90 \leq h \leq 0 \\
forward_1 \rightarrow 0 \leq \ell_1 \leq w & \neg forward_1 \rightarrow -w \leq \ell_1 \leq 0 \\
forward_2 \rightarrow 0 \leq \ell_2 \leq w & \neg forward_2 \rightarrow -w \leq \ell_2 \leq 0
\end{array}$$

We can add other constraints, for example constraining the length of the traveled path to be at least $w/2$ and at most w:

$$w/2 \leq |\ell_1 + \ell_2| \leq w$$

This leaves four degrees of freedom and a total of $2^4 = 16$ tests. The number of tests grows exponentially in the number of degrees of freedom. We can explore a much smaller set of tests by restricting the search for *pair-wise* independent tests by seeking a set of tests that cover each pair-wise combination of the $\binom{4}{2} = 6$ pairs $(up, right)$, $(up, forward_1)$, $(up, forward_2)$, $(right, forward_1)$, $(right, forward_2)$, $(forward_1, forward_2)$.

The algorithm used in [18] for enumerating pair-wise independent choices involves also a notion of seed and other concepts, but we will here in the interest of keeping the exposition simple provide a basic algorithm for choice enumeration. For this purpose let us introduce a propositional variable $c_{(p,q)}$ for each of the 6 choices. For example with the choice $(up, right)$ we introduce the variable $c_{(up,right)}$. The set of choice variables are called *choices* and we assert

$$\bigvee_{c \in choices} c$$

as our objective is to force at least one new pair to be covered during each test case. We can reduce the number of test cases by *maximizing* the set of new choices that are covered with one test. This can be achieved by adapting algorithms for MaxSAT in the context of SMT. The Z3 distribution comes with two sample programs that compute MaxSAT [1].

We can now enumerate new pairs by successively adding constraints that enforce that a new assignment is chosen. We sketch an algorithm below. It first checks whether the current constraints are satisfiable, and if it is the case maximizes the number of propositional variables in *choices* that are satisfied in the assignment. Then, for each pair p, q and associated proposition $c_{(p,q)}$ it evaluates p and q in the current model. Then it adds a constraint that forces $c_{(p,q)}$ to be *false* whenever a subsequent model evaluates p and p to the same value.

```
while (null != $model \leftarrow$ MaxSAT($choices$)) {
    for $c_{(p,q)} \in choices$ {
        $v_p \leftarrow model \rightarrow eval(p)$;
```

[1] `http://research.microsoft.com/en-us/um/redmond/projects/z3/group__maxsat__ex.html`

```
        $v_q \leftarrow model \rightarrow eval(q)$ ;
        assert $p \simeq v_p \wedge q \simeq v_q \rightarrow \neg c_{(p,q)}$
    }
}
```

We encoded this algorithm and with $w = 100$, Z3 produces the following six (instead of 16) test vectors:

```
(= h -1)     (= h 0)      (= h -1)     (= h 0)      (= h 0)      (= h -1)
(= v -1)     (= v 0)      (= v -1)     (= v 0)      (= v -1)     (= v 0)
(= l1 -51)   (= l1 51)    (= l1 51)    (= l1 -1)    (= l1 1)     (= l1 -49)
(= l2 1)     (= l2 -1)    (= l2 -1)    (= l2 51)    (= l2 49)    (= l2 -1)
```

We see that some pairs are covered more than once, but this is unavoidable.

The constraints can of course also have an effect on how many pair-wise independent tests are exercised. Suppose we add

$$\ell_1 < 0 \vee \ell_2 \geq 0,$$

then solutions where ℓ_1 is non-negative and ℓ_2 is negative are ruled out. With this constraint Z3 generates 5 test vectors.

```
(= h -1)     (= h 0)      (= h -1)     (= h 0)      (= h -1)
(= v -1)     (= v -1)     (= v 0)      (= v 0)      (= v -1)
(= l1 49)    (= l1 -49)   (= l1 -51)   (= l1 49)    (= l1 -51)
(= l2 1)     (= l2 -1)    (= l2 -1)    (= l2 1)     (= l2 1)
```

