Software Engineering and Automated Deduction

Willem Visser
Stellenbosch University
Stellenbosch, South Africa
visserw@sun.ac.za

Nikolaj Bjørner
Microsoft Research
Redmond, WA, USA
nbjorner@microsoft.com

Natarajan Shankar
SRI Computer Science Lab
Menlo Park, CA, USA
shankar@csl.sri.com

ABSTRACT

Software poses a range of engineering challenges. How do we capture the expected behavior of the software? How can we check if such behavioral descriptions are consistent and valid? How do we generate test instances that explore and examine different parts of the software. We focus on the underlying technology by which a number of these problems can be reduced to a logical form and answered using automated deduction. In the first part we briefly summarize the use of automated deduction within software engineering. Then we consider some of the current and future trends in software engineering and the type of advances it may require from automated deduction. We observe that in the past software engineering problems were solved by merely leveraging advances in automated deduction, especially in SAT and SMT solving, whereas we are now entering a phase where advances in automated deduction are also driven by software engineering requirements.

1. INTRODUCTION

Can software engineering really be that hard? Software is composed of a number of lines of code. Each line has a well-specified effect on program execution. Modern computers can execute these lines of code rapidly and deliver quick feedback. Yet, software projects continue to fail in spectacular ways even as massive resources are devoted to testing and debugging software. An empirical approach to software development has its strengths, but for many critical applications, we need analytical tools and techniques that are rooted in sound theory but are practical enough to predict the behavior, both correct and anomalous, of something as complex as software.

Logic is integral to thinking about software, and indeed to computing as a whole [57]. Like programming, logic is characterized by the use of a formal language. While a programming language is read as a recipe for a computation, logical formulas describe states of affairs. Logic can thus be used to characterize the possible inputs to a program, the range of values assigned to variables at a program point, the possible execution steps of a program, the effects of these execution steps on the program state, or the properties of execution traces of a program. The suitability of logic for computing stems from the ability of logic to abstractly characterize states of affairs coupled with the technology to reason systematically with these abstract descriptions. Logic can then be used as an interface layer to mediate between problems and tools, so that many different kinds of problems can be mapped to queries of logical validity or satisfiability, and many different tools can be used to answer these queries.

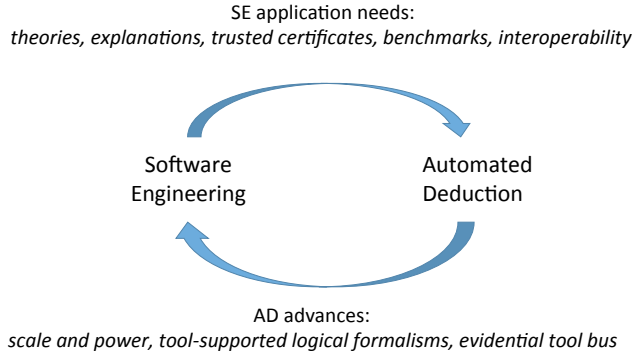

Figure 1: A virtuous cycle of influences between software engineering needs and advances in automated deduction.

The connection between logic and software was established early on. Before there was even a concept of a programming language, logical formalisms for lambda calculus, automata, and equational reasoning provided a foundation for computability. Already in the late 1940s, Turing [79] and von Neumann and Goldstine [82] worked out examples of assertional verification of imperative programs. McCarthy [77] introduced the idea of recursion–induction as a way of reasoning about functional programs over recursively defined datatypes. He also showed how assertional reasoning about flowchart programs could be reduced to recursion–induction. Floyd [49] outlined an assertional method for reasoning about the total correctness of flowchart programs, and Hoare [58] presented an axiomatic framework for assertional proofs of programs. These early ideas have been extended in a number of directions such as algebraic, denotational, and operational semantics, inductive theorem proving, predicate transformer semantics, temporal logics, model checking, test generation, and program synthesis. An early survey paper by Elspas, Levitt, Waldinger, and Waksman [45] describes some of the deep links between logic and program correctness, and a more recent survey by Shankar [101] describes a number of advances in the use of automated deduction for verification.

The focus of this paper is on the use of logic, specifically automated deduction, to solve problems in software engineering. In addition, we provide a roadmap for the future of deduction-based software engineering. Until recently, one could argue that software

ICSE 2014 Hyderabad, India

engineering techniques simply benefitted from the exceptional advances in automated deduction, whereas we are now in a phase where a number of advances in automated deduction are actually driven by the needs and constraints of software engineering. Figure 1 summarizes this view as a virtuous cycle including the main themes covered in this paper.

1.1 The Value Proposition for Software Engineering

Many software engineering tasks depend critically on the ability to solve logic problems. We enumerate a few examples:

Semantics: What is the semantics of signed twos-complement representation of integers? An N-bit word or bit-vector can be represented as a function from the subrange interval $[0,N)$ to the Booleans $\{0,1\}$. Given an N-bit word A, let $|A|$ represent the unsigned magnitude of A, namely $A[0]+A[1]\times 2+\ldots+A[N-1]\times 2^{N-1}$. The signed magnitude $||A||$ is $|A|$, if $|A|<2^{N-1}$, and is $|A|-2^N$, otherwise. The same approach can be used to give the semantics of floating point arithmetic, machine language instructions, and for programming languages in general.

Specifications: Write a program that given a bit-vector returns the next highest bit-vector (according to the unsigned semantics) with the same number of 1-bits. Specifications succinctly and precisely characterize the range of inputs and the expected output for each input. Specifications can be given for anything ranging from a small subroutine to a library or a large application. In the case of reactive programs, the specification might capture a temporal pattern of interaction between a module and its environment. Deductive methods can be used to check specifications for various properties including consistency and completeness.

Correctness: Does a given program meet its specification? For example, does a binary search procedure find the index of a given element in an array, when it exists? Automated deduction techniques can be used to show termination, i.e., that the program halts on all well-formed inputs, and partial correctness, i.e., that the result when the program halts conforms to the specification. In some cases, the correctness is demonstrated by means of a refinement proof between the specification and the program.

Deductive Synthesis: Is there a procedure that can construct a program meeting a given specification? Techniques such as resolution and inductive theorem proving can be used to systematically synthesize algorithmic solutions to given problems. Decision procedures can also be used to fill holes in algorithmic templates, to synthesize reactive programs for temporal specifications, and to construct environment models.

Static analysis: Can we derive properties of variables and control states from the program? Deduction is used to construct abstract domains, *transfer functions* that approximate program behavior on the abstract domains, and for extended type checking and assertion checking.

Test generation: Is there a valid input that drives a program along a specified path or to a specific state? There are several variants on this basic procedure, but the question of finding a test input is essentially a logical one. A classic example is the generation of test cases to reach program statements. An approach based on symbolic execution was introduced in the mid-seventies by Boyer, Elspas, and Levitt [15], Clarke [31] and King [65], to find a model for a conjunction of constraints over the input variables (i.e., assignments to the input variables that will reach a program location). Symbolic execution never really got traction in software engineering until satisfiability checkers became more efficient, about 20 years later (mid-nineties).

