From *Communications of the Computer Society of India*, Formal Methods Theme Issue, Vol. 31, No. 2, May 2007, pp. 28–32.
Archived in Journal of Universal Computer Science Vol. 13, No. 5, pp. 650–660

Automated Formal Methods Enter the Mainstream

John Rushby
Computer Science Laboratory, SRI International
Menlo Park, California USA email: rushby@csl.sri.com

I. Introduction

Formal methods are approaches to the design and analysis of computer-based systems and software that employ mathematical techniques. They directly correspond to the mathematical techniques used in other fields of engineering, such as finite-element analysis for bridges and other structures, or computational fluid dynamics for airplane wings and streamlined cars.

Each field of engineering develops ways to model its designs and the phenomena of concern in some branch of mathematics so that calculation can be used to predict and explore the properties and behaviors of designs prior to construction. Most traditional engineering fields deal with physical systems, so the appropriate branches of mathematics are typically based on differential equations. Computers, on the other hand, deal with symbols: their hardware and software set up patterns of 0s and 1s that represent some state of affairs (e.g., the numbers 20 and 100), and perform operations on these symbols that yield a representation of some new state of affairs (e.g., a list of the prime numbers between 20 and 100, or the control signals to be sent to an actuator that will slew a telescope from its current bearing of 20° to a desired bearing of 100°). The branch of mathematics that deals with symbols and their manipulation and their relation to the world is formal logic, and this is the origin of the "formal" in formal methods.

Automated calculation in formal logic—that is to say, automated deduction, or theorem proving—developed more slowly than calculation in traditional engineering mathematics. In part this was because many researchers in the field had different agendas than raw calculational efficiency—they were interested, for example, in using theorem proving in the teaching of logic, or in reproducing mathematical proofs, or in modeling cognition for Artificial Intelligence—and in part it was because calculation in formal logic is intrinsically harder than most numerical mathematics: almost all computational problems in formal logic are at least NP-hard, many are superexponential, and some are undecidable. An intuitive reason why many of these problems are so hard is that they deal with discrete choices—corresponding to the `if-then-else` and iterative control structures that give computers their power—and the total number of possible combinations that has to be considered is exponential in the number of choices. But this exponential is also the reason why software is so hard to get right, and why testing is not very effective: a feasible number of tests examines only a tiny fraction of the possible number of different behaviors and, because behavior involving discrete choices is discontinuous, there is no basis for extrapolating from examined to unexamined cases. The attraction of modern formal methods is that, despite the intrinsic computational complexity, they often can perform effective calculations and thereby examine *all possible* behaviors.

Effective formal calculation was not available when formal methods began their evolution. Early formal methods focused on mathematical approaches to programming, on proving small programs correct by hand, and on formal specification languages. These early methods stimulated much research and produced many fundamental insights, but they did not attract enthusiasm from practicing software engineers and programmers. They were included, nonetheless, in some university courses and were advocated or required for certain kinds of critical systems—and this led to a widespread perception that formal methods are an esoteric activity of little practical utility that should be undertaken only if insisted upon by a professor or regulator.

Recently, however, a number of developments have combined to make formal methods an attractive technology for many areas of software engineering. First, steady progress in the automation that can be applied to formal calculations, coupled with the raw power of modern processors, has allowed construction of tools that deliver practical benefits in return for a modest investment in formal methods. Second, developers of formal methods and tools learned that outcomes that fall short of full functional correctness can be valuable nonetheless; examples include extended static checking, debugging and exploration, test case generation, and verification of limited properties. Third, the emergence of model-based development caused formally analyzable artifacts to become available early in the development lifecycle.

The remainder of this paper introduces some of the ways in which formal methods can be applied in software and systems engineering, outlines the technology that underlies these applications, reports on some industrial experience, and sketches some prospects for the future.

II. Applications of Formal Methods

Extended typechecking is probably the most straightforwardly effective application of automated formal methods. Programmers are familiar with the typechecking performed by a Java or Ada compiler, and the rather weaker checking performed in C; the Java and Ada typecheckers catch errors such as adding a number to a Boolean, and even C will protest if an array is added to a number. Typechecking by a compiler is actually a simple kind of formal analysis but, because it is expected to operate in linear time, and to deliver no false alarms, its power is quite limited. If we are prepared to use methods of formal calculation that are potentially more expensive, then we can extend the range of properties that can be checked. For example, an extended typechecker can warn about programs that might generate runtime exceptions due to division by zero, array bounds violation, or dereferencing a null pointer.