5 Several Other Tools and Applications

There are many other tools and applications of Z3. Several of these are surveyed in other places [6, 14] and we will not repeat a detailed treatment of these here, but only summarize some developed at Microsoft. The Static Driver Verifier tool SDV [1] uses Z3 to extract and check a finite state abstraction of programs. Windows 7 ships with SDV 2.0 using Z3. SDV is available for external parties writing drivers and it has identified hundreds of bugs in internal drivers. SLAM is related to the BLAST tool [23]. They use SMT solvers to help build a finite abstraction (a Boolean program). Newer tools refine also effectively build finite state abstractions using SMT solvers, but use internally different algorithms. The SLAyer tool [4] also targets device drivers, but uses an engine based on separation logic to help find memory errors (bugs that involve pointer-de-referencing). Yogi [19, 28] uses Z3 as part of the DASH/Synergy algorithms to refine abstract states for a program. An abstract state is a control location (program counter) together with a formula that summarizes a set of states. The states are connected by transitions that correspond to the control flow. The abstract states and transitions are refined by computing weakest pre-conditions. The weakest pre-conditions are simplified using models: a symbolic simulation of a candidate counter-example is used to prune case analysis for may-aliases. Yogi is not the only tool that takes advantage of the additional information available from dynamic symbolic execution. The SAGE and Pex [17] tools realize *smart white-box fuzzing*. SAGE and Pex collect explored program paths as formulas and use Z3 to identify new test inputs that forces execution into new branches. SAGE is used internally at Microsoft as part of a substantial security testing effort, and Pex is available for .NET-based development in the form of a Visual Studio power-tool.

Model programs are behavioral specifications that can be described succinctly and at a high-level of abstraction as Abstract State Machines or ASMs. ASMs can be represented as guarded commands encoded as logical formulas. Furthermore, the abstract data-types typically used in abstract state machine descriptions can often be directly encoded using theories for arrays, sets and bags. The use of bounded model checking techniques and SMT solvers is investigated in a sequence of recent papers. Bounded model program checking (BMPC) problems are investigated in [36, 37, 33]. Bounded Conformance Checking [26] is a variant of BMPC where it is checked if two model programs are related using a refinement relation. The Bounded Input Output Conformance Problem [25] checks if programs are input-output conformant (ioco). It can be checked directly for input-output model programs or reduced to BMPC. Regular and context-free string automata, regular string transducers and regular tree automata and transducers can be treated as useful special cases of abstract state machines. Unlike general abstract state machines, several decision problems, such as language inclusion, equivalence and idempotence under composition are decidable. Similar to general abstract state machines, it is possible to exploit symbolic representations of the transition relation, as pursued in [34, 24, 32, 7, 35].

Program verification is a natural stronghold for the use of SMT solvers. The programming system Spec# [2] integrates contracts for extended type safety. To generate verification conditions, Spec# programs are translated into a low-level procedural language Boogie [15], which is used for generating verification conditions (logical formulas) that are handled by Z3. Boogie can also be used in stand-alone mode and several other tools described next are based on Boogie. HAVOC, the Heap-Aware Verifier for C Programs uses the same Boogie verification condition generator, but targets extended type safety and heap properties of low level code [30, 13]. The Verifying C Compiler system [16] uses Boogie just like Spec# and HAVOC, but targets more ambitious functional correctness properties of the Viridian Hyper-Visor. The Hyper-V is a relatively small (100K lines) operating system layer. The VCC system was analyzed in multi-year joint project between Microsoft Research EMIC in Aachen and Saarbrücken. Substantial portions of the Hyper-V were verified. The system F7 [3] checks refinement types of F7 programs. It produces verification conditions directly as formulas which are then processed by Z3. F7 is an extension of F#. The FINE tool [11] also integrates refinement types. It adds certificates and features for refinement. Chris Hawblitzel has used Boogie directly to encode low-level assembly level components of the Singularity research operating system kernel. Using this approach, he has verified several different garbage collector implementations [22]. The project also includes verifying low-level (assembly-level) operating system kernel code [38].