Note also that software engineering problems are typically highly structured, whereas automated deduction tools have historically been more general purpose. Exploiting the structure of the problems within the automated deduction tools allow more efficient analysis and is a major research trend.

Value Propositions	Newer Trends
Semantics	Model based development
Specifications	Trusted Security
Correctness	High-integrity Systems
Deductive Synthesis	Inductive Deductive Synthesis
Static analysis	Education
Test-case generation	Biology

Table 1: Established Value Propositions and Newer Trends

1.2 Software Engineering Trends

In Section 4.4 we highlight a number of trends in software engineering research that will critically depend on advances in automated deduction for their success. In Table 1 we summarize the established value propositions just mentioned together with a set of newer trends that are identified in Section 4.4. Note that some of the trends are not new and have been around for decades, but we believe there is a resurgence in these areas (synthesis and education being prime examples).

1.3 Paper Layout

The first part of the paper contains a description of automated deduction tools and how they address software engineering issues. The second part of the paper will highlight some areas we believe where the future research focus will be for the use of automated deduction in software engineering.

2. AUTOMATED DEDUCTION TOOLS

The main question addressed by (automated) deduction can be summarized as: Given a logical formula φ, establish that it is *valid*. Dually, given a logical formula φ establish that it is *satisfiable*, e.g., establish that there is a model M that satisfies φ. We write $M \models \varphi$ to say that M satisfies φ. Note that φ is valid if and only if $\neg\varphi$ is unsatisfiable. In model checking, one is given a model M and formula φ and the question is "only" to check if $M \models \varphi$ holds. Here the model is (a Kripke structure) extracted from a program and it can be very large or even infinite. On the other hand, deriving program invariants and other properties is an instance of deriving an abstraction from a model M. To summarize, we have the following themes:

$$\begin{array}{rl} ? \models \varphi & \text{deduction} \\ M \models \varphi & \text{model checking} \\ M \models ? & \text{abstraction} \end{array}$$

One can also recast problem domains in terms of others. Model checking is deduction in disguise: associate with a model M the characteristic formula $\chi(M)$, and check validity of $\chi(M) \rightarrow \varphi$ [20, 90]. Logics and encodings are key to presenting these problems in an effective way: $\chi(M)$ can be directly encoded in linear time temporal logic or higher-order logics [99]. If $\chi(M)$ is given as a least fixed-point $\mu X.\chi(M,X)$, then it suffices to check satisfiability of $\forall\vec{x}.(\chi(M,Reach)(\vec{x}) \rightarrow Reach(\vec{x})) \wedge (Reach(\vec{x}) \rightarrow \varphi)$ where $\vec{x}$ are the free variables in φ and *Reach* is a fresh predicate. Reformulations help encode problems according to the domains handled

by automated deduction tools, but solving such problems may require specialized automated deduction strategies and decision procedures.

2.1 Logics and Automated Deduction

Let us first summarize some main trends in automated deduction. *Automated first-order theorem proving* tools, commonly known as ATP tools, prove properties of theorems expressed in (pure classical) first-order logic. Classical first-order logic has been shown to be very versatile even though it is based on a limited set of basic notions. It has its roots in Aristotelean logic and it has been used to encode problems from domains ranging from abstract algebra to linguistic analysis. *Propositional logic* (SAT) comprises what one can reasonably characterize as the smallest sensible subset of first-order logic. From the point of view of first-order logic theorem proving, it is a trivial case, but algorithms and tools that have proved effective for solving propositional formulas have become a highly active area in itself in the past two decades. *Satisfiability Modulo Theories* (SMT) has been used to characterize an area of theorem proving that integrates built-in support for domains that are commonly found in programs and specialized algorithms to solve these formulas. This contrasts pure first-order theorem proving that has no built-in support for domains, such as integers, reals, or algebraic data-types. Specialized support for such theories is critical in modeling and analyzing software systems. The area of automated deduction includes also substantial developments in *non-classical* and *higher-order* logics.

Thus we see automated deduction aligned around a set of broad categories defined by the logics they handle. The lines along these categories are naturally blurred as developments in one area influence other areas.

2.1.1 First-order Theorem Proving

Algorithms for automated deduction were developed well before software engineering became a field. First-order logic is rooted in logic and foundations for mathematics. Recall that formulas over first-order logic are built from constants, bound variables, compound terms and formulas are formed by combining predicates using logical connectives. Taking a minimalistic approach, formulas are

$$\begin{array}{rcl} \mathit{term} & ::= & \mathit{var} \mid f(\mathit{term}^*) \mid \mathit{const} \\ \varphi & ::= & P(\mathit{term}^*) \mid \mathit{term} = \mathit{term} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \forall \mathit{var}.\varphi \end{array}$$

Logical conjunction $\varphi \wedge \varphi'$ and negation $\neg\varphi$ can be used to define disjunction $\varphi \vee \varphi' := \neg(\neg\varphi \wedge \neg\varphi')$ and other connectives. First-order satisfiability is concerned whether there is an interpretation (model) that satisfies the given formula. A first-order model comprises of a domain (a set A), for each free constant a value in A, and for each function a graph over A. Formulas are evaluated over first-order models, and formulas that evaluate to true are *satisfied*.

Herbrand's theorem from 1930, laid a foundation for effective methods for first-order validity. A sentence is a formula with no free variables, and it is valid if its negation is unsatisfiable. A first-order sentence such as $\exists x.(p(x) \implies (\forall y.p(y)))$ is valid if its negation $\forall x.p(x) \wedge \neg(\forall y.p(y))$ is unsatisfiable. The latter formula can be placed in prenex form by moving all the quantifiers outwards to get $\forall x.\exists y.p(x) \wedge \neg p(y)$. This formula can be Skolemized as $\forall x.p(x) \wedge \neg p(f(x))$, where f is a freshly chosen function symbol. The latter formula is unsatisfiable if there is some Herbrand expansion, namely a disjunction of instances of the formula, that is propositionally unsatisfiable. In this case, the expansion $(p(c) \wedge \neg p(f(c))) \vee (p(f(c)) \wedge \neg p(f(f(c))))$ is propositionally unsatisfiable. Herbrand's theorem thus reduces first-order unsatisfiability to Boolean unsatisfiability, and methods like resolution took advantage of propositional reasoning to systematically find suitable Herbrand expansions through proof search. Our example above succumbs easily to resolution: The Skolemized form $\forall x.p(x) \wedge \neg p(f(x))$, yields two implicitly universally quantified clauses $p(x)$ and $\neg p(f(y))$, and with the unifier $\{x \leftarrow f(y)\}$, these clauses resolve to yield a refutation. Tools for first-order theorem proving mainly use the TPTP [107] (Thousand of Problems for Theorem Provers) interchange format. The example from above can be written in TPTP as:

```
fof(sample_first_order, conjecture,
    ( ? [X] : ( p(X) => ! [Y] : p(Y) ) )).
```

