Z3: An Efficient SMT Solver

Leonardo de Moura and Nikolaj Bjørner

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

Abstract. Satisfiability Modulo Theories (SMT) problem is a *decision problem* for logical first order formulas with respect to combinations of background theories such as: arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is a new and efficient SMT Solver freely available from Microsoft Research. It is used in various software verification and analysis applications.

1 Introduction

Satisfiability modulo theories (SMT) generalizes boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulas in these theories. SMT solvers enable applications such as extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains, to mention a few.

Z3 is a new SMT solver from Microsoft Research. It is targeted at solving problems that arise in software verification and software analysis. Consequently, it integrates support for a variety of theories. A prototype of Z3 participated in SMT-COMP'07, where it won 4 first places, and 7 second places. Z3 uses novel algorithms for quantifier instantiation [4] and theory combination [5]. The first external release of Z3 was in September 2007. More information, including instructions for downloading and installing the tool, is available at the Z3 web page: http://research.microsoft.com/projects/z3.

Currently, Z3 is used in Spec#/Boogie [2, 7], Pex [13], HAVOC [11], Vigilante [3], a verifying C compiler (VCC), and Yogi [10]. It is being integrated with other projects, including SLAM/SDV [1].

2 Clients

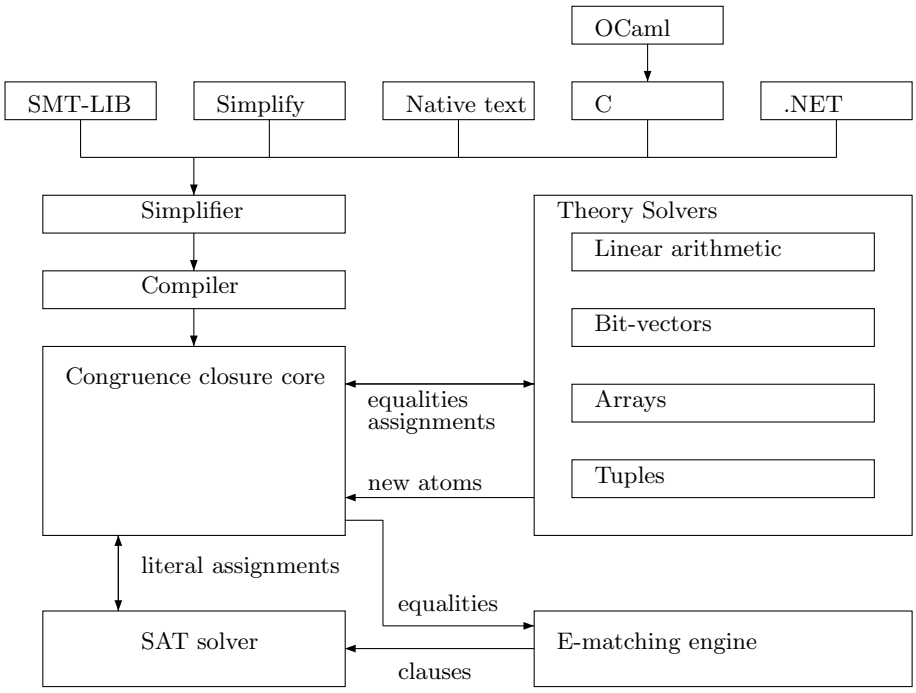
Before describing the inner workings of Z3, two selected uses are briefly described. Front-ends interact with Z3 by using either a textual format or a binary API. Three textual input-formats are supported: The SMT-LIB [12] format, the Simplify [8] format, and a low-level native format in the spirit of the DIMACS format for propositional SAT formulas. One can also call Z3 procedurally by using either an ANSI C API, an API for the .NET managed common language runtime, or an OCaml API.

Spec#/Boogie generates logical verification conditions from a Spec# program (an extension of C#). Internally, it uses Z3 to analyze the verification conditions, to prove the correctness of programs, or to find errors on them. The formulas produced by Spec#/Boogie contain universal quantifiers, and also use linear integer arithmetic. Spec# replaced the Simplify theorem prover by Z3 as the default reasoning engine in May 2007, resulting in substantial performance improvements during theorem proving.

Pex (Program EXploration) is an intelligent assistant to the programmer. By automatically generating unit tests, it allows to find bugs early. In addition, it suggests to the programmer how to fix the bugs. Pex learns the program behavior from the execution traces, and Z3 is used to produce new test cases with different behavior. The result is a minimal test suite with maximal code coverage. The formulas produced by Pex contains fixed-sized bit-vectors, tuples, arrays, and quantifiers.

3 System Architecture

Z3 integrates a modern DPLL-based SAT solver, a *core theory solver* that handles equalities and uninterpreted functions, *satellite solvers* (for arithmetic, arrays, etc.), and an *E-matching abstract machine* (for quantifiers). Z3 is implemented in C++. A schematic overview of Z3 is shown in the following figure.



Simplifier Input formulas are first processed using an incomplete, but efficient simplification. The simplifier applies standard algebraic reduction rules, such as $p \wedge \texttt{true} \mapsto p$, but also performs limited contextual simplification, as it identifies equational definitions within a context and reduces the remaining formula using the definition, so for instance $x = 4 \wedge q(x) \mapsto x = 4 \wedge q(4)$. The trivially satisfiable conjunct $x = 4$ is not compiled into the core, but kept aside in the case the client requires a model to evaluate x.

Compiler The simplified abstract syntax tree representation of the formula is converted into a different data-structure comprising of a set of clauses and congruence-closure nodes.

Congruence Closure Core The congruence closure core receives truth assignments to atoms from the SAT solver. Atoms range over equalities and theory specific atomic formulas, such as arithmetical inequalities. Equalities asserted by the SAT solver are propagated by the congruence closure core using a data structure that we will call an E-graph following [8]. Nodes in the E-graph may point to one or more theory solvers. When two nodes are merged, the set of theory solver references are merged, and the merge is propagated as an equality to the theory solvers in the intersection of the two sets of solver references. The core also propagates the effects of the theory solvers, such as inferred equalities that are produced and atoms assigned to `true` or `false`. The theory solvers may also produce fresh atoms in the case of non-convex theories. These atoms are subsequently owned and assigned by the SAT solver.

Theory Combination: Traditional methods for combining theory solvers rely on capabilities of the solvers to produce all implied equalities or a pre-processing step that introduces additional literals into the search space. Z3 uses a new theory combination method that incrementally reconciles models maintained by each theory [5].

SAT Solver Boolean case splits are controlled using a state-of-the art SAT solver. The SAT solver integrates standard search pruning methods, such as two-watch literals for efficient Boolean constraint propagation, lemma learning using conflict clauses, phase caching for guiding case splits, and performs non-chronological backtracking.

Deleting clauses: Quantifier instantiation has a side-effect of producing new clauses containing new atoms into the search space. Z3 garbage collects clauses, together with their atoms and terms, that were useless in closing branches. Conflict clauses, and literals used in them, are on the other hand not deleted, so quantifier instantiations that were useful in producing conflicts are retained as a side-effect.

Relevancy propagation: DPLL(T) based solvers assign a Boolean value to potentially all atoms appearing in a goal. In practice, several of these atoms are *don't cares*. Z3 ignores these atoms for expensive theories, such as bit-vectors, and inference rules, such as quantifier instantiation. The algorithm used for discriminating relevant atoms from don't cares is described in [6].

Quantifier instantiation using E-matching Z3 uses a well known approach for quantifier reasoning that works over an E-graph to instantiate quantified vari-

ables. Z3 uses new algorithms that identify matches on E-graphs incrementally and efficiently. Experimental results show substantial performance improvements over existing state-of-the-art SMT solvers [4].

Theory Solvers Z3 uses a linear arithmetic solver based on the algorithm used in Yices [9]. The array theory uses lazy instantiation of array axioms. The fixed-sized bit-vectors theory applies bit-blasting to all bit-vector operations, but equality.

Model generation Z3 has the ability to produce models as part of the output. Models assign values to the constants in the input and generate partial function graphs for predicates and function symbols.

4 Conclusion

Z3 is being used in several projects at Microsoft since February 2007. Its main applications are extended static checking, test case generation, and predicate abstraction.

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. 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.
4. L. de Moura and N. Bjørner. Efficient E-matching for SMT Solvers. In *CADE'07*. Springer-Verlag, 2007.
5. L. de Moura and N. Bjørner. Model-based Theory Combination. In *SMT'07*, 2007.
6. L. de Moura and N. Bjørner. Relevancy Propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.
7. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research, 2005.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
9. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV'06*, LNCS 4144, pages 81–94. Springer-Verlag, 2006.
10. 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.
11. S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *POPL'2008*, 2008.
12. S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.
13. N. Tillmann and W. Schulte. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE software*, 23:38–47, 2006.