6 Conclusions

The leading SMT solver, Z3, is developed at Microsoft Research. It is currently used in an array of products and research proto-types at Microsoft. The rich and expressive ways of interfacing with Z3 allows for building new and diverse set of tools on top of it. We believe it presents a growing set of opportunities for building new tools for software analysis and development. We are constantly encouraging and looking for new and useful ways to apply the technology underlying Z3.

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, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *CSF*, pages 17–32. IEEE Computer Society, 2008.

[4] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*. Springer, 2011.

[5] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2007.

[6] Nikolaj Bjørner and Leonardo Mendonça de Moura. Tapas: Theory combinations and practical applications. In Joël Ouaknine and Frits W. Vaandrager, editors, *FORMATS*, volume 5813 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2009.

[7] Nikolaj Bjørner and Margus Veanes. Symbolic transducers. Technical Report 2011-3, January 2011.

[8] Andreas Blass and Yuri Gurevich. Pairwise testing. *Bulletin of the EATCS*, 78:100–132, 2002.

[9] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[10] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[11] Juan Chen, Ravi Chugh, and Nikhil Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In Zorn and Aiken [40], pages 412–423.

[12] Michael Colón. Schema-guided synthesis of imperative programs by constraint solving. In Sandro Etalle, editor, *LOPSTR*, volume 3573 of *LNCS*, pages 166–181. Springer, 2004.

[13] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In Shao and Pierce [29], pages 302–314.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Bugs, moles and skeletons: Symbolic reasoning for software development. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 2010.

[15] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research, 2005.

[16] E. Cohen and M. Dahlweid and M. Hillebrand and D. Leinenbach and M. Moskal and T. Santen and W. Schulte and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOL*, 2009.

[17] 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.

[18] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. Interaction coverage meets path coverage by smt constraint solving. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *TestCom/FATES*, volume 5826 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2009.

[19] 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.

[20] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 281–292. ACM, 2008.

[21] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Zorn and Aiken [40], pages 292–304.

[22] Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In Shao and Pierce [29], pages 441–453.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

[24] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2011.

[25] Margus Veanes and Nikolaj Bjørner. Input-Output Model Programs. In *ICTAC*, 2009.

[26] Margus Veanes and Nikolaj Bjørner. Symbolic Bounded Conformance Checking of Model Programs. In *PSI*, 2009.

[27] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 476–490. Springer, 2005.

[28] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.

[29] Zhong Shao and Benjamin C. Pierce, editors. *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM, 2009.

[30] Shuvendu K. Lahiri and Shaz Qadeer and Zvonimir Rakamarić. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In *CAV'09*. Sringer Verlag, 2009.

[31] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Vs3: Smt solvers for program verification. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 702–708. Springer, 2009.

[32] Margus Veanes and Nikolaj Bjørner. Symbolic tree transducers. In *PSI*, 2011.

[33] Margus Veanes, Nikolaj Bjørner, and Alexander Raschke. An SMT Approach to Bounded Reachability Analysis of Model Programs. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *FORTE*, volume 5048 of *LNCS*, pages 53–68. Springer, 2008.

[34] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *ICST*, pages 498–507. IEEE Computer Society, 2010.

[35] Margus Veanes, David Molnar, and Benjamin Livshits. Decision procedures for composition and equivalence of symbolic finite state transducers. Technical Report 2011-32, Microsoft Research, March 2011.

[36] Margus Veanes and Ando Saabas. On Bounded Reachability of Programs with Set Comprehensions. In *LPAR*, pages 305–317, 2008.

[37] Margus Veanes and Ando Saabas. Using satisfiability modulo theories to analyze abstract state machines (abstract). In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *LNCS*, page 355. Springer, 2008.

[38] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Zorn and Aiken [40], pages 99–110.

[39] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified SMT formulas. In *CAV*, 2009.

[40] Benjamin G. Zorn and Alexander Aiken, editors. *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. ACM, 2010.