The area of automated first-order deduction subsequently flourished with constructive methods for proof search [103, 91]: Gentzen in 1934 developed sequent calculi; twenty years later, Beth introduced semantic tableaux methods; in 1960 Davis and Putnam, thanks to a grant by the NSA, introduced ordered resolution for proving theorems in first-order logic and when Davis later implemented the procedure with Logemann and Loveland, identified the DLL procedure that is at the heart of modern SAT solvers. A major breakthrough in first-order automated deduction came in 1965 when J. Robinson developed first-order resolution, unification, and subsumption (in the same paper [92]). This spurred a frenzy of tool developments and gave rise to the use of theorem proving in artificial intelligence and the Prolog systems. Today's main tools, E [97], Spass [114], Vampire [67], for first-order theorem proving are based on refinements of Robinson's resolution method.

2.1.2 Propositional Logic Theorem Proving

Formulas in propositional logic are first-order formulas without any terms or quantifiers, thus SAT formulas can be built using just propositional variables P, conjunction and negation:

$$\varphi \quad ::= \quad P \mid \varphi \wedge \varphi \mid \neg\varphi$$

The satisfiability problem is also simpler to represent: a model for a propositional formula is an assignment of the propositional variables to truth values 0 or 1, such that the formula evaluates to 1 under this assignment. The DIMACS format is the most popular way to interface with SAT solvers. It represents formulas as a sequence of *clauses*. Each clause is a disjunction of atomic propositions or their negations (called literals). Suppose we wish to prove that implication is transitive. As a formula, we would write $(P_1 \Rightarrow P_2) \Rightarrow ((P_2 \Rightarrow P_3) \Rightarrow (P_1 \Rightarrow P_3))$ and check if this formula is a tautology. Dually, the negation of the formula is unsatisfiable. We can write the negation as a set of clauses: $(\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_3) \wedge P_1 \wedge \neg P_3$ which in the DIMACS format is written, with `p` for "problem", as follows:

```
c This is a CNF example with 3 variables
c and 4 clauses. Each clause ends with 0.
p cnf 3 4
-1 2 0
-2 3 0
1 0
-3 0
```

The current trend in SAT solvers is around substantial refinements of the DLL procedure. A renaissance was initiated in the early 1990's by a community effort around benchmarking and evaluations. Some of the recent highlights in SAT solving include clause indexing introduced in the SATO tool. A groundbreaking development in GRASP was to extend the basic DLL scheme with

Year	Advance	Solver
1960	Davis-Putnam procedure	[38]
1962	Davis-Logemann-Loveland	[37]
1984	Binary Decision Diagrams	[19]
1992	DIMACS SAT challenge	[21]
1994	Clause indexing	SATO
1997	Conflict clause learning	GRASP
1998	Search Restarts	[52]
2001	2-watch literal, VSIDS	zChaff
2005	Preprocessing techniques	SatELite
2007	Phase caching	RSAT
2008	Cache optimized indexing	MiniSAT
2009	In-processing, clause management	Glucose
2010	Blocked clause elimination	Lingeling

Table 2: Progress in SAT solving

clause learning [76]. In a nutshell it gives SAT solvers the same power as resolution, while retaining the better space properties of DLL. It spurred a *decade of SAT triumphs* that we summarize in Table 2. Following GRASP and zChaff, several SAT solving tools have become available and are used as either stand-alone tools for solving SAT formulas or integrated in model checkers. The main tools are Lingeling [11], MiniSAT [44], and many tools built around MiniSAT, e.g., Glucose that applies sophisticated garbage collection [1], and several reference[1] and proprietary SAT solvers. The state of SAT solving until 2009 is collected in [12] and is updated in a forthcoming volume on SAT by Knuth.

2.1.3 Satisfiability Modulo Theories

The theory of arithmetic has been a central topic in logic ever since symbolic logic took shape. Solving logical formulas with arithmetical symbols is an instance of Satisfiability Modulo Theories (SMT). SMT formulas extend first-order logic with theories, such as the theory of arithmetic. Yet most of the motivation for current SMT solvers originate from using automated logic engines for program analysis. These include the solver for the Stanford Pascal Verifier [74], NQTHM [16], EHDM [93], and PVS [86], and more recently Barcelogic [13], Boolector [17], CVC [6], MathSAT [18], STP [50], Yices [43], Z3 [40].

SMT formulas are first-order formulas, except that some function and predicate symbols may have pre-defined interpretations. For example, the formula:

$$y = x+2 \;\Rightarrow\; f(\mathit{select}(\mathit{store}(a,x,3),y-2)) = f(y-x+1)$$

uses arithmetical functions $+$, $-$ besides constants $1,2,3$. It uses the functions *select* and *store* where $\mathit{store}(a,i,v)$ is characterized by $\mathit{select}(\mathit{store}(a,i,v),j) = \textbf{if } i = j \textbf{ then } v \textbf{ else } \mathit{select}(a,j)$. The function f is not interpreted by any built-in theory. The SMT-LIB2[2] interchange format is used by current SMT solvers and our example formula can be given to SMT solvers as:

```
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun a () (Array Int Int))
(declare-fun f (Int) Int)
(assert (= y (+ x 2)))
(assert (not
          (= (f (select (store a x 3) (- y 2)))
             (f (+ (- y x) 1)))))
(check-sat)
```

Note the following: The sort of integers `Int` is built in, the built-in sort constructor for arrays, `Array`, takes two sorts for the domain and range, and a unary uninterpreted function `f` is declared to take one `Int` and produce an `Int`. The original formula is negated and split into two assertions. The left side of the implication is asserted separately, and the negation of the right side of the implication is also asserted. Finally, we check *satisfiability* of the assertions. The conjunction of assertions are *unsatisfiable*, in other words, the negation of the conjunction is *valid*.

Many applications rely on solving logical formulas using arithmetic, and this theory remains an important theory in SMT solving. There are many sub-categories of arithmetical formulas that are handled by modern SMT solvers: linear (additive, where multiplication between variables is disallowed) arithmetic over integers and real numbers, polynomial arithmetic over reals, linear arithmetic over two variables (with unit coefficients) per inequality. But there are many other theories that are of relevance to software engineering: the theory of arrays is used when modeling memory heaps, the theory of bit-vectors is used for representing machine integers, algebraic data-types are natural when analyzing functional programs and specifications with Herbrand terms, and more recently SMT solvers have started supporting floating point numbers with IEEE semantics. SMT solving technologies leverage several techniques developed in the context of first-order theorem proving and SAT solvers. The over-arching focus is on how to efficiently integrate specialized solvers for theories, such as arithmetic, that solve efficiently satisfiability problems for Boolean combinations of logical constraints.

SMT solvers are a good fit for many software engineering applications due to the support for domains that are directly relevant to software development and analysis. We summarized also first-order automated deduction and SAT solvers. Both first-order theorem proving and SAT solving are integral components in SMT solving, and applications that can be described in propositional logic or in pure first-order logic can be solved using dedicated SAT solvers or ATP systems.

2.2 Deduction with Interaction

Automation is critical to many of the applications of logic in computing mentioned above. Formulas arising from hardware and software analysis and planning and scheduling can be large. Software and hardware verification can generate proof obligations that are in a where there is no complete inference procedure. Even when the inference procedure is complete, i.e., it is capable of proving any valid proof obligation, the automation might not be be efficient enough to handle large formulas or search spaces. An automated attempt to verify a proof obligation could fail due to the incompleteness or inefficiency of the inference procedure, or because the proof obligation was not valid. In situations where the inference procedure can fail, it becomes important to interact with the system to locate the root cause of the failure. In either case, interaction is needed to find the source of the problem and repair it.

Automation can also fail for mathematically challenging proofs that need clever inductions or careful decomposition into lemmas. Interaction is critical for nudging the proof search along a productive direction by allowing the user to suggest induction schemes, lemmas, case splits, and quantifier instantiations. Without such guidance, an automated strategy can spend a lot of time in fruitless paths in the search space by employing irrelevant lemmas, definition expansions, and quantifier instantiations. Interaction is also

[1] `http://www-cs-faculty.stanford.edu/~knuth/programs/sat13.w`

[2] http://www.smtlib.org

helpful as a way of composing automated techniques and exploring proof strategies for the purpose of mechanization. Most interactive provers support user-defined proof strategies that are typically developed using the insights gained from interactive proofs.

A productive deductive environment combines automation with interaction augmented with user-defined proof strategies. Fully automatic inference procedures are not that helpful in providing feedback when they fail, and lack the controls for fine-tuning the proof search. On the other hand, interaction without automation can quickly turn tedious. Even when it is possible to define proof strategies, these are not as efficient as bespoke inference procedures such as SAT solvers, SMT solvers, and rewriting engines. Automation is therefore useful in focusing user attention on the mathematically interesting aspects of a proof. Proof environments such as ACL2 [64], Coq [8], HOL [54], Isabelle [84], Nuprl [33], and PVS [86] explore different sweet spots in the space of automation, interactivity, and customization. PVS, for example, has a specification language based on higher-order logic enhanced with predicate subtypes, recursive datatypes, parametric theories, and theory interpretations. Many of these features make even the typechecking undecidable, but the undecidability only manifests itself through the generation of proof obligations. Simplification and rewriting supported by decision procedures for combinations of theories makes it possible to discharge these proof obligations with a significant degree of automation. ACL2 uses a logic based on an applicative subset of Common Lisp so that the functions defined in the ACL2 logic are directly executable as Lisp programs. Other interactive theorem provers feature logics that have executable fragments and code can be generated for expressions in these fragments.

Interactive deductive environments or proof assistants have been used to formalize large libraries of mathematical knowledge, and significant accomplishments include the formalization of foundational proofs in metamathematics [98]; the CLI stack verifying a hardware processor, an assembler, a compiler, and an operating system kernel [9]; the verification of the seL4 hypervisor [66]; the verification of air traffic control algorithms [23]; the Four Color Theorem [53]; and the verification of compilers for realistic languages [73]. In all of these instances, the proofs were outlined by a human while the gaps were filled in either by the built-in automation (e.g., SMT solvers, automated theorem provers) or through user-defined automated proof strategies. The integration of automation and interaction has made it possible to efficiently develop large-scale formalizations in mathematics and computing.

2.3 Evidential Tool Bus

Most software analysis methods involve a workflow integrating multiple tools such as type checkers, static and dynamic analyzers, SAT and SMT solvers, termination checkers, and theorem provers. Examples of such workflows include counterexample-guided abstraction refinement, concolic execution, test generation, and compositional verification. An assurance case integrates a number of claims, both formal and informal, about the software. These claims can cover traceability between system requirements, software requirements, and various design and implementation artifacts such as the specification, source code, object code, test cases, and performance measurements. An assurance case consists of claims about these artifacts that are supported by arguments based on evidence. The arguments include formal deductive arguments as well as those based on empirical and subjective evaluation. Both the formal and informal parts of the argument can employ tools to process the evidence. A tool integration framework is needed to systematically define workflows involving multiple tools to construct arguments. In contrast to an *ad hoc* script for applying different tools, a tool integration framework offers a unified platform where tools can be plugged in and invoked as services, and woven into robust workflows.

The Evidential Tool Bus (ETB) is a distributed framework for orchestrating workflows for building assurance cases [94, 35]. ETB uses Datalog as its metalanguage both for representing workflows and arguments. Tools are invoked using Datalog predicates through wrappers, so that checking the satisfiability of an SMT-LIB formula in a file handle f, could be written as $smtCheck(f, Result)$, where the variable *Result* is bound to either `sat` or `unsat`. Workflows are defined by Datalog programs so that for example, a bounded model checking query could be written as the Horn clause

$$bmc(M,P,K,Result) \\ :- \quad unfold(M,P,K,F), smtCheck(F,Result),$$

where *unfold* is implemented by a tool wrapper that takes a model M (from a file) and property P, and generates the unfolded formula F (in a file). Evaluating a query like $bmc(m,p,k,Result)$ on a specific model m, property p, and bound k triggers the corresponding execution of the wrappers associated with the predicates *unfold* and *smtCheck*. An ETB network consists of a collection of servers, where each server can offer a set of services. These services can be accessed through client interface for uploading files and launching queries to interactively construct the artifacts, claims, evidence, and arguments that go into an assurance case.

3. SOFTWARE ENGINEERING TOOLS

Here we consider the software engineering domains that most benefit from automated deductions technologies. We will try to provide a reasonable, but by no means complete, list of tools and techniques and how they use automated deductions tools.

Proof Assistants: One of the classic ways of using automated deduction during a verification process is when automated theorem proving is used to assist humans in proving functional properties of code. One of the best, recent, examples of this is the verification of the seL4 microkernel by using the Isabelle/HOL theorem prover[66]. In general theorem provers like ACL2, Coq, HOL, Isabelle, Nuprl, and PVS can be used for formalizing a range of computational models, including the semantics of programming languages and hardware representations. ACL2 formalizes an applicative fragment of the Common Lisp programming language. However the human effort required for using these tools to verify software can be prohibitive. For example, the seL4 effort required 20 person years for doing the proofs, and the authors estimate it will take 6 person years, if they had to reuse the methodology on a similar project. This amount of effort is why completely automated approaches are far more popular.