It is difficult to make such analyses precise, so we generally tolerate some false alarms, and may also allow some errors to go undetected. The balance between these kinds of imprecisions can be adjusted according to the needs of the application: for debugging, we may allow some real errors to go undetected in return for few false alarms, while for certification we tolerate more false alarms in return for guaranteed detection of all real errors in the analyzed class.

False alarms and undetected errors can arise for several reasons. One is related to the power of the automated deduction involved. Suppose for example, that an extended typechecker looking for possible divide by zero exceptions encounters the expression $q/(z+2)$ in a context where $z = 3 \times x + 6 \times y - 1$, and x and y are arbitrary integers. A theorem prover that is competent for integer linear arithmetic will easily establish that $z + 2$ is nonzero under these constraints, whereas one that lacks special knowledge about integers will be unable to do so and will generate a false alarm. A different source of difficulty lies in programming constructs such as heap structures and pointers; formal analyzers may maintain only abstract properties of heap objects (e.g., their "shape," such as whether they are a list or a tree) and will raise a warning for any division operation whose denominator comes from the heap. Much of the research and recent progress in formal methods addresses these difficulties, but there are limits to what can be achieved without additional information from the programmer.

There are two kinds of information that a programmer can supply to a formal analyzer: information that tells it what to check for, and information that helps it do its analysis. One fairly unobtrusive way to accomplish both of these is through extended type annotations: instead of declaring a variable to be simply an integer, we could specify that it is a `nonzero` integer, and we could further specify that an array contains `nonzero` integers, or that it is `sorted`. The analyzer must check that the extended type annotations are never violated, and it can then use them as assumptions in checking other properties. For example, if the type annotation for w declares it to be an `even` integer, then it becomes trivial to verify that $p/(w + 1)$ will not raise a division by zero exception.

Properties specified by types are *invariants*: they are required to hold throughout execution; simple assertions can be used to specify properties that must hold whenever execution reaches a given location, and technically these are also invariants (of the form "control at given location implies property"). Properties that are not invariants can be specified by alternative techniques that include those based on state machines, regular expressions, typestate annotations (in which allowed types can depend on the state), and temporal logic. These permit, for example, specification of the requirement that the lock on a resource should be alternately `claimed` and `released`: two `claims` in succession is an error. Formal analysis for invariants and these other kinds of properties often reveals bugs in otherwise well-tested programs. This is because *static analysis*, as this general class of automated formal methods is known, considers all possible executions, whereas dynamic analysis (i.e., testing) considers only the executions actually performed.

Formal methods and testing are not antithetic, however. Formal methods may assume properties of the environment (e.g., that the compiler for the programming language is correct, just as structural mechanics assumes properties of steel beams) that might be violated in particular cases—and testing in the real environment may reveal those violations. The specifications and methods of formal calculation that underlie static analysis also can be used in support of testing, thereby extending the range of benefits of formal methods. For example, specifications can be compiled into runtime monitors that check for violations and invoke exception handling [1]. The benefits over conventional testing are that the monitored properties are synthesized from specifications rather than programmed by hand (e.g., as `assert` and `print` statements), which is particularly advantageous where properties of concurrent programs are concerned. Another example is in the generation of test cases themselves: some methods of static analysis can construct a *counterexample* when a specification violation is detected, and this capability can be inverted to provide automated test case generation. The task of the test engineer then becomes that of specifying interesting properties to test, rather than constructing the test cases themselves.

To move beyond static analysis of local properties and toward full program verification—that is, toward analysis of properties that get to the essential purpose of the program, or to critical attributes of its operation, such as security or safety—requires specification methods capable of describing system requirements, designs, and implementation issues, together with methods and tools for verifying correctness across several levels of hierarchical development. Suitable specification languages and associated methods and tools for formal verification do exist (mostly associated with interactive theorem provers such as ACL2, COQ, HOL, Isabelle, and PVS [2]) but they require an investment of skill and time that currently limits their use to research environments and certain regulated classes of applications, such as the highest "Evaluation Assurance Levels" (EALs) for secure systems. Instead, approaches that seem most promising for wide deployment are those attached to model-based design environments such as Esterel/SCADE, Matlab/Simulink/Stateflow, or UML. These environments provide graphical specification notations based on concepts familiar or acceptable to engineers (e.g., control diagrams, state machines, sequence charts), methods for simulating or otherwise exercising specifications, and some means to generate or construct executable programs from the models. Until the advent of model-based methods, artifacts produced in the early stages of system development were generally descriptions in natural language, possibly augmented by tables and sketches. While they could be voluminous and precise, these documents were not amenable to any kind of formal analysis. Model-based methods have changed that: for the first time, early-lifecycle artifacts such as requirements, specifications, and outline designs have become available in forms that are useful for mechanized formal analysis.

Some of the notations used in model-based design environments have quite awkward semantics, but they present no insuperable difficulties and formal methods have been applied successfully to most model-based notations. Formal analysis can be used for model-based designs in similar ways to programs—for example, through extended typechecking and other methods of static analysis—but, because model-based descriptions can be higher-level and more abstract than programs, it is often possible to explore properties that are closer to the real intent or requirements for the design. Furthermore, because model-based designs are explored through simulation, they generally include a model of the environment (e.g., the controlled plant, in the case of embedded systems) and thereby provide a more complete foundation for formal analysis than an isolated program. In many cases, the environment model is a *hybrid system* (i.e., a state machine with real-valued variables and differential equations) and the properties of interest involve real time. Formal analysis of hybrid and timed systems is challenging, but is becoming effective.

Intermediate-level requirements are often stated informally as "shalls" and "shall nots" (e.g., "the microwave shall not be on when the door is open"), sometimes with temporal constraints added (e.g., "the windshield wipers shall turn off if no rain is detected for 10 seconds"). These can be formalized as invariants—and given a sufficiently rich extended type system (e.g., one that has predicate subtypes) they can even be stated as types—but these approaches require extensions to the modeling notation. Engineers may prefer to stay within their familiar framework, so an alternative approach is to specify requirements as *synchronous observers*: that is, as separate models that observe the state of the system and raise an error flag when a requirement is violated (rather like an `assert` statement in a program). Formal analysis then searches for circumstances that can raise the error flag, or verifies their absence. Counterexamples generated by formal analysis can be used to drive the simulator of the modeling environment, or they can be presented to the user in one of its modeling notations (e.g., as message sequence charts). By modifying the observer to recognize interesting, rather than undesired, circumstances, we can use formal analysis to help explore models (e.g., "show me an execution in which both these states are active and this value is zero"), and a variant of this approach can be used to generate test cases at the integration and system levels.

As these examples illustrate, the applications of formal methods in software engineering are becoming increasingly diverse and powerful. I now provide a brief survey of the technology that has enabled these advances.

III. The Technology of Formal Methods

The applications of formal methods outlined in the previous section are made possible by remarkable recent progress in the automation of formal calculations. This progress is not the result of any single breakthrough: it comes from steady progress in a number of basic technologies and from new methods for using these technologies in synergistic combination.

I gave an example earlier where it was necessary to decide if $z + 2$ could be zero when $z = 3 \times x + 6 \times y - 1$, and x, y, and z are integers. A program that can solve this kind of problem is called a *decision procedure*; in this particular case it is a decision procedure for the theory of integer linear arithmetic. Efficient decision procedures are known for many theories that are useful in computer science and software engineering, including the unquantified forms of uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, and arrays; furthermore, these decision procedures can work together to solve problems in their combined theory.

A decision procedure reports whether a set of constraints is satisfiable or not; the example above is unsatisfiable on the integers but satisfiable on the reals—a satisfying instance is $x = -\frac{1}{3}$, $y = 0$, and $z = -2$. A decision procedure that can provide instances for satisfiable problems is called a *satisfiability solver*. Propositional calculus (the theory of "Boolean" variables and their connectives) is a particularly useful theory: it can be used to describe hardware circuits and many finite-state problems (these are encoded in propositional calculus through what amounts to circuit synthesis). A satisfiability solver for propositional calculus is known as a SAT solver, and modern SAT solvers can deal with problems having millions of variables and constraints. A satisfiability solver for the combination of propositional calculus with other theories is called an SMT solver (for Satisfiability Modulo Theories), and a recent surge of progress (spurred, like that of SAT solvers, by an annual competition) has led to SMT solvers that can handle large problems in a rich combination of theories [3].

Many formal calculation tasks can be formulated in terms of SAT or SMT solving. For example, to discover if a certain program or executable model can violate a given assertion in k or fewer steps (where k is a specific number, like 20), we translate the program or model into a state machine represented as a transition relation over the decided theories, conjoin the k-fold iteration of this relation with the negation of the assertion after each step, and submit the resulting formula to an SMT solver. A satisfying instance provides a counterexample that drives the program to a violation of the assertion. This approach is called *bounded model checking* and a modification (essentially, where the assertion is not negated) can be used for the generation of test cases or exploration instances. Invariants can be verified using a different modification called *k-induction* [3]. Unfortunately, k-induction is not a complete procedure and its complexity is exponential in k, so there are many true invariants that cannot be verified in this way. One way to strengthen it is to supply additional invariants. These could be suggested by the user but often a large number of rather trivial auxiliary invariants are needed and it is unreasonable to expect the user to suggest them all: instead, they must be discovered automatically.

One way to do this is by *dynamic analysis* [4]: a large number of putative invariants are generated systematically (e.g., for two variables x and y, we could generate $x = y$, $x < y$, $x \leq y$, $x \geq y$ and so on); then the program is run for a short period and monitored to see which of the putative invariants are violated during the execution. Those that survive are considered candidate invariants whose verification is worth attempting by k induction or other means.

Some other ways to generate or prove invariants use abstraction: that is, aspects of the program or model are approximated or simplified in some way in the hope that analysis will thereby become more tractable without losing so much detail that the desired property is no longer true. For example, we could replace an integer variable by one that records only its sign (i.e., `negative`, `zero`, or `positive`) and likewise abstract the operations on the variable so that, for example, `negative` + `negative` yields `negative` but `negative` + `positive` nondeterministically yields any of the three sign values. Then we simulate execution of the program to see if any of the abstracted variables converges to a fixed point, thereby yielding an invariant. This method of *abstract interpretation* [5] is sensitive to the abstraction chosen, to the way in which abstracted values are combined when different execution paths join, and to the "widening" method used to accelerate or force convergence to fixed points.

In addition to generating invariants, abstractions are also used in verifying them: because the abstracted system should be simpler, or have fewer states, than the original, a verification method that is defeated by the scale of the original may succeed on the abstraction. *Predicate abstraction* [6] is often used for this purpose; whereas abstract interpretation approximates the values of variables, predicate abstraction approximates relationships between variables; for example, we may eliminate two integer variables x and y and replace them by a trio of Boolean variables that keep track of whether or not $x < y$, $x = y$, or $x > y$. Decision procedures or SMT solvers are used in construction of the abstracted system corresponding to a given set of predicates. The resulting system will have a finite number of states so it becomes feasible to compute those that are reachable—perhaps by *explicit state model checking*, which is essentially exhaustive simulation, or by *symbolic model checking*, which usually employs a potent data structure called reduced ordered binary decision diagrams, or BDDs. Invariants are easily checked once the reachable states are known, and more general properties can be checked through translation to Büchi automata. Methods for abstraction and reachability analysis in timed and hybrid systems can be developed along similar lines to those for state machines.

The selection of predicates on which to abstract is a crucial choice; there are several good heuristics (e.g., start with the predicates appearing in guards or conditional expressions, or symbolically propagate desired invariants through the program) [7], but an initial selection may lose too much information, so that instead of verifying a desired invariant, analysis of the abstraction yields a counterexample. If the abstract counterexample yields a corresponding "concrete" one in the original system, then we have refuted the desired invariant; but if not, differences in the way the counterexample plays out in the abstracted and the original systems can suggest ways to refine the abstraction (e.g., by suggesting additional predicates). This technique is known as *counterexample-guided abstraction refinement* (CEGAR) [8]. Related methods alternate abstraction and test generation in the attempt to find just the right abstraction to make analysis tractable [9].

Several of these technologies can be used in interleaved combinations to yield highly automated tools (often referred to as *software model checkers*) that are able to analyze fairly intricate properties of source code with little guidance [10].

IV. Industrial Adoption

Formal methods have long occupied a niche in the development and assurance of safety-critical and secure systems. Recently, their appeal has expanded into more mainstream areas, aided by great increