Snippet of Kyle's Reading List

Kyle Dewey

August 28, 2018

1 Dealing with Loops for Dynamic Invariant Detection

- Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM

  Attempts to automatically generate partial loop invariants by inferring what is changed concretely per loop iteration.

- Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 225–236, New York, NY, USA, 2009. ACM

  Models loop-dependent variables as symbolic variables to better handle loops.

2 Dealing with Functions for Dynamic Invariant Detection

- Patricia Mouy, Bruno Marre, Nicky Willams, and Pascale Le Gall. Generation of all-paths unit test with function calls. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 32–41, Washington, DC, USA, 2008. IEEE Computer Society

  Attempts to abstract functions away in symbolic execution to make interprocedural analysis more feasible. Related to [24]. Major difference: summaries are provided statically by previous runs / other techniques as opposed to being dynamically generated.

- Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM

  Summarizes functions to make interprocedural analysis cheaper for symbolic execution. These summaries are based on concrete executions of the functions. If we ever encounter an input to the function that is symbolically idential to a previous input, then we need only return that input's symbolic value.
  NOTE TO SELF: What about non-local side effects?

  This whole process is very similar to deriving the invariants generated by DySy [15], except that these invariants can later be used in place of function calls.

3 Parallelizing DSE/SE

- J.H. Siddiqui and S. Khurshid. Parsym: Parallel symbolic execution. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 1, pages V1–405 –V1–409, oct. 2010

  Parallelizing concolic execution.

- Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 523–536, New York, NY, USA, 2012. ACM

  Consider the entire DSE exploration space as a single tree of paths. Note that this is applicable only to concolic execution. In practice, this tree can be infinite, but assume that it's not. The basic idea is that we can partition this tree recursively into sections, each of which can be tested independently of each other. This allows for a parallel DSE. Partitions can be made either statically or dynamically. Given that we do not know the structure of this tree a priori in general, these partitions seem to be best made dynamically. Implemented on top of KLEE [10]. Showed 6.6X average speedup for 10 workers, though greater than 10X was possible likely due to efficient caching benefits across multiple executions due to KLEE. They generate path conditions from a single concrete input. This allows them to store only inputs as opposed to entire paths, which can dramatically decrease the size of what needs to be stored to encode a path (i.e. a path can be derived unambiguously through concrete execution of the stored inputs). Defines an ordering between all paths by considering the true side of a branch to the false side. Considers two sets of inputs to be equivalent if they result in the same path condition (in the same vein as [48], though they don't cite [48]).

The basic idea is that the start and end of a range are defined by τ_{start} and τ_{end}, respectively. The current test is defined simply by τ. If at any point $\tau \implies \tau_{end}$, then we have reached the end of the range that we were intended to execute. Additionally, for any branch point discovered by analysis of the range between τ_{start} and τ_{end}, we can dynamically split it up, adding additional ranges between τ_{start} and τ, and between τ and τ_{end}. This does cause for some redundant execution (i.e. after this split the test case defined by τ will be executed twice, once at the end of the $[\tau_{start}, \tau]$ range and again at the beginning of the $[\tau, \tau_{end}]$ range), though the extra computation was not significant in the paper's evaluation. The rest of the paper goes over some various applications of this ranged SE technique.

4 DSE / SE in Practice

- Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20. Springer, 2004

  How SLAM came to be and how it has been used within Microsoft.

- Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM

  A survey of different symbolic execution techniques that have been used in practice.

5 Building Tests using DSE / SE

- Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag

  Discussion of Pex for .NET testing. Builds an actual test suite for a given program. Uses Z3 [19] on the backend as an SMT solver. According to [33], it will encode floating point operations as uninterpreted functions (coined *custom arithmetic solvers*) instead of doing them directly in Z3 in an (unsound) attempt to avoid some of the problems mentioned in [6,36]. Also a set of really good references.

Features advanced search strategies that take program structure into account. According to the authors, this strategy is more effective even than BFS. Can also handle pointer arithmetic, though at the cost of soundness. If a handful of assumptions are true, it can actually be used for verification purposes. Other than common bugs (e.g., index out of bounds, null pointer dereferencing, assertion violations), does not handle the oracle problem.

- Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag

  Symbolic execution for specifically object-oriented programs. Actually builds stateful unit tests that will explore different code paths.

- Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 243–257, Washington, DC, USA, 2006. IEEE Computer Society

  Generates disks instead of individual inputs, specifically for testing filesystem code. Uses EXE [11] as part of this process. Found that few loops involve symbolic constraints in the context of filesystem code. Internally uses STP [23] for constraint solving, and models all program data as bitvectors to allow for the extensive sorts of type-unsafe casting seen in the C code of filesystems. Will replace division and modulo by symbolic values with an appropriate bit shift or bitmask, since STP does not support these operations. Good real-life application; they had to run nearly the entire Linux kernel with `exe-cc` [11]. Handles loops in a fairly naive way, using the usual k-limiting. They attempt to choose paths that are of greater value with some heuristics, but overall it doesn't seem very sophisticated. Were able to find bugs in all the filesystems tested.

- Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM

  Basic observation: compilers and interpreters are very difficult to test with DSE, as we end up symbolically going through the parser. This dramatically increases the search space in a way that we typically find uninteresting; that is, if we are interested in testing the internals of a compiler, we probably don't care about bugs in the parser. The idea here is to augment the usual constraint solver with the capability to understand the grammar of the input language, such that we only generate well-formed inputs. This avoids performing symbolic execution over raw programs, instead focusing things on ASTs.

Essentially, this treats whole tokens symbolically, as opposed to individual bytes. Onto the path condition, a precondition is added which roughly states that the input is parseable. There are more details in the paper pertaining to exactly how they implemented this; they implemented a custom decision procedure for this part, but the idea is fundamentally the same. Evaluation-wise, the data shows that in comparing grammar-based black-box generation to their technique, slightly more inputs reach the code generator, and overall they achieve 81.5% code generator coverage with their technique versus 61.2% code generator coverage for grammar-based black box techniques.
NOTE TO SELF: In both cases, one would expect that 100% of inputs would reach the code generator, but their grammar definition is an over-approximation. For example, they will emit `break` statements even if they are not in a loop. As for the higher code coverage of whitebox techniques here, it should be noted that they used a random search strategy for the grammar-based end of things. It seems with a bit more smarts on the black-box end we might be able to push the limit more.

6 Assorted Existing DSE / SE Frameworks

- Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc

  The original SLAM paper.

- Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association

  Introduces KLEE, a symbolic interpreter over LLVM [34] bitcode.

- Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *NASA Formal Methods*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 121–125, 2009

  Adds concolic execution to Java PathFinder [60].

- Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008

  Discussion of SAGE, which is another concolic framework used extensively with Microsoft.

- Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM

  An early application of DSE to real-world scenarios. Claims that running many tests can be more expensive than static analysis, at least for their application. Uses concolic execution. If it finds something that the constraint solver cannot handle, then it will fall back to the concrete value (i.e. multiplication involving two symbolic variables). Claims that it usually has to fall back to a concrete value somewhere when it throws information to the constraint solver.

- Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM

  An analysis framework for C that can handle all the values seen in C, except for floating-point operations. Implemented as a compiler for C that adds all the neccessary DSE instrumentation to the code. Was codesigned with STP [23] and internally uses it heavily, representing most everything using bit-vectors to allow for all sorts of nasty C-style bit manipulation operations. Can reason about what it terms as "symbolic pointers", which seem to be normal pointers except they make the typical assumption that the programmer will not attempt to escape from an object. Specifically, given a concrete array a, a symbolic integer i with the constraint $i \geq 0 \land i \geq 10$, and a conditional **if** $(a[i] == 10)$..., it is able to reason about all the possible different positions in the array (concretely, this constraint is equivalent to $a[0] == 10 \lor a[1] == 10 \lor ... \lor a[10] == 10$).

- Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM

  Uses concolic execution to find SQL injection attacks in database applications. They get symbolic values via a special `input()` function. Features a string solver that can handle **unbounded** strings over regular languages. They deal with both strings and integer arithmetic, but a look at their constraint language reveals that arithmetic is extremely limited, and they are forced to resort to concrete values likely a lot. They break an initial constraint into both a string-only and an arithmetic-only constraint, and solve them independently. Given the inherient lossiness of this process, this explains why their constraint language is so limited (it doesn't even feature integer arithmetic or string concatenation). They claim that this isn't a problem in their applications, which may be true specifically for generating SQL queries. Tightly coupled with a database. Finding a satisfying assignment for a constraint involves both program variables and database entries; their framework actually inserts database entries as it runs (the database can be viewed as a special kind of auxilliary store). States that determining satisfiability of strings in the precense of a `length` function is an open problem, citing [20, 47] as evidence.

7 When and How Often Daikon [22] is Wrong

- Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200, New York, NY, USA, 2011. ACM

  Used Daikon with custom templates to infer contracts for Eiffel. Based on their analysis the resulting contracts are between 88-100% correct, usually leaning towards the mid to high 90's. Note that since they were using templates, this is likely biased to be correct whenever they find a matching template.

- Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 188–198, New York, NY, USA, 2012. ACM

  Given programs and Daikon-inferred invariants, users were asked to classify them as either correct or incorrect. The authors found that correct invariants are misclassified as incorrect 9.1% to 31.7% of the time, and incorrect invariants were misclassified as correct an alarming 26.1% to 58.6% of the time. This motivates that Daikon alone is not good enough for generating invariants for its own sake, despite the fact that researchers frequently jump to Daikon when they want dynamic invariants (evidenced by many of the papers that cite both [22] and [15]).

- Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 93–104, New York, NY, USA, 2009. ACM

Compared programmer-provided invariants to those generated by Daikon in Eiffel code. Found that Daikon will produce about 5x as many assertions as do programmers. However, of these assertions, roughly a third of them are either incorrect or irrelevant. Additionally, the Daikon assertions seem pretty weak, as implied by the fact that it can find only around 60% of the assertions specifed by the programmers. The conclusions here are in the same vein as [57]: Daikon alone is not strong enough to provide sound enough and precise enough invariants to be useful just for the point of deriving invariants.

- Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: an empirical evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20, November 2002

  Many researchers claim that determining what property to check is a more difficult problem than checking the property itself. Compares a custom implementation of Houdini (static invariants) to Daikon (dynamic invariants), within the ESC/Java framework (Java with a layer for unsound verification). Their own Houdini implementation was used since it is not available; they call it Whodini. Users were just given the Daikon invariants. Claims that when Daikon is given an adequate test suite, it gives good annotations and inferred invariants, citing [42, 43] as evidence. (Of course, it never defines what it means by an "adequate test suite". It was stated that tests were run for several minutes; these are presumed random.)

  The actual study seems a bit biased; they measured how far off users were from a correct answer based on the number of edits needed by the authors to make it correct. Measured number of annotations (which can be reduced without any loss of meaning via conjunction), and semantic sets. Did a lot of hand correction, it seems. Used $\alpha = 0.10$ for statistical p-values, instead of the usual $\alpha = 0.05$, which indicates to me that their results could not have been great. They were only able to show that the program under question had the most variance of the results, which makes sense given that they were of various verification difficulty levels. With $p = 0.07$, any tool predicted success, with no tools at all resulting in a 33% success rate. This high success rate without anything at all, combined with the problems surrounding added difficulty, indicates to me that these tests only go to show that verification on toy problems is easy regardless of what you use. Essentially, the study concludes that annotation assistance tools are useless, which sort of indicates that their study was flawed.

  Found that ESC/Java was so slow that users generally would just think hard about the problem and write down invariants this way rather than going incrementally. This is backed up by the fact that the pure Whodini group started to write more redundant assertions that would have been inferred by Whodini as they proceeded through the activities. This is the nail in the coffin for this study - the tools being studied are too slow to use, so in practice everyone is merely going manual. Additionally, Daikon has an unfair advantage: if it infers an incorrect invariant, ESC/Java simply says that it's not correct. The penality for incorrect invariants is thus very small with Daikon. The only real negative that users noted about Daikon is that the inferred invariants were generally textually large, which reflects its overly specific nature. It claims that the Daikon test suite was poor, but given how short the code was coupled with how long the tests took indicates to me that this is untrue.

For even more evidence that this study was too small and contrived to be of any real value, see the summary for [50].

8 Miscellaneous SE / DSE

- Koushik Sen. Scalable automated methods for dynamic program analysis. In *PhD Dissertation*, 2006

  Discusses how to analyze a number of different program features (such as lazy evaluation) using symbolic execution.

9 Representing Polyhedra

- Christoph Scholl, Stefan Disch, Florian Pigorsch, and Stefan Kupferschmid. Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 383–397, Berlin, Heidelberg, 2009. Springer-Verlag

  Uses an SMT solver to help find tautologies and contraditions in subconstraints of non-convex polyhedra to simplify constraints. It seems that such simplification only needs to be performed for the non-convex versions; as to why I do not understand as of this writing. The simplification seems to be in the same vein as what is discussed in [15], but in far more detail.

- Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. A new encoding and implementation of not necessarily closed convex polyhedra. In *UNIVERSITY OF SOUTHAMPTON*, page 161. Publications, 2003

  An efficient way to represent NNC polyhedra, using only a linear number of constraints (in one representation) and a constant number of generators of rays / lines (in a second representation that is maintained in parallel with the first representation).

- Doran K. Wilde. A library for doing polyhedral operations. Technical report, 1993

  The original master's thesis that introduced the Parma Polyhedral Library. Goes very in-depth with how polyhedra were represented, right down to the members of C structs. Overall a very good introduction to convex polyhedra and how to represent them in practice.

10 Application of Polyhedra to Programming Languages

- Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM

  Original paper that introduced convex polyhedra as an abstract domain for the discovery of linear constraints among integer program variables. Difficult but worthwhile read.

- Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. *CoRR*, abs/cs/0703077, 2007

  Discusses how to model operations involving floating-point numbers (specifically the IEEE-754 numbers we all know and love) in different numerical domains commonly used in abstract interpretation, including that of convex polyhedra. Employs a series of sound approximations in order to do this. The term given to this is linearization (as in, we are representing what were originally non-linear constraints in a linear domain). Goes over the reasons why floating point numbers, along with non-linear operations like multiplication and division, are difficult to represent with traditional abstract domains.

  Floating-point numbers are themselves approximations of real numbers, as are the operations that can be performed on them. These approximated operations are neither associative nor distributive like their precise counterparts, which (based on my understanding) means that monotonicity is lost if we were to represent this in a lattice. Related to this is a hint at a reason why convex polyhedra only support linear constraints: multiplication and division are not neccessarily closed operations. I.e. the product of two convex polyhedra is not necessarily itself a convex polyhedron.

- Liqian Chen, Antoine Min, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In Jens Palsberg and Zhendong Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 309–325. Springer, 2009

Introduces a new abstract domain that combines convex polyhedra, non-convex polyhedra, and it appears even powersets of both of these into a single unified domain: interval polyhedra. The domain is strictly more general than these and thus potentually more precise, at the cost of worse performance. They can better handle non-linear operations like multiplication and division, without resorting to the sort of lossy linearization techniques detailed in [37]. They can also natively handle constraints involving infinity (like true intervals) and certain forms of negation and disjunction. (Indicentally, negation and disjunction cannot be precisely handled by normal convex polyhedra, as these operations are not closed. The results of these operations are not necessarily convex polyhedra themselves.)

11 Invariants on Demand

- Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 474–488, Berlin, Heidelberg, 2006. Springer-Verlag

  While the paper applies convex polyhedra to programming languages, this is merely an application of their technique. They describe a way to lift a widening operator on some abstract domain to the powerset of that abstract domain. They also use pre-image computation to determine when widening should be used and when a more precise join (least upper bound in their paper) should be used. This pre-image computation is done on demand, in a manner similar to [35]. At the beginning, they will always choose to widen over join, until they are unable to prove an assertion. At this point, they use pre-image computation to figure out exactly at which widening point the amount of precision necessary to prove the assertion was lost, and they subsequently restart from this point, applying join instead. Since widening is typically much cheaper than join, this avoids the extra computing cost for getting precision that may not even be needed. This paper is relevant as a client application (see the summary for [35]), and also to convex polyhedra. They are able to infer a disjunctive invariant due to the usage of a powerset of convex polyhedra, without resorting to heuristics.

- K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 119–134, Berlin, Heidelberg, 2005. Springer-Verlag

The same sort of on-demand approach described in [29] is used, in that the analysis will a sort of backtracking when it is unable to prove an assertion. In this paper, they will instead opt for a more precise abstract domain in this case. For example, the analysis may switching from using normal intervals to convex polyhedra. In this way, they avoid deriving information that is unneccessarily precise for proving some given invariant, saving possibly significant computation time. (This is unfortunately an assertion on the part of the authors; they do not evaluate their approach in the same manner that [29] does.)

12 SAT Solvers

- Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960

  Original paper describing the DPLL procedure. Conjunctive normal form (CNF) is used because it's easier to reason about a normal form, and there is no exponential blowup possible as with disjunctive normal form. Has three rules for simplifying clauses:

  1. For eliminating one-literal clauses:
     (a) In a series of conjunctions, whenever $\ldots \wedge p \wedge \ldots \wedge \neg p \wedge \ldots$ is disovered, the whole clause can be replaced with `false`.
     (b) If p is a clause in CNF form, then all other clauses that contain p affirmatively can be removed, and all instances of $\neg p$ can be removed.
     (c) If $\neg p$ is a clause in CNF form, then all other clauses that contain $\neg p$ can be removed, and all instances of p can be removed.
  2. For all CNF clauses, if the parity of p is the same, then all clauses which contain p (or $\neg p$) can be deleted.
  3. Let the given CNF formula be put into the form $(A \vee p) \wedge (B \vee \neq p) \wedge R$, where A, B, and R are free of p. This can then be replaced with $(A \vee B) \wedge R$.

  Empty clauses become `true` in this system. By repeatedly applying the above simplifications, it is possible to refute any formula, although this algorithm is non-terminating in the case of satisfiability.

- Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962

  First implementation of DPLL [17]. Instead of using elimination rule #3 as-is, they split $(A \vee p) \wedge (B \vee \neq p) \wedge R$ into cases for $(A \wedge R)$ (when $p = 0$) and $(B \wedge R)$ (when $p = 1$). This was done to avoid the addition of many new clauses.

- Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM

  Advantage of CNF: if ay clause is unsat, the whole problem is unsat. Learning why the unsat clause is unsat can reveal a great deal of information. Unit clause: when a given clause is not yet satisfied, and all but one literals are assigned. Boolean constraint propagation uses unit clauses. The necessary value to make the clause hold is referred to as an *implication.* Implications are given integer ids based on the height of the stack. This way, if backtracking must occur to pop down to a stack depth of n, then any implications with ids $> n$ need to be removed (i.e. these are for decisions made after the point we backtracked to).

  Repeated applications of the unit clause rule are for a sort of forced decision - we know for certain that we cannot proceed unless certain variables have certain values. These sort of repeated applications are known as binary contraint propagation (BCP).

  The authors observe that for most SAT problems, 90% of execution time is spent in BCP. Efficient BCP implementation necessitates a way to quickly determine which variables are newly implied by the addition of a given assignment.

  The basic idea is to "watch" two literals in a clause. We only need to examine a clause if one of the literals has been assigned to 0. While this does not guarantee that we will be able to apply the unit clause rule (as with initially three unassigned literals and ony assigning one of them, but the one assigned was watched), it increases the likelihood. Additionally, in the case that the unit clause rule is applicable, then the unassigned watched literal is the new literal to assign.

  The authors hint that many previous evaluations have focused only on the number of decision points made, i.e. how many random choices were needed. They point out that this is a poor metric as it doesn't incorporate BCP at all; it is possible to have a case with few decisions that is BCP-heavy, as well as a case with many decisions that is light on BCP. Given that BCP is the dominating factor, if we only optimize for the number of decisions, then we can actually hurt performance.

  They introduce the *Variable State Independent Decaying Sum* (VSIDS) heuristic, which priorities BCP on clauses in a way that focuses on conflicts. There are five basic steps to this heuristic:

  1. Each literal has a counter initialized to 0
  2. When a clause is added to the database, the counter corresponding to each literal is incremented.
  3. At decision points, the literal with the highest counter is chosen

4. Ties are broken randomly by default
5. Periodically, all counters are divided by a constant (focusing on more recent conflicts)

The basic intuition behind this scheme is that difficult problems tend to introduce many conflict clauses, to the point where the conflict clauses contribute to the majority of the problem. These tend to drive the search of the problem, and so this heuristic tends to favor searching in a conflict-driven way.
NOTE TO SELF: There is a whole lot of "tends to" in here. Given the NPC nature of the problem, there isn't much choice but to resort to heuristics that can potentially fail.

Using this heuristic, generally smaller problems were completely unaffected in terms of performance relative to competing implementations. However, for many larger problems, Chaff saw between one to two orders of magnitude improvement in performance.

- Joao P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society

  Introduces GRASP. Key insight: the introduction of non-chronological backtracking. Seems to be the first paper that allows for the addition of clauses dynamically. Previous techniques appear to utilize learning, though not with this technique and with a more coarse-grained approach. Introduces the implication graph. For determining the conflicting clause, one needs to go backwards from a conflict node in the implication graph. If we find that the conflicting clause was from assignments at levels n and m, where $m \geq n$ (not including the current level), then we can set the current backtracking level to m. This is the non-chronological part; a chronological backtrack would still explore it's part of the state space completely before backtracking to m. In other words, we always backtrack to the highest available level (i.e. lowest in the tree).

- Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, July 1978

  Introduces Shostak's decision procedure for EUF without disjunction. With a union-find data structure, the algorithm runs in $O(n\mathtt{lg}(n))$. Computes the *congruence closure* by putting variables in equivalence classes. If it ever becomes the case where two variables x and y are in the same equivalence class, and there exists a literal in the formula of the form $x \neq y$, then the formula is unsatisfiable. However, if all classes stabilize and such a case is never encountered, then the formula is satisfiable.

To illustrate, consider the following formula:

$$x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1 \wedge \mathtt{F}(x_1) \neq \mathtt{F}(x_3)$$

The algorithm will first generate the following equivalence classes:

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{\mathtt{F}(x_1)\}\{\mathtt{F}(x_3)\}$$

The first two classes can be merged, since they both contain x_2:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{\mathtt{F}(x_1)\}\{\mathtt{F}(x_3)\}$$

Since now x_1 and x_3 are in the same class, $\mathtt{F}(x_1)$ and $\mathtt{F}(x_3)$ can be merged as well:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{\mathtt{F}(x_1), \mathtt{F}(x_3)\}$$

At this point, both $\mathtt{F}(x_1)$ and $\mathtt{F}(x_3)$ are in the same equivalence class. However, the original formula contains the disequality $\mathtt{F}(x_1) \neq \mathtt{F}(x_3)$. As such, the formula is unsatisfiable.

This method can be extended to handle disjunction via case-splitting or putting the initial formula into DNF (as opposed to CNF). With DNF, the process needs to be run on all formulas. Of course, either way introduces an exponential time complexity. Because this method does not learn from disjunctions, it is poorly suited for more realistic cases of SAT.

- Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. pages 470–482. Springer-Verlag, 1999

  Introduces Bryant's reduction for reducing equality logic with uninterpreted functions down to SAT. They deal with EUF (theory of equality + uninterpreted functions) with an `if-then-else` (ITE) operator. By exploiting ITE, they achieve a reduction that is potentially smaller than Ackermann's reduction [1] in terms of the number of variables introduced. This reduction is illustrated in a step-by-step fashion below:

$$x = y \implies g(f(x)) = g(f(y)) \quad (1)$$
$$\neg(x = y) \vee g(f(x)) = g(f(y)) \quad (2)$$
$$\neg(x = y) \vee g(vf_1) = g(ITE(y = x, vf_1, vf_2)) \quad (3)$$
$$\neg(x = y) \vee vg_1 = ITE(ITE(y = x, vf_1, vf_2) = vf_1, vg_1, vg_2) \quad (4)$$

  In addition, we must also add constraints involving the functions, which follow in a straightforward manner:

$$vf_1 \wedge ITE(y = x, vf_1, vf_2) \wedge ITE(ITE(y = x, vf_1, vf_2) = vf_1, vg_1, vg_2)$$

For validity, it must be the case that the above constraints apply the transformed formula, and for satisfiability the conjunction is instead appropriate.

Another example is illustrated below:

$$o_1 = i \land o_2 = g(o_i, i) \land o_3 = g(o_2, i) \quad (1)$$
$$o_1 = i \land o_2 = g_1 \land o_3 = ITE(o_1 = o_2 \land i = i, g_1, g_2) \quad (2)$$

...with the additional constraints:

$$g_1 \land ITE(o_1 = o_2 \land i = i, g_1, g_2)$$

Individual levels of function calls are stripped away from the inside out. Function calls with a given set of parameters are replaced with variables representing these calls. If the parameters are the same, then these auxilliary function call variables are the same.

- Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Logic*, 3(4):604–627, October 2002

  This introduces a sparse method of adding the sort of transitivity constraints seen in Ackermann's reduction [1] and Bryant's reduction [7]. With this method, only those constraints which are shown to be necessary are added, which is typically far fewer than those possible. The basic idea is to construct a graph where vertices are atoms (irrespective of polarity) and edges represent both $=$ and $\neq$ (they are conflated). With this representation, variable relationships which may be influenced by transitivity constraints become apparent through the precense of cycles. The nature of such cycles indicates which transitivity constraints must be added. For example, consider a simple cycle x_1, x_2, x_3, x_4, x_1. Assuming there are no other edges, an edge must be added either between x_1 and x_3 or between x_2 and x_4. This will form a triangle (a particular representation is chosen which will make this true, and this representation can be derived in polynomial time). This process must be completed until all cycles in the graph form triangles (i.e. a *chordal* version). For each triangle, constraints must be added such that each pair of edges implies the third edge (i.e. equality between any two variables in the triangle implies equality with the third variable). This adds three implications per triangle.

  This results in fewer overall clauses introduced, while still avoiding the introduction of additional variables. Additionally, according to [32], this can be used to generate smaller equality graphs, and to reduce the size of the state space (i.e. in the spirit of [45]).

- Amir Pnueli, Yoav Rodeh, Ofer Strichmann, and Michael Siegel. The small model property: how small can it be? *Inf. Comput.*, 178(1):279–293,

October 2002

The theory of equality deals with some intentionally unspecified domain. Concretely, as with a particular satisfying assignment, variable values are members of some given domain. This paper deals with encoding this sort of information in pure SAT in a way that is unspecific to the given domain. Given n variables from an arbitrary domain D, an upper bound on this mapping is n^n. In other words, every variable can take on up to n values from D, and with n^n members of D (i.e. subsets of D which do not overlap with respect to n) we can clearly allow for each variable to take on completely different vaues. With an example, consider the formula $x_1 = x_2 \wedge x_2 = x_3$, which contains 3 variables. If x is over the natural numbers (i.e. $\mathbb{N}$), then we can partition $\mathbb{N}$ like so: $[0, 1, 2], [0, 1, 2], [0, 1, 2]$.

A lower bound on this can be achieved. From the same example above, we can also partition like so: $[0, 1, 2], [0, 1], [0]$. This still permits each variable to be different. Using this method, the mapping is $n!$, which is clearly smaller than n^n. While this still may appear large, even n^n can be encoded compactly: each variable can be represented with $\mathtt{lg}(n)$ bits, as with typical binary encoding. As such, even for the n^n mapping, for n variables we introduce $n\mathtt{lg}(n)$ binary variables, which is only polynomial with respect to the original formula.

- Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 358–372, Berlin, Heidelberg, 2007. Springer-Verlag

Real systems generally work at the word level - data acts as real values within a word, and as bits beyond a word. Bit-blasting (simply converting everything down to SAT) loses high-level structure, and performs poorly in the precense of disjunctions [23]. At the other extreme, reasoning over arbitrary precision types is also highly innaccurate.

Argues for a counterexample-guided approach. Given some input formula ϕ, they construct an underapproximation $\underline{\phi}$, in which each variable is modeled using only a subset of the bits specified in the source program. This construction is performed in such a way that if $\underline{\phi}$ is satisfiable, then ϕ is satisfiable. However, they are not equisatisfiable: if $\underline{\phi}$ is unsatisfiable, ϕ may still be satisfiable. In this case, an unsatisfiable core is emitted, which is used to generate an overapproximation $\overline{\phi}$. This overapproximation is based on only a subset of the clauses in $\underline{\phi}$. If $\overline{\phi}$ is unsatisfiable, then ϕ is also unsatisfiable. If not, then $\underline{\phi}$ is refined by increasing, for at least one bit-vector variable, the number of bits modeled. Given that the domain is finite, termination is guaranteed. In the worst case, the original query ϕ will be run directly.

While this process may seem complex, it is very efficient when either the satisfiability or unsatisfiability of the input query ϕ can be reasoned about using only a small subset of the bits needed to model ϕ directly.

- Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979

  Original paper discussing the Nelson-Oppen technique for combining solvers for theories with certain restrictions in a sound and complete manner. It is a requirement that each theory include equality, as it is used as the means of communication between theories. The basic idea is to first *purify* the input formula so that each atom involves only a single theory. This way, theories can be reasoned about separately. Next, using a particular theory T_i across the formula, equality constraints within T_i are discovered. If the formula is found to be unsatisfiable, then the whole formula under all theories is unsatisfiable. If not, then the additional equality constraints which have been learned are passed onto the other theories, and the process repeated. Once we reach a point where no more equalities can be propagated, the whole formula must be satisfiable, and any assignments made are returned.

  This process only works if the input theories are *convex*, which intuitively means that for a series of disjunctions, if the series is implied then it must be the case that at least one of the disjunctions are implied. This is not always true, as with the non-convex theory of integer arithmetic, as illustrated through the formula below:

  $$x_1 = 1 \land x_2 = 2 \land 1 \leq x_3 \land x_3 \leq 2 \implies (x_3 = 1 \lor x_3 = 2) \qquad (1)$$

  The above formula implies neither $x_3 = 1$ nor $x_3 = 2$ individually, but rather $x_3 = 1 \lor x_3 = 2$ as a whole. To handle such non-convex theories, case-splitting is required at such points. Using again the above example, a case for $x_3 = 1$ would be propagated, and a second case where $x_3 = 2$ would be propagated. If either case is satisfiable, then the formula as a whole is satisfiable. However, if all are unsatisfiable (as with this example), then the formula as a whole is unsatisfiable.

  Only equalities ever need to be propagated between theories. If any theory discoveres an inequality that conflicts with some propagated equality, then it will make the algorithm terminate with an unsatisfiable result.

- Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, May 2008

  A more modern way of combining theories than pure Nelson-Oppen [40], used by Z3 [19]. This is based on the observation that for most queries, the number of inconsistencies within a theory is much bigger than the number of inconsistencies between theories. For a given theory, all equalities that hold in the model are propagated, even if they are not implied by the model. While these can be invalidated later forcing backtracking, this is rare. As such, substantial performance benefits are seen.

- Dejan Jovanović and Clark Barrett. Being careful about theory combination. *Form. Methods Syst. Des.*, 42(1):67–90, February 2013

  The basic idea is to add to the Nelson-Oppen [40] algorithm an *equality propagator* and a *care function*. The equality propagator determines for a particular theory which equalities and disequalities are implied over interface variables, where interface variables are used to communicate between theories. The care function determines which of the equalities and disequalities are necessary for maintaining the satisfiability of a given formula.

- Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, JELIA '02, pages 308–319, London, UK, UK, 2002. Springer-Verlag

  Most theory-specific work deals with non-disjunctive queries, relying on other techniques to handle disjunctions. They develop a calculus that models DPLL [16, 17], and show that essentially all heuritsics surrounding DPLL naturally fit into this calculus. They then introduce DPLL($\mathbb{T}$), which applies DPLL to a specific theory. The calculus for DPLL($\mathbb{T}$) is the same as for DPLL, except that operations describing whether or not certain literals are implied by the current set of assignments are all theory-specific. The empty theory in this context refers to DPLL (i.e. DPLL($\mathbb{T}$) instantiated without a theory and with Σ being a set of propositional variables). The interesting part about this formulation is that it makes explicit which heuristics are possible given a particular theory: certain operations for heuristics become intractable or even undecidable depending on the theory at hand. The empty clause refers to a conflict.

- Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of the 18th International Conference on Automated Deduction*, CADE-18, pages 195–210, London, UK, UK, 2002. Springer-Verlag

  Discusses a basic naive lazy encoding, wherein original formulas are converted into propositional abstractions. To illustrate, the first formula below has been abstracted into propositional logic in the separate formula:

$$x_1 = x_2 \wedge (x_2 \neq x_3 \vee x_3 = x_4) \tag{1}$$

$$e_1 \wedge (e_2 \vee e_3) \tag{2}$$

If the abstraction is unsatisfiable, then the original formula is unsatisfiable. If the abstraction is satisfiable, then it is mapped back to the original formula, where we determine if the conjunction of the returned values is satisfiable in the logic. Going along with the example above, this means checking the satisfiability of the formula below, assuming that the SAT solver used the assignment `true` for all variables:

$$x_1 = x_2 \land x_2 \neq x_3 \land x_3 = x_4$$

In this case, the above statement is satisfiable. If this were not satisfiable, then we could generate a SAT lemma stating that this assignment cannot be true (i.e. conjoin the lemma $\neg e_1 \lor \neg e_2 \lor \neg e_3$). This added lemma is referred to as a blocking clause. We can also be more intelligent about forming this clause to make it more minimal, but such is an optimization.

- Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006

  Discusses theory propagation. Proposes an initial calculus that is essentially the same as in [59]. On top of this they add components to their calculus for non-chronological backtracking [56], conflict-driven learning, and restarts.

  Divides the handling of different theories into *eager* and *lazy* approaches. Eager approaches perform a direct translation to SAT. While these are straightforward, one needs to prove that the translation preserves the original semantics. Additionally, these tend to see poor performance. The basic idea behind lazy techniques is to develop theory-specific direct solvers which do not handle disjunction, and then solve the original query in DNF form. Of course, due to the translation into DNF, exponential blowup is both possible and common. Mentions the technique in [2], and notes some inefficiencies.

  They endorse DPLL($\mathbb{T}$), saying that it combines the advantages of both the eager and lazy approaches. With theory propagation, whenever a new assignment is made, a theory is asked to provide all assignments which are consequences of the given assignment. This may trigger additional rounds of theory propagation, in much the same manner as binary constraint propagation (BCP). After performing theory propagation, BCP is performed, which may trigger further rounds of theory propagation. If a conflicting clause is ever discovered, then non-chronological backtracking is performed (i.e. backjumping). As usual, we can learn binary lemmas via backjumping.

  At least at the time, this technique was a substantial improvement over all contenders, winning first place in an SMT competition in all categories where they had a solver for the category (four in total).

References

[1] W. Ackermann. *Solvable cases of the decision problem.* Studies in logic and the foundations of mathematics. North-Holland Pub. Co., 1954.

[2] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of the 18th International Conference on Automated Deduction*, CADE-18, pages 195–210, London, UK, UK, 2002. Springer-Verlag.

[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. A new encoding and implementation of not necessarily closed convex polyhedra. In *UNIVERSITY OF SOUTHAMPTON*, page 161. Publications, 2003.

[4] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20. Springer, 2004.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[6] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations: Research articles. *Softw. Test. Verif. Reliab.*, 16(2):97–121, June 2006.

[7] Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. pages 470–482. Springer-Verlag, 1999.

[8] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 358–372, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Comput. Logic*, 3(4):604–627, October 2002.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[12] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM.

[13] Liqian Chen, Antoine Min, Ji Wang, and Patrick Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In Jens Palsberg and Zhendong Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 309–325. Springer, 2009.

[14] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[15] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[16] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[17] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[18] Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, May 2008.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Voler Diekert. Makanin's algorithm. In *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Pree, 2002.

[21] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[22] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2006.

[23] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[25] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[27] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[28] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM.

[29] Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 474–488, Berlin, Heidelberg, 2006. Springer-Verlag.

[30] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *NASA Formal Methods*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 121–125, 2009.

[31] Dejan Jovanović and Clark Barrett. Being careful about theory combination. *Form. Methods Syst. Des.*, 42(1):67–90, February 2013.

[32] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[33] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. Flopsy: search-based floating point constraint solving for symbolic execution. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 142–157, Berlin, Heidelberg, 2010. Springer-Verlag.

[34] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[35] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 119–134, Berlin, Heidelberg, 2005. Springer-Verlag.

[36] Claude Michel. Exact projection functions for floating point number constraints (pdf). In *AMAI*, 2002.

[37] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. *CoRR*, abs/cs/0703077, 2007.

[38] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[39] Patricia Mouy, Bruno Marre, Nicky Willams, and Pascale Le Gall. Generation of all-paths unit test with function calls. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 32–41, Washington, DC, USA, 2008. IEEE Computer Society.

[40] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979.

[41] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006.

[42] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. *Electr. Notes Theor. Comput. Sci.*, 55(2):255–276, 2001.

[43] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 229–239, New York, NY, USA, 2002. ACM.

[44] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: an empirical evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20, November 2002.

[45] Amir Pnueli, Yoav Rodeh, Ofer Strichmann, and Michael Siegel. The small model property: how small can it be? *Inf. Comput.*, 178(1):279–293, October 2002.

[46] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[47] J. Richard Bchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, 34(4):337–342, 1988.

[48] Raul Santelices and Mary Jean Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 195–206, New York, NY, USA, 2010. ACM.

[49] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 225–236, New York, NY, USA, 2009. ACM.

[50] Todd W. Schiller and Michael D. Ernst. Reducing the barriers to writing verified specifications. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 95–112, New York, NY, USA, 2012. ACM.

[51] Christoph Scholl, Stefan Disch, Florian Pigorsch, and Stefan Kupferschmid. Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 383–397, Berlin, Heidelberg, 2009. Springer-Verlag.

[52] Koushik Sen. Scalable automated methods for dynamic program analysis. In *PhD Dissertation*, 2006.

[53] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, July 1978.

[54] J.H. Siddiqui and S. Khurshid. Parsym: Parallel symbolic execution. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 1, pages V1–405 –V1–409, oct. 2010.

[55] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 523–536, New York, NY, USA, 2012. ACM.

[56] Joao P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[57] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 188–198, New York, NY, USA, 2012. ACM.

[58] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[59] Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In *Proceedings of the European Conference on Logics in Artificial Intelligence*, JELIA '02, pages 308–319, London, UK, UK, 2002. Springer-Verlag.

[60] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

[61] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200, New York, NY, USA, 2011. ACM.

[62] Doran K. Wilde. A library for doing polyhedral operations. Technical report, 1993.

[63] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[64] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution.

In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 243–257, Washington, DC, USA, 2006. IEEE Computer Society.