Model checking is one of the most popular fully automated approaches to verifying properties of software systems. Most model checkers analyze designs of systems expressed in various custom notations, but more recently there has been a move to analyze code written in C, Java and other programming languages. Some model checkers, especially so-called explicit-state model checkers, such as SPIN [59] and Java PathFinder (JPF) [111] use almost no automated deduction in their core functionality. Symbolic model checkers (for example, in NuSMV [28] and SAL [39]), using binary decision diagrams (BDDs), can in certain cases analyze extremely large state spaces by exploiting some of the special properties of BDDs. Bounded model checkers [29], that translates transition systems and properties to be checked into constraints uses SAT and SMT solvers as backends. Another approach to model checking that has been very popular is the use of abstraction and refine-

Year	Program Verifier	Solver	Reference
1979	Pascal Verifier	Custom	[81]
1998	ESC Modula-3	Simplify	[71, 42]
2002	ESC-Java	Simplify	[48, 26]
2003	SPARK	Custom	[4]
2004	Spec#	Boogie/Z3	[5]
2007	KeY	Custom	[7, 96]
2009	VCC	Boogie/Z3	[32]
2010	HAVOC	Boogie/Z3	[3]
2010	Dafny	Boogie/Z3	[70].
2012	Frama-C	Various	[36]

Table 3: Program Verifiers and their Solvers

ment in the so-called counter-example guided abstraction refinement (CEGAR) loop [30]. Here automated deduction tools such as SMT solvers and other decision procedures (especially for linear arithmetic) plays a major role. The approach involves starting with an over approximation of a system description and then checking whether a possible counter-example (i.e. a path to an error) is feasible for the real system; if it is not a decision procedure is used to derive a new abstract system that is refined according to predicates that make the infeasible path no longer executable. This process continues until a real counter-example is found or an abstract system doesn't exhibit an error. Some of the first model checkers targeting source code, used this approach: Static Driver Verifier (previously SLAM) [2] and BLAST [10].

Program Verifiers: There are a large number of verification tools that operate at the level of annotated code for C, Java, and related languages (see Table 3). The Pascal Verifier [81], followed by ESC Modula-3 [71], ESC-Java [48, 26], and the Spec# [5] system addressed extended type safety of programs: programs that successfully pass a check by these systems are guaranteed to not encounter a class of runtime errors, such as division by 0 and null-pointer dereferences. In order to ensure these properties, these tools rely on Floyd-Hoare-Dijkstra calculi for analyzing program statements that may potentially produce a runtime error. The result of the analysis is a logical formula that is passed to a theorem prover. An experience has been that the more suitable automated deduction tools for this task combine strong domain reasoning for data-types that are part of programs (integers, bit-vectors, pointers), with relatively light-weight handling of quantification (to encode properties of data-structures and heaps). The Simplify [41] theorem prover was built to support ESC Modula-3 [42] and was since used in many software engineering projects. Simplify is a so-called SMT solver and modern SMT solvers have by now supplanted Simplify. The Boogie intermediate language and tool [72] is intended as an intermediate layer on which program verifiers can be built. It uses the Z3 SMT solver to check properties of the annotated Boogie code. A number of program verifiers use Boogie as a backend: VCC [32], HAVOC [3] and Dafny [70]. Frama-C is a software analysis platform for C code that supports various analysis plug-ins [36]. One of these plug-ins support weakest pre-condition based reasoning that discharges verification conditions with the Alt-ergo SMT solver or the Coq proof assistant. A related plugin uses Jessie which in turn uses the Why3 platform (that the Krakatoa tool for Java also uses as a backend) that can translate the wide variety of proof assistants and automatic solvers. A more crosscutting approach is taken by the KeY formal software development tool [7] which can analyze UML diagrams to check Object Constraint Language (OCL) specifications as well as Java code. It uses its own theorem prover backend that supports first-order dynamic logic for Java [96]. One of the most successful commercial applications of verification technology is AdaCore's SPARK tool [4], that analyses annotations in a restricted subset of Ada code. It uses the *Examiner* tool to analyze the code for conformance to the semantics of the language subset and generates verification conditions that are then analyzed by the *Simplifier* tool.

Symbolic execution: As mentioned in the introduction, a program analysis technique that has benefitted a great deal from the advances in automated deduction has been symbolic execution. Symbolic execution is a static analysis that "executes" code with symbolic inputs and collects constraints (path conditions) on the input for paths through the code. Whenever a branch condition is reached during the analysis both the true and the false branch condition is added and checked for feasibility. This check is performed with the aid of decision procedures, such as those found in SMT solvers. A number of symbolic execution based systems exists and two of the most popular is KLEE [24] and Symbolic PathFinder [89]. Dynamic symbolic execution (also referred to as concolic execution) has recently become very popular. The approach involves the concrete execution of a program, but on an instrumented environment where all the constraints on the path being executed can be recorded. One of the constraints is then negated and the resulting constraints are then checked for feasibility; if feasible a solution for each input variable is extracted and the code is run with these new inputs. A number of concolic execution systems exist with some of the most popular being DART, CUTE, CREST, Pex and SAGE [25]. For instance, SAGE checks for classic security issues such as buffer overflows and has had great success at Microsoft in finding serious issues that the standard fuzz testing missed [14].

Static runtime checking: Symbolic execution forms the basis of a number of static analysis tools that checks for runtime errors. The Prefix program analysis tool [22] pioneered large scale program analysis by building and using symbolic procedure summaries. It uses a custom incomplete theorem prover for checking path feasibility. The Z3 solver was integrated with Prefix to check path feasibility with bit-precise semantics such that Prefix could report integer overflow bugs. Coverity [78], likewise, relies on a combination of analyses for defect reporting. SAT solvers and other automated deduction engines have on occasion been mentioned although they seem somewhere deep in the trenches, engulfed by the hard reality of prioritizing actionable bugs to customers.

Test case generation: Writing unit tests by hand to obtain high code coverage can sometimes be a cumbersome exercise. Symbolic execution is ideally suited to this task, since a path condition that reaches the coverage condition (statement, branch, etc.) just need to be solved to find assignments to the input. Concolic execution systems of course requires the functionality to generate tests, by definition, since it needs to be able to run on the newly created inputs. The more classic symbolic execution tools, such as SPF, has the facility to generate tests for obtaining a number of different coverage criteria [112].

4. FUTURE OF DEDUCTION-BASED SOFTWARE ENGINEERING

Here we will highlight some new trends in software engineering and the requirements they have of automated deduction. We first consider changes that will likely come to automated deduction due to requirements in software engineering and internally (Section 4.1). In Section 4.2 we consider the role of competitions and how it influences the automated deduction field, and how it is starting to make an impact in software engineering. Standard notations

in automated deduction made a number of advances possible, and we we will consider the influence of these in Section 4.3. Lastly, in Section 4.4, we list a number of recent software engineering research directions that require the use of automated deduction tools.

4.1 Automated Deduction

We here outline selected trends in automated deduction of relevance to software engineering. We use these trends as an overall characterization of a very large number of efforts in the context of automated deduction and we describe how applications from software engineering have inspired and are driving these directions.

4.1.1 Scale and Expressive Power

An important enabling factor for making automated deduction relevant in software engineering tools has been fueled by the last decade of triumphs in SAT solving: from a very high-level point of view, the fundamental insights have been to understand how to prune search space efficiently using conflict learning, how to tailor data-structures for efficiently indexing and managing (memory) overhead during search, and how to prioritize search heuristics. These advances have been mainly inspired by applications in hardware and software engineering and one can expect this virtuous cycles to continue.

A foundational challenge in automated deduction is how to integrate first-order reasoning (with quantification) and theory solving. The combination leads quickly to highly intractable problem classes, but software engineering applications do not necessarily need to draw on the full power of both quantification and theories. As a case in point, automated deduction and heaps has been a very popular topic in the past decade [85]. Many heap reachability properties are essentially what can be characterized as *locally finite* theories: only a fixed finite number of quantifier instantiations are required to solve problems. The resulting ground problems can be solved using decision procedures. Combinations with monadic second-order logic are also possible and are used for analyzing programs with data-structures [75]. The encoding of model checking as satisfiability given in Section 3 produces quantified formulas that are furthermore Horn clauses with constraints (a Horn clause is a disjunction that has at most one positive uninterpreted relation, other uses of uninterpreted relations are negated; the constraints are arbitrary interpreted formulas over background theories). Automated deduction specialized to Horn clauses is an active area of research. Solving satisfiability of Horn clauses is intimately connected to finding inductive invariants for inductive program verification and overall proof by induction. The *EA* fragment (EFSMT), e.g., quantified formulas of the form $\exists\forall\varphi$, is also particularly well suited for several classes of systems analysis, including cyber-physical systems [27].

Practical methods for decidable classes and extended decidable classes is a fertile area where software engineering applications stimulate progress in automated deduction: The classical theory of *strings* is receiving current attention thanks to applications in security, such as verifying and synthesizing web-facing string sanitizers, and applications such as linguistics. The even more classical theory of *non-linear* arithmetic over polynomials is receiving a breath of fresh air thanks to insights from model-producing SAT solving search. Finally, sub-classes of first-order logic that are trivial from the point of view of expressive power (relative to full first-order logic) are finding applications in symbolic model checking of parametric hardware descriptions and heap manipulating programs. These fragments include QBF (Quantified Boolean Formulas), EPR (Effectively PRopositional logic, also known as the Bernay's Schönfinkel class), and even more succinct logics, such as quantified formulas over bit-vectors and uninterpreted functions.

4.1.2 Information from Deduction

Answering just the question whether a formula is satisfiable is valuable, but many applications need much more information. Test-case generation tools rely on *models* to produce test cases and high-integrity applications benefit from *proofs* to certify correctness beyond a blind trust in a debugged, but potentially buggy theorem prover. Other useful information can be extracted from automated deduction: Interpolants can be extracted as by-products of proofs, or alternatively as by-products of models and unsatisfiable cores. An example of an application that requires this functionality is automated fault localization, where an unsatisfiable core can indicate the part of the program that must be changed to remove an error. This is used within the BugAssist tool [63]. An extension of bug analysis is automated program repair [83] that rely on automated deduction tools for finding repairs, typically specified over a grammar of possible legal program fragments. Abstract interpretation based on logical formulas can use consequence finding (the set of consequences that can be derived from logical formulas). Quantitative information, such as extracting the number of solutions to a problem (#SAT), and obtaining *optimal* solutions from deduction problems is increasingly sought after. The number of solutions allow one to calculate the probability of an execution occurring in a program (given an input distribution) [51, 95] as well as the reliability of the code [46]. However this work has so far mostly been applied to programs manipulating linear integer arithmetic, whereas the domains of heaps and strings (for example) brings new challenges.

In an informal survey of software engineering researchers, the question was posed as to what would they want to see from automated deduction tools to enable new research directions and/or solve current problems they might have. The answer was almost unanimous: they want more insight into the working of the deduction systems. In short they want to know *why* the result was obtained, not just *what* the result is. This is not a new request and the ability to provide the unsatisfiable core to the user (rather than just to say a formula is unsatisfiable) is an example of where new research were spawned when the deduction tools provided some of the *why*. This is a trend we believe should accelerate in the future.

4.1.3 Deduction with Interaction

There are two related trends in interactive verification. One trend is toward integrating automated tools, particularly through the use of SAT and SMT solvers as well as first-order proof search engines. SAT and SMT solvers can greatly simplify reasoning in specialized theories and theory combinations, for example, the combination of bit-vectors, arrays, and uninterpreted function symbols. The second trend is toward increasing levels of trust. Some systems like HOL, HOL-Light, and Isabelle have simple logics with small kernels, but others like Coq and PVS have complex logics with correspondingly large proof kernels. The Kernel of Truth (KoT) project [100] associated with PVS defines a kernel for ZF set theory that can be used to capture the semantics of other formalisms. We will continue to see interactive theorem provers evolve toward greater automation coupled with deeper trust.

4.1.4 Evidential Tool Bus

The ETB architecture is evolving toward wider distribution from a system operating over a local-area network to one that is distributed across the internet. Deduction and analysis services can then be offered over the cloud, so that distributed workflows can be defined in terms of these web services. These workflows can

be operating continuously to preserve relationships defined by the workflows even as some of the inputs change.

4.2 Competitions

The automated deduction community uses a variety of competitions to stimulate advances in the field. The CADE ATP Systems Competition (CASC) [109, 88] that evaluates fully automatic theorem proving systems has been running since 1996 at the CADE and IJCAR conferences. The most recent one was with IJCAR 2013 [108]. A competition for SMT solvers, called SMT-COMP[3], has been running since 2005 and has been co-located with various conferences. Such competitions have a number of positive impacts on the field, not least of which is a set of benchmarks for comparison. Competitions is something the software engineering community can do well to try and emulate. They provide useful impetus to address a pervasive shortage of good benchmarks in the field.

Recently there has been some attempts at doing this, with the Software Verification Competition (SV-COMP) the most well known example. SV-COMP[4] has been running since 2012 and is co-located with the TACAS conference. The problem however is that to make a competition like this work it has to be carefully scoped, and for SV-COMP it is that they only evaluate tools using C as input. SV-COMP uses benchmark programs provided from a variety of sources, including the participants of the competition. The RERS challenge [60] on the other hand evaluates model checkers, but on artificially produced examples. The Static Analysis Tool Exposition (SATE[5]), run by NIST, attempts to evaluate static analysis tools and has been going since 2008. SATE is explicitly not a competition, but rather aims to collect large benchmarks, encourage tool development and to spur adoption of the tools via validation on large, real-world benchmarks. SATE has three language tracks (Java, C/C++ and PHP) and focusses on security vulnerabilities. SATE uses open source programs with known vulnerabilities as example cases, thus allowing tools to be evaluated on real-world examples. New competions directly relevant to software engineering problems are emerging: The SyGuS-COMP 2014[6] addresses synthesis problems and expressible as search over a grammar and SynthComp 2014[7] aims to evaluate reactive synthesis solvers.

We believe organizations with real-world problems should be encouraged to provide benchmark examples and to sponsor awards for solutions that they can adopt, in this way it will be a win-win situation.

4.3 Interoperability

One of the reasons why competitions are so successful in the automated deduction environment is that tools take standard notations as input (for example those mentioned before such as TPTP, DIMACS and SMT-LIB(2)). Note that automated deduction tools historically also supports their own custom interface in addition to the standard ones. A good practice when integrating with these tools is to interface with the standard notation, since this allows one to plug and play to see which tools delivers the best results. Often times the custom interfaces provide additional functionality and also the textual format of the standard notations can be expensive to parse, so once one determines the tool that performs the best it might be worth considering the custom interface.

A trend in software engineering tools is to use layers above the deduction tools to allow simple switching of the backend tools and to add some additional functionality that can speed up the analysis. For example in symbolic execution two such front-end tools are metaSMT [56] and GREEN [110]. metaSMT is used by KLEE [87] and provides a translation to the native interfaces of a variety of SAT and SMT solvers, thus reducing the burden of a developer to translate to each one themselves. GREEN also supports various backends, but its main feature is that it uses caching across analysis runs, to allow results from the deduction tools to be reused (KLEE also supports caching but only within one run).

4.4 New Software Engineering Trends

Here we list a few software engineering trends that we think we will have an impact on automated deduction technology in the future.

Synthesis: There is a renewed focus on program synthesis in various disguises [55, 104], such as synthesizing program fragments based on templates also known as sketching [105] and syntax guided synthesis. *inductive* methods: a set of input-output examples are supplied to a deductive engine that searches a state-space of program templates. Related to synthesis is the use of deduction for doing a semantic search for code in a library [106].

Model-Based Development: Some systems for model-based software development, such as [61, 62], rely increasingly on automated deduction engines. Not unlike the efforts on program synthesis, automated deduction is here used for synthesis, such as deriving system configurations. Model-based analysis of safety-critical algorithms, such as [80], has spawn substantial investment into automation and interactive tool support. This may lower the barrier of entry for future analysis efforts, and a possible future opportunity is to also use qualitative and quantitative results from automated deduction to synthesize parts of designs.

Education: Being able to program is becoming a required skill and many initiatives are springing up to try and teach people from all walks of life how to write code. The use of automated deduction to aid the understanding of what a piece of code is doing can facilitate this learning. An interesting example of this is the Pex4Fun project[8] that allows one to try and write code to match a hidden specification. Behind the scene the Z3 SMT solver is leveraged to generate test cases to show why the code fails or succeeds. One can also see how analyzing large numbers of programs and the mistakes programmers make can be used to find common programming mistakes which can then in turn help with teaching.

Trusted Security: has long been a hot topic and we have mentioned already the need for analyses over the string domain in this context. Moreover, recent prominence around privacy and security motivates security software to be open and certifiable (i.e. to ensure no unwanted surveillance code was inserted or that cryptographic functionality is not open to backdoors). This certification process will require new (or at least more efficient) approaches to program analysis and automated deduction.

High-integrity Systems: Besides the security demands already mentioned, high-integrity software systems can also require stringent safety demands to be met. Avionics software is a well known example where demanding certification processes must be adhered to in order to be used on commercial airplanes. DO-178C, the most recent version of the certification process required for flight software now includes mention of formal methods based approaches for the first time. However, the burden for certification is about to tested by the widespread use of self-driving car technology. Exactly, how one will be able to certify such software is an open problem both from a technical point of view (since these are probabilis-

[3] http://smtcomp.sourceforge.net/
[4] http://sv-comp.sosy-lab.org/
[5] http://samate.nist.gov/SATE.html
[6] http://www.sygus.org
[7] http://www.syntcomp.org/
[8] http://pex4fun.com

tic systems) and from a procedural point of view (how to make all the car manufactures work together). The analysis of probabilistic systems is a popular new research direction with techniques using probabilistic and statistical model checking [68, 69] as well as probabilistic symbolic execution [51] appearing recently.

Biology: Although not strictly speaking software engineering, the analysis of biological systems with the use of techniques from the analysis of software and hardware is becoming an increasingly active research field [47]. Although these analyses already use automated deduction (see for example the use of Z3 in [34]) we believe new theories and search algorithms may be required especially for analyzing the probabilistic aspects of the systems.

5. CONCLUSIONS

Software engineering has benefitted greatly from the advances in automated deduction in the last few decades. Many great ideas from decades past, suddenly became feasible with the major advances in automated deduction. Similarly the constant optimizations of the deduction tools and the addition of new theories has kept on fueling new software engineering research. However, it is fair to say software engineering is starting to give something back: research in automated deduction is now also driven by software engineering requirements. Automated deduction tools tended to be very general in the past, but in future some aspects will be fine-tuned for the more structured problems coming from software engineering. In addition deduction tools tended to be almost black-box, but now software engineers want to look under the hood to make use of some of the internal results. We foresee a much closer relationship between software engineering tools and the underlying automated deduction tool infrastructure.

6. ACKNOWLEDGMENTS

This work is based on the research supported in part by the National Research Foundation of South Africa (Grant Number 88210), as well by NASA Cooperative Agreement NNA13AC55C and NSF Grant CNS-0917375.

7. REFERENCES

[1] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In C. Boutilier, editor, *IJCAI*, pages 399–404, 2009.

[2] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 119–122, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2010.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[6] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. Coq home page: `http://coq.inria.fr/`.

[9] W. R. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, Dec. 1989.

[10] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.

[11] A. Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *Proceedings of SAT Competition 2013*, volume B-2013-1 of Department of Computer Science Series of Publications B, pages 51–52, 2013.

[12] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[13] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. The Barcelogic SMT solver. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer, 2008.

[14] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.

[15] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.

[16] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, NY, 1988.

[17] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer Berlin Heidelberg, 2009.

[18] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer, 2008.

[19] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[20] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[21] M. Buro and H. K. Büning. Report on a SAT Competition. *Bulletin of the EATCS*, 49, 1993.

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

[23] R. Butler, G. Hagen, J. Maddalon, C. Muñoz, A. Narkawicz, and G. Dowek. How formal methods impels discovery: A short history of an air traffic management

project. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 34–46, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[24] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[25] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 1066–1071. ACM, 2011.

[26] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2005.

[27] C.-H. Cheng, N. Shankar, H. Ruess, and S. Bensalem. EFSMT: A logical framework for cyber-physical systems. *arXiv preprint arXiv:1306.3456*, 2013.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002.

[29] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[30] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, Sept. 2003.

[31] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, Sept. 1976.

[32] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[33] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986. Nuprl home page: `http://www.cs.cornell.edu/Info/Projects/NuPRL/`.

[34] B. Cook, J. Fisher, E. Krepska, and N. Piterman. Proving stabilization of biological systems. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 134–149. Springer Berlin Heidelberg, 2011.

[35] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool Integration with the Evidential Tool Bus. In *VMCAI*, pages 275–294, 2013.

[36] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C a software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[37] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[38] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[39] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer Berlin Heidelberg, 2004.

[40] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[41] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[42] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[43] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[44] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[45] B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman. An assessment of techniques for proving program correctness. *ACM Comput. Surv.*, 4(2):97–147, 1972.

[46] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in Symbolic Pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[47] J. Fisher, D. Harel, and T. A. Henzinger. Biology as reactivity. *Commun. ACM*, 54(10):72–82, Oct. 2011.

[48] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In J. Knoop and L. J. Hendren, editors, *PLDI*, pages 234–245. ACM, 2002.

[49] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, volume XIX, pages 19–32. American Mathematical Society, Providence, Rhode Island, 1967.

[50] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[51] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 166–176, New York, NY, USA, 2012. ACM.

[52] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting combinatorial search through randomization. In J. Mostow and C. Rich, editors, *AAAI/IAAI*, pages 431–437. AAAI Press / The MIT Press, 1998.

[53] G. Gonthier. Formal proof: The four-color theorem. *Notices of the AMS*, 55(11):1382–1394, Dec. 2008.

[54] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993. HOL home page: `http://www.cl.cam.ac.uk/Research/HVG/HOL/`.

[55] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM.

[56] F. Haedicke, S. Frehse, G. Fey, D. Grosse, and R. Drechsler. metaSMT: Focus On Your Application Not On Solver Integration. In *Proceedings of the First International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS),*, 2011.

[57] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[58] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

[59] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[60] F. Howar, M. Isberner, M. Merten, B. Steffen, and D. Beyer. The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems. In T. Margaria and B. Steffen, editors, *ISoLA (1)*, volume 7609 of *Lecture Notes in Computer Science*, pages 608–614. Springer, 2012.

[61] D. Jackson. Alloy: A new technology for software modelling. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, page 20. Springer, 2002.

[62] E. K. Jackson and W. Schulte. Formula 2.0: A language for formal specifications. In Z. Liu, J. Woodcock, and H. Zhu, editors, *ICTAC Training School on Software Engineering*, volume 8050 of *Lecture Notes in Computer Science*, pages 156–206. Springer, 2013.

[63] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *SIGPLAN Not.*, 46(6):437–446, June 2011.

[64] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Kluwer, 2000.

[65] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[66] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[67] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In Sharygina and Veith [102], pages 1–35.

[68] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. *SIGMETRICS Perform. Eval. Rev.*, 36(4):40–45, Mar. 2009.

[69] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer Berlin Heidelberg, 2010.

[70] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[71] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, 1998.

[72] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 312–327, Berlin, Heidelberg, 2010. Springer-Verlag.

[73] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[74] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA, Mar. 1979.

[75] P. Madhusudan and X. Qiu. Efficient Decision Procedures for Heaps Using STRAND. In E. Yahav, editor, *SAS*, volume 6887 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2011.

[76] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, 1996.

[77] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, 1960.

[78] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 554–564, New York, NY, USA, 2013. ACM.

[79] F. L. Morris and C. B. Jones. An Early Program Proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.

[80] C. A. Muñoz, V. Carreño, G. Dowek, and R. W. Butler. Formal verification of conflict detection algorithms. *STTT*, 4(3):371–380, 2003.

[81] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[82] J. v. Neumann and H. H. Goldstine. Planning and coding of problems for an electronic computing instrument. *Institute for Advanced Study, Princeton, New Jersey*, 1948. Reprinted in [113].

[83] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE*, pages 772–781. IEEE / ACM, 2013.

[84] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002. Isabelle home page:

http://isabelle.in.tum.de/.

[85] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[86] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEETSE*, 21(2):107–125, Feb. 1995. PVS home page: http://pvs.csl.sri.com.

[87] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In Sharygina and Veith [102], pages 53–68.

[88] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[89] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[90] S. Rajan, N. Shankar, and M. Srivas. An integration of model-checking with automated proof checking. In P. Wolper, editor, *CAV*, volume 939 of *LNCS*, pages 84–97, Liege, Belgium, June 1995. Springer Verlag.

[91] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science, 2001.

[92] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965. Reprinted in Siekmann and Wrightson [103], pages 397–415.

[93] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, CSL, MP, Feb. 1991.

[94] J. M. Rushby. An evidential tool bus. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 36–36. Springer, 2005.

[95] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 447–458, New York, NY, USA, 2013. ACM.

[96] P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In B. Beckert and C. Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.

[97] S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[98] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*. Cambridge Tracts in Theoretical Computer Science. CUP, Cambridge, UK, 1994.

[99] N. Shankar. Using decision procedures with a higher-order logic. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *LNCS*, pages 5–26, Edinburgh, Scotland, Sept. 2001. Springer Verlag.

[100] N. Shankar. Trust and automation in verification tools. In S. S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *6th International Symposium on Automated Technology for Verification and Analysis (ATVA 2008)*, volume 5311 of *LNCS*, pages 4–17. Springer Verlag, Oct. 2008.

[101] N. Shankar. Automated deduction for verification. *ACM Comput. Surv*, 41(4):20:1–56, 2009.

[102] N. Sharygina and H. Veith, editors. *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*. Springer, 2013.

[103] J. Siekmann and G. Wrightson, editors. *Automation of Reasoning: Classical Papers on Computational Logic, Volumes 1 & 2*. Springer-Verlag, 1983.

[104] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.

[105] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[106] K. T. Stolee and S. Elbaum. Toward Semantic Search via SMT Solver. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 25:1–25:4, New York, NY, USA, 2012. ACM.

[107] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[108] G. Sutcliffe. The 6th IJCAR Automated Theorem Proving System Competition - CASC-J6. *AI Communications*, 26(2):211–223, 2013.

[109] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[110] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[111] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, Apr. 2003.

[112] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[113] J. von Neumann. *John von Neumann, Collected Works, Volume V*. Pergamon Press, Oxford, 1961.

[114] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer-Verlag, July 27-30 2002.