Z3[10]: Applications, Enablers, Challenges and Directions

Nikolaj Bjørner and Leonardo de Moura

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

Abstract. Modern program analysis and model-based tools are increasingly complex and multi-faceted software systems. However, at their core is invariably a component using a logic for describing states and transformations between system states. Logic inferences engines are then critical for the functionality of these systems. A commonly adapted approach has been to use a custom solver, built and tailored for the specific application. Custom solvers come with custom limitations: extending and scaling these often require a high investment. Taking as a starting point the solver Z3, developed at Microsoft Research; we describe how an efficient, scalable and expressive solver for Satisfiability Modulo Theories (SMT) is part of changing this landscape. Tools can now use the SMT solver with advantage to solve logic-related problems at a relatively high-level of abstraction while attaining scalability and features that custom solvers would have to duplicate.
We summarize 10 current applications of the Z3 solver and relate these to 10 main technological enabling factors. With every application there is a new opportunity, and with every solution there is a new challenge problem. Thus, we also summarize 10 challenges and 10 aspiring directions in the context of Z3 in particular, and for SMT solvers in general.

1 Introduction

This paper takes as starting point several of the applications that currently use the SMT solver Z3 [15]. Figure 1 summarizes 10 current applications, 10 main technological enabling factors in Z3, 10 challenging problem areas, and 10 aspiring directions for future applications. The rest of the paper is devoted at introducing the current applications and point out where the enabling factors are used and where the challenging problem areas arise. Most of the 10 aspiring directions are of course left for future work. We will on the other hand devote most of our attention diving into a couple of the challenging problems for SMT solvers.

The scope of the paper is narrowly centered around Z3 and we don't attempt a comprehensive survey of other tools, methods and applications here. Several of the challenges and the aspiring directions are already being pursued in recent and even classical literature. The challenges mainly reflect what we see as valuable research directions for an SMT tool directed towards applications. We assume

basic knowledge of SMT terminology, but explain some of the applications in more detail.

	Applications	Enablers	Challenges	Directions
1.	Dynamic Symbolic Execution	Finite model generation	Infinite model generation	Model-guided Dynamic Symbolic Execution
2.	Static Program Analysis	Succeed/fail fast	Truth maintenance	Static Analysis using Symbolic Execution
3.	Program Model Checking	Cooperating satisfiability and simplification	Proofs and interpolants	Game Programming
4.	Extended Static Checking	Back-jumping, learning strong lemmas [5]	Optimal lemma learning	Non-linear constraints for non-linear Systems
5.	Program Verification	Scaling quantifier instantiation with triggers [11]	Harnessing and understanding triggers	Real-time systems verification
6.	Model-based Testing	Using relevancy, avoiding irrelevancy [13]	Predicting and diagnosing search behavior	Program synthesis
7.	Combinatorial Test-input generation	Model-based theory combination [12]	Harnessing parallelism	Non-convex optimization
8.	Model-program analysis	A portfolio of decidable, succinct logics [16] [36]	a larger portfolio of succinct logics	Models for security and authentication
9.	Model-based Development	Free and absolutely free functions	Free functions in non-disjoint combinations	Biological Systems
10.	Quantitative Termination	Quantifier reasoning [17] [34][51]	Quantifier elimination	Qualitative Termination

Fig. 1. Ten Applications, Enablers, Challenges and Directions

The organization of the paper follows Figure 1. Dynamic Symbolic execution applications are described in Section 2. An essential feature of the SMT solver is the ability to generate finite models. Infinite models and models that mix finite and infinite domains pose new challenges. Recent and ongoing work in the context of Dynamic Symbolic Execution includes using models together with programs to guide test-input generation. An experience with static analysis is summarized in Section 3. In this context, the time to solve small queries proved highly critical, and poses a challenge for SMT solvers that are generally perceived efficient. The use of SMT solving in program model checking is sum-

marized in Section 4. Section 5 discusses extended static checking and verified software. Two main challenge problems are described in more depth here: the use and harnessing of triggers for quantifier instantiation, and the problem of generating precise theory lemmas in the context of SMT solvers. Section 6 discusses a range of diverse applications under a common *model-based* umbrella. We found a common trait in these applications that they use high-level theories for describing software. So a set of rich theories, facilities for succinctly encoding problems, and a good theory combination strategy was highly valuable. Section 7 points to uses of SMT solving for program termination and program complexity analysis.

2 Dynamic Symbolic Execution

Dynamic symbolic execution [20] has recently gained attention in the context of test-case generation and smart white-box file fuzzing. It extends static symbolic execution [27] by using concrete execution traces to obtain symbolic constraints. In order to explore a different execution path it suffices to modify one of the extracted symbolic traces by selecting and negating a branch condition. The modified path condition is checked for satisfiability. It is a logical constraint that uses theories typically supported in SMT solvers. A satisfying assignment to the modified path condition is a new input that can be used for steering execution into new paths.

```
int GCD(int x,int y) {
  while (true) {
    int m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

Program 2.1: GCD Program

To illustrate the basic idea of dynamic symbolic execution consider the greatest common divisor program 2.1. It takes the inputs x and y and produces the greatest common divisor of x and y.

Program 2.2 represents the static single assignment unfolding corresponding to the case where the loop is exited in the second iteration. The sequence of instructions is equivalently represented as a formula where the assignment statements have been turned into equations. The resulting path formula is satisfiable. One satisfying assignment is of the form:

$$x_0 = 2,\ y_0 = 4,\ m_0 = 2,\ x_1 = 4,\ y_1 = 2,\ m_1 = 0$$

Thus, the call `GCD(2,4)` causes the loop to be entered twice. Dynamic symbolic test-case generation is of course not limited to branch conditions that occur explicitly in the program. It can also be used for creating inputs that cause the program to enter an error state. For instance, it can be used to create inputs that cause remainder to be called with 0. Also, if we add a post-condition to

```
int GCD(int x0,int y0) {
  int m0 = x0 % y0;
  if (m0 == 0) return y0;
  x1 = y0;
  y1 = m0;
  int m1 = x1 % y1;
  if (m1 == 0) return y1;
}
```

$$
\begin{array}{ll}
(m_0 = x_0 \ \% \ y_0) \wedge \\
\neg(m_0 = 0) & \wedge \\
(x_1 = y_0) & \wedge \\
(y_1 = m_0) & \wedge \\
(m_1 = x_1 \ \% \ y_1) \wedge \\
(m_1 = 0)
\end{array}
$$

Program 2.2: GCD Path Formula

say that the result be non-negative we would discover that this implementation does not satisfy this property.

There are today several tools based on dynamic symbolic execution. Some of the earliest such tools include CUTE [37], DART, and Exe [8]. Microsoft has developed several related tools including SAGE, Pex, and Yogi [20], and Vigilante [9]. SAGE, for instance, is used with significant scale and success as part of the security testing efforts for Microsoft parsers for media formats. Pex integrates directly into Microsoft's Visual Studio development environment. It allows programmers to use parametrized unit testing directly during development.

Common to the tools is a reliance on a solver that can represent each program instruction as a, preferably equivalent, logical formula. Thus, the main requirements these tools impose are for a solver to provide (1) a way to encode machine arithmetic, (2) arrays and heaps and (3) generate finite models for driving execution.

The frontiers for dynamic symbolic execution include using information from static analysis for pruning the search space of the dynamic execution traces. The purpose it to prune exploration of redundant traces. Another frontier is combining dynamic symbolic execution with model-based techniques: library routines can be replaced by higher-level models, such that the solver can work on highly succinct representations of the actual execution traces that are encountered. For example, string library routines [6], can be modeled at the level of the theory of strings.

3 Static analysis

We recently [50] integrated the static program analysis tool PREfix [7] with Z3. The bit-vector theory in Z3 helped turning PREfix into a bit-precise static analysis tool. 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 using PREfix, 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. As anticipated, when running the integration on a large legacy code base for the next version of a Microsoft product we uncovered a number of bugs related to integer overflows.

An interesting lesson with the integration was that a bit-precise analysis tool could very easily wrongly flag false positive integer overflow bugs. The most common form of false positives were when overflows are parts of the program intent. Another form of false positives occurred in situations where pointer arithmetic on string pointers could potentially overflow in the presence of very large strings. Such situations are however often ruled out by simple limits of the virtual address space. So we developed useful filters for avoiding false positives and instead narrow the analysis on safety critical situations where bugs occur in spite of programmer intent.

Another valuable lesson was that a tool like PREfix relies on a massive set of relatively small queries. Straight-forward bit-blasting is prohibitively expensive in this context.

4 Program Model Checking

The tools SLAM/SDV [1] and Yogi [22] both use Z3 to extract and check a finite state abstraction of programs. They are currently mainly applied to driver verification. Yogi maintains symbolic states using formulas. It uses Z3's simplifier to prune redundancies from the states and uses satisfiability checking for testing subsumption between states. In contrast to BLAST [24], these tools do not use interpolation as part of the abstraction. There are several ways to extract interpolants from an SMT solver. One uses proof objects. Proof objects are now available in Z3 [14] and are currently being integrated with the Isabelle theorem prover to reconstruct LCF style proofs [1].

5 Extended Static Checking and The Ideal of Verified Software

The grand challenge of verified software [25] strives towards scientific ideals of verified software. The main idea of assigning meanings to programs using logical assertions was conceived by Floyd and Hoare in the 1960's. Such logical assertions, also known as *code contracts* [2], can be a *pre-condition assumption* that specifies how a procedure may be called, a *post-condition assertion* that specifies what the resulting state of a procedure call should satisfy, and *loop and object invariants* that capture properties of intermediary states. Core to assigning meanings is a verification condition generator that converts code annotated with contracts into logical formulas.

[1] http://www4.in.tum.de/~boehmes/
[2] http://research.microsoft.com/contracts/

The programming system Spec# [2] integrates contracts for extended type safety. To generate verification conditions, Spec# programs embed into a low-level procedural language Boogie. Boogie can also be used in stand-alone mode. Boogie and Spec# were developed based on experiences with the *extended static checker* ESC/Modula 3 and ESC/Java [19]. The system HAVOC [10][40] uses the same Boogie verification condition generator, but targets extended type safety and heap properties of low level code. Heap properties are also the subject of [41]. The system [3] checks refinement types of F_7 programs. It produces verification conditions directly into the Simplify format, which can be processed by Z3.

The VCC system [18] uses Boogie just like Spec# and HAVOC, but targets more ambitious functional correctness properties of the Viridian Hyper-V written in C. The Hyper-V is a relatively small (100K lines) operating system layer. The VCC system is used in a project involving around 20 researchers over 3 years for specifying and verifying aspects of the Hyper-V. It is thus used in one of the largest formal verification efforts to date. Serveral other operating system verification projects are surveyed in [28]. Finally, it is entirely possible to use Boogie directly as a target of assembly programming for operating systems [23].

5.1 Verification Condition Generation

A verification condition generator for an imperative language can be specified using Dijkstra's weakest liberal precondition predicate transformer *wp* with a few basic rules. Figure 2 contains the definition of *wp* for basic program statements.

$$
\begin{aligned}
wp(x := E, \varphi) &= \varphi[E/x] \\
wp(havoc(x), \varphi) &= \forall x.\varphi \\
wp(assert(P), \varphi) &= P \wedge \varphi \\
wp(assume(P), \varphi) &= P \rightarrow \varphi \\
wp(S;T, \varphi) &= wp(S, wp(T, \varphi)) \\
wp(S \square T, \varphi) &= wp(S, \varphi) \wedge wp(T, \varphi)
\end{aligned}
$$

$$
\begin{aligned}
&wp(\textbf{if } P \textbf{ then } S \textbf{ else } T, \varphi) = \\
&\quad wp(assume(P); S, \varphi) \\
&\quad \wedge\ wp(assume(\neg P); T, \varphi)
\end{aligned}
$$

$$
wp\left(\begin{array}{l}\textbf{while } P \textbf{ do} \\ \quad assert(R);\ S\end{array}, \varphi\right) =
$$

$$
wp\left(\begin{array}{l} assert(R); \\ havoc(x); \\ assume(R); \\ \textbf{if } P \textbf{ then} \\ \quad S; assert(R); \\ \quad assume(false) \end{array}, \varphi\right)
$$

Fig. 2. Weakest pre-conditions for program statements

The transformer $wp(S, \varphi)$ takes a program statement S and a predicate φ and produces the most permissive (weakest) predicate R, such that if R holds before executing S, then S does not enter an error state, and if S terminates, it

terminates in a state satisfying φ. Our imperative language contains a few non-standard constructs, such as assertions, assumptions and havoc. An assertion is a statement that causes the program to enter an error state if the execution that leads to the assertion statement does not satisfy the assertion formula. An assumption is a statement that can be used to filter out execution traces that violate the formula in the assumption. Havoc statements take one or more program variables as argument. It non-deterministically assigns a random value to the program variables. Furthermore, it may assign different values to the variables in different executions. Even though a program does not contain a havoc statement, this construct is useful for modeling while loops. While loops annotated with loop invariants reduce to the existing constructs. When producing the weakest liberal precondition for a while loop of the form **while** P **do** $assert(R)$; S where R is the loop invariant, it suffices to summarize the effect of each loop iteration by a single pass through the loop. The havoc construct is used to "fast forward" to an arbitrary iteration of the loop.

The program in Figure 3 shows a simple Spec# program that uses contracts. The object invariant for C states that the value of the attribute z remains non-negative. There is a pre-condition to the method M, which requires that the parameter a be positive. The object invariant corresponds to implicit pre- and

```
class C {
  private int z;
  invariant z >= 0;

  public C() { z = 1; }

  public void M(int a) {
    assume(a > 0);
    z = 100/a;
  }
};
```

$$wp\left(\begin{bmatrix} assume(z \geq 0); \\ assume(a > 0); \\ assert(a \neq 0); \\ z := 100/a; \\ assert(z \geq 0) \end{bmatrix}, true\right)$$
$$= (z \geq 0 \wedge a > 0) \rightarrow (a \neq 0 \wedge 100/a \geq 0)$$

Fig. 3. A program with contracts and a weakest pre-condition

post-conditions. To simultaneously check the object invariant and that division does not enter an error-state (by being invoked by $a = 0$), we compute the formula in the right column of Figure 3.

5.2 From Control Flow to Verification Conditions

The program 3 illustrated using a simple example how checking for the absence of errors and object invariants can be provided for during verification condition generation. For larger programs, the verification conditions may become much larger or more intricate. Boogie and ESC/Java use techniques [30] for controlling the number and size of verification conditions that are generated from a

single method body. These systems create one formula corresponding to multiple paths. A contrived method example is provided in Program 4 of a method body, that produces a simple but tricky verification condition. It is not unrea-

```
public void Diamond(int a) {
  if (p1(a))
      a++;
   else
      a--;
  }
  ...
  if (p100(a))
      a++;
   else
      a--;
  }

  assert(old(a) − 100 ≤ a ≤ old(a) + 100);
 }
};
```

$$\left[\begin{array}{l} \left(\begin{array}{c}(p_1(a_0) \wedge a_1 \simeq a_0 + 1)\\ \vee\ (\neg p_1(a_0) \wedge a_1 \simeq a_0 - 1)\end{array}\right)\\ \wedge \left(\begin{array}{c}(p_2(a_1) \wedge a_2 \simeq a_1 + 1)\\ \vee\ (\neg p_2(a_1) \wedge a_2 \simeq a_1 - 1)\end{array}\right)\\ \wedge \ldots\\ \wedge \left(\begin{array}{c}(p_{100}(a_{99}) \wedge a_{100} \simeq a_{99} + 1)\\ \vee\ (\neg p_{101}(a_{99}) \wedge a_{100} \simeq a_{99} - 1)\end{array}\right) \end{array}\right]$$
$$\rightarrow$$
$$a_0 - 10 \leq a_{100} \leq a_0 + 10$$

Fig. 4. A Diamond Program and corresponding verification condition

sonable to expect a verification condition generator to extract from the program a formula that is of the same size and equivalent to the the formula listed next to the program. While this particular example is contrived, verification conditions of these forms pose hard challenges for most of the main DPLL(T) based solvers. In a nutshell, DPLL(T) based solvers typically are only able to produce intermediary lemmas that use literals already present in the input. They tend to avoid producing lemmas using literals that don't come from the input. However, the only short (resolution) proofs of theorems like these require intermediary lemmas of the form $a_0 - 1 \leq a_1 \leq a_0 + 1$, $a_0 - 2 \leq a_2 \leq a_0 + 2$, etc. in order to converge to short proofs. The literals used in these lemmas don't occur in the original formula. Consequently, an interesting direction of research is to develop efficient techniques for improving lemma learning for DPLL(T)-based SMT solvers [49][5], and of course investigating alternatives to DPLL(T).

5.3 Programming Triggers

It is not uncommon that the assertions, assumptions and loop invariants use quantified formulas for expressing program properties. For example, the invariant for a loop that initializes an array A using the loop index i can be summarized as $\forall j : 0 \leq j < i \rightarrow \mathit{is\_initialized}(A[j])$. It is therefore a natural minimal requirement that theorem provers that support program verification be able to support formulas using quantifiers. A more distinguished source of quantified formulas in

program verification is from axiomatization of theories that arise from modeling how procedures affect the program heap and from modeling language properties, such as object-oriented type systems.

There are therefore several sources for integrating strong quantifier support in the context of SMT solvers. A current main approach to integrating quantifiers with SMT solving is by producing quantifier instantiations. The instantiated quantifiers are then quantifier free formulas that are handled by the main ground SMT solving engine. It is an art and craft to control quantifier instantiations to produce just the useful instantiations [35]. We will here touch on selected topics of quantifier instantiation for SMT solving in the context of program verification.

$$\mathcal{A}: \begin{array}{l} \forall t \;.\; t <: t \\ \forall t, u, v \;.\; t <: u \wedge u <: v \;\rightarrow\; t <: v \\ \forall t, u \;.\; t <: u \wedge u <: t \;\rightarrow\; t \simeq u \\ \forall t, u, v \;.\; t <: u \wedge t <: v \;\rightarrow\; u <: v \vee v <: u \\ \forall t, u \;.\; t <: u \;\rightarrow\; \mathit{Array}\langle t\rangle <: \mathit{Array}\langle u\rangle \end{array}$$

In object-oriented type systems used for Java and C# it is the case that objects are related using a single inheritance scheme. In other words, every object inherits from at most one unique immediate parent. It is also a commonly adapted feature that arrays behave in a monotone way with respect to inheritance. We can specify, using first-order axioms, that the inheritance relation is a partial order satisfying the single inheritance property, and that the array type constructor is monotone with respect to inheritance.

Let us call the axioms $\mathcal{A}$. Suppose we are provided a quantifier-free formula φ that uses only uninterpreted functions, the predicate $<:$ as well as the function *Array*. We wish to check when φ is consistent with respect to the background theory for single-inheritance provided above. A question is which instantiations $\mathcal{A}_{inst}$ of the quantified axioms are necessary and sufficient for reducing satisfiability checking to ground satisfiability. For controlling which instantiations of the axioms are produced let us consider an annotation of the axioms using *triggers*. A trigger annotated universal formula is a formula of the form

$$\psi_{annot}: \quad \forall \boldsymbol{x} \;.\; \{p_1(\boldsymbol{x}), \ldots, p_k(\boldsymbol{x})\} \;.\; \psi(\boldsymbol{x})$$

where $k \geq 1$, $p_1, \ldots, p_k$, are terms that contain all variables from $\boldsymbol{x}$. Given a model $\mathcal{M}$ of a quantifier free formula φ we say that φ is *saturated with respect to* ψ_{annot} if whenever there are sub-terms $t_1, \ldots, t_k$ in φ and a substitution θ, such that $p_1\theta^{\mathcal{M}} = t_1^{\mathcal{M}}, \ldots, p_k\theta^{\mathcal{M}} = t_k^{\mathcal{M}}$, then φ implies the conjunction $\psi\theta$.

Returning to the axioms for the single inheritance object system. The first question is what are the useful triggers? If we use the following trigger annotation:

$$\forall t, u \;.\; \{t <: u\} \;.\; t <: u \;\rightarrow\; \mathit{Array}\langle t\rangle <: \mathit{Array}\langle u\rangle$$

then given the formula $\varphi := t_0 <: u_0$ we end up generating an infinite number of instantiations:

$$
\begin{array}{l}
t_0 <: u_0 \;\rightarrow\; \mathit{Array}\langle t_0\rangle <: \mathit{Array}\langle u_0\rangle \\
\mathit{Array}\langle t_0\rangle <: \mathit{Array}\langle u_0\rangle \;\rightarrow \\
\quad \mathit{Array}\langle \mathit{Array}\langle t_0\rangle\rangle <: \mathit{Array}\langle \mathit{Array}\langle u_0\rangle\rangle \\
\mathit{Array}\langle \mathit{Array}\langle t_0\rangle\rangle <: \mathit{Array}\langle \mathit{Array}\langle u_0\rangle\rangle \;\rightarrow \\
\quad \mathit{Array}\langle \mathit{Array}\langle \mathit{Array}\langle t_0\rangle\rangle\rangle <: \mathit{Array}\langle \mathit{Array}\langle \mathit{Array}\langle u_0\rangle\rangle\rangle \\
\ldots
\end{array}
$$

We say that the triggers cause a *matching loop*.

$$
\mathcal{A}_{annot}: \quad
\begin{array}{l}
\forall t \;.\; \{t <: t\} \;.\; \; t <: t \\
\forall t, u, v \;.\; \{t <: u, u <: v\} \;. \\
\quad t <: u \wedge u <: v \;\rightarrow\; t <: v \\
\forall t, u \;.\; \{t <: u, u <: t\} \;. \\
\quad t <: u \wedge u <: t \;\rightarrow\; t \simeq u \\
\forall t, u, v \;.\; \{t <: u, t <: v\} \;. \\
\quad t <: u \wedge t <: v \;\rightarrow\; u <: v \vee v <: u \\
\forall t, u \;.\; \{\mathit{Array}\langle t\rangle <: \mathit{Array}\langle u\rangle\} \;. \\
\quad t <: u \;\rightarrow\; \mathit{Array}\langle t\rangle <: \mathit{Array}\langle u\rangle
\end{array}
$$

Let us instead consider the pattern annotations listed in $\mathcal{A}_{annot}$. We first observe that if $\mathcal{M}$ is a model of φ, then there is only a finite set $\mathcal{A}_{inst}$ of instantiations of $\mathcal{A}_{annot}$ such that the formula $\varphi \wedge \mathcal{A}_{inst}$ is saturated with respect to $\mathcal{M}$. This follows because the first four axioms only introduce at most all combinations $t <: u$, where t and u are existing terms. Instantiating the last axiom only introduces occurrences of $<:$ for smaller terms than these already present. We can also establish, using an ad hoc model construction, that consistency with respect to the saturation of $\mathcal{A}_{annot}$ implies ground consistency. It is of course a valuable research direction to automatically infer pattern annotations that are complete by construction, and infer also the corresponding models [51].

The choice of triggers is not always unique. We could have considered the following trigger for the monotonicity axiom:

$$\forall t, u \;.\; \{t <: u, \mathit{Array}\langle t\rangle, \mathit{Array}\langle u\rangle\} \;.\; t <: u \;\rightarrow\; \mathit{Array}\langle t\rangle <: \mathit{Array}\langle u\rangle$$

The resulting annotation still leads to a terminating and complete saturation, but the number of instances that it produces can vary significantly.

The example of the single-inheritance type system with monotonicity illustrated matching loops, performance trade-offs with different triggers, and completeness of trigger-based saturation. It shows that programming triggers can be *tricky*. Nevertheless, it appears to be quite useful. A decision procedure for the non-extensional theory of arrays can be encoded using axioms with triggers. A theory of reachability and linked lists has also been encoded using triggers in [29]. In the general case there are simple but useful ways to control quantifier instantiation using triggers. One approach is used in the verification of garbage

collectors [23]. It introduces an auxiliary predicate $\mathtt{T}(x)$, and the axiom $\forall x.\mathtt{T}(x)$. The quantifier triggers can then use the term $\mathtt{T}(p)$ to select patterns p for instantiation. The approach works as long as the encoding ensures that $\mathtt{T}$ is only used in positive contexts.

5.4 Annotation inference and strengthening

One of the most challenging problems in deductive program verification is to find inductive program invariants [4], [31]. The task is even more challenging when the inductive invariants require quantifiers. While the previous section illustrated how the SMT solver can handle quantified invariants it does not find these. The inference of auxiliary invariants is therefore an important challenge.

One technique tested in HAVOC [39] is the inference of auxiliary assertions by pruning a candidate conjunction of invariants until a fixed-point is reached. It is a cheaper and often as effective method as predicate abstraction used in software model-checking.

Another technique that was studied in [45] is to use test-case generation techniques to debug annotations with non-inductive quantified invariants. The idea is to extract path conditions from programs with (quantified) contracts and loop invariants, and use the dynamic symbolic execution techniques to find path conditions that violate the invariants. It allows one potentially to analyze test cases with a traditional debugger to determine the cause of the error; the developer may then correct the program or the contracts and repeat the process.

The VS3 project [43][42] uses Z3 to derive program invariants that involve quite complex formulas.

Nevertheless, the experience with scaling deductive program verification for invariants of large programs remains highly challenging. A particularly frustrating experience in the context of VCC has been when verifying a procedure body using pre-conditions imposed by calling contexts. When the pre-conditions are changed, also the entire procedure body needs to be re-established. In the scale of verification condition formulas taking several MB, the time to re-establish and fix annotations can be quite demanding.

6 Model-based techniques

Model-based techniques tend to use high-level description formalisms and abstract domains for describing systems. There are several systems in the space of model-based techniques at Microsoft using Z3 in crucial ways. We summarize some of these here.

Model Programs 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 (even though not all syntactically well-formed guarded command formulas are legal ASMs). 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 [48], [46], [47]. The case of *basic* model programs maps in general into Presburger arithmetic, but often directly into Z3's ground decidable fragment. Bounded Conformance Checking [33] 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 [32] checks if programs are input-output conformant (`ioco`). It can be checked directly for input-output model programs or reduced to BMPC.

SpecExplorer Model-based testing is used at a large scale at Microsoft in the context of the disclosure and documentation of Microsoft network protocols [21]. Z3 is currently being integrated as part of this effort for combinatorial test-input generation as well as generating model-based tests.

Model-based development The FORMULA [26] system uses a specification style inspired by Prolog for model-based design and development. Z3 is used as part of a symbolic execution engine. The execution engine relies on a custom quantifier elimination procedure for the theory of term algebras.

7 Qualitative and Quantitative Termination

The vast majority of analysis applications are devoted to *safety* properties of programs. They are concerned about whether a program can enter an error state or whether a program or a module satisfies a prescribed contract. Of equal importance are *liveness* properties: does a program terminate [3]? An even more timely quest is obtaining more qualitative information from programs, that is more precise information about the running time of programs.

8 Conclusion

We have summarized an array of applications currently being integrated with the SMT solver Z3. We also touched on two selected areas that pose significant challenges and opportunities for research on SMT solvers. There are several other challenges and future applications that we hope to describe and tackle in future work.

[3] http://www.foment.net/byron/fsharp.shtml

References

1. Tom Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram 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. Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and assertions. In Ugo Montanari and Francesca Rossi, editors, *CP*, volume 976 of *Lecture Notes in Computer Science*, pages 589–623. Springer, 1995.
5. Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura. Accelerating DPLL(T) using Joins - DPLL(⊔). In *Short papers at LPAR 08*, 2008.
6. Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2009.
7. 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.
8. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS*, pages 322–335, New York, NY, USA, 2006. ACM Press.
9. Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 319–328, New York, NY, USA, 2008. ACM.
10. Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. Unifying type checking and property checking for low-level code. In Shao and Pierce [38], pages 302–314.
11. Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT Solvers. In *CADE'07*. Springer-Verlag, 2007.
12. Leonardo de Moura and Nikolaj Bjørner. Model-based Theory Combination. In *SMT'07*, 2007.
13. Leonardo de Moura and Nikolaj Bjørner. Relevancy Propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.
14. Leonardo de Moura and Nikolaj Bjørner. Proofs and Refutations, and Z3 . In *IWIL*, 2008.
15. Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS 08*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.
16. Leonardo de Moura and Nikolaj Bjørner. Deciding Effectively Propositional Logic using DPLL and substitution sets. In Allesandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR 2008*, 2008.
17. Leonardo de Moura and Nikolaj Bjørner. Engineering DPLL(T) + saturation. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. 4th IJCAR*, volume 5195, pages 475–490, 2008.
18. Ernie Cohen and Michał Moskal and Wolfram Schulte and Stephan Tobies. A Precise Yet Efficient Memory Model For C. In *Proceedings of Systems Software Verification Workshop (SSV 2009)*, 2009. To appear.

19. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.
20. Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.
21. Wolfgang Grieskamp, Nicolas Kicillof, Dave MacDonald, Alok Nandan, Keith Stobie, and Fred L. Wurden. Model-based quality assurance of windows protocol documentation. In *ICST*, pages 502–506. IEEE Computer Society, 2008.
22. 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.
23. Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. In Shao and Pierce [38], pages 441–453.
24. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.
25. Tony Hoare. The Ideal of Verified Software. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 5–16. Springer, 2006.
26. Ethan K. Jackson and Wolfram Schulte. Model Generation for Horn Logic with Stratified Negation. In Suzuki et al. [44], pages 1–20.
27. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
28. Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.
29. Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using smt solvers. In George C. Necula and Philip Wadler, editors, *POPL*, pages 171–182. ACM, 2008.
30. K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
31. K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2005.
32. Margus Veanes and Nikolaj Bjørner. Input-Output Model Programs. In *ICTAC*, 2009.
33. Margus Veanes and Nikolaj Bjørner. Symbolic Bounded Conformance Checking of Model Programs. In *PSI*, 2009.
34. Maria Paola Bonacina and Christopher Lynch and Leonardo de Moura. On deciding satisfiability by DPLL($\Gamma + T$) and unsound theorem proving.
35. Michał Moskal. Programming with Triggers. Draft.
36. Nikolaj Bjørner and Joe Hendrix. Linear Functional Fixed-points. In *CAV*, 2009.
37. Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.
38. 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.
39. Shuvendu K. Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *CADE*. Springer Verlag, 2009. MSR-TR-2009-2012.
40. 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.

41. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *ECOOP*, 2009.
42. Saurabh Srivastava and Sumit Gulwani. Program Verification using Templates over Predicate Abstraction. In *PDLI*, 2009.
43. Saurabh Srivastava, Sumit Gulwani, and Jeffrey Foster. VS3: SMT Solvers for Program Verification. 2009.
44. Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors. *Formal Techniques for Networked and Distributed Systems - FORTE 2008, 28th IFIP WG 6.1 International Conference, Tokyo, Japan, June 10-13, 2008, Proceedings*, volume 5048 of *Lecture Notes in Computer Science*. Springer, 2008.
45. Dries Vanoverberghe, Nikolaj Bjørner, Jonathan de Halleux, Wolfram Schulte, and Nikolai Tillmann. Using dynamic symbolic execution to improve deductive verification. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *SPIN*, volume 5156 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 2008.
46. Margus Veanes, Nikolaj Bjørner, and Alexander Raschke. An SMT Approach to Bounded Reachability Analysis of Model Programs. In Suzuki et al. [44], pages 53–68.
47. Margus Veanes and Ando Saabas. On Bounded Reachability of Programs with Set Comprehensions. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 305–317. Springer, 2008.
48. 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 *Lecture Notes in Computer Science*, page 355. Springer, 2008.
49. Chao Wang, Aarti Gupta, and Malay Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 235–240, New York, NY, USA, 2006. ACM.
50. Yannick Moy and Nikolaj Bjørner and David Sielaff. Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis. Technical Report MSR-TR-2009-57, Microsoft Research, 2009.
51. Yeting Ge and Leonardo de Moura. Complete instantiation for quantified SMT formulas. In *CAV*, 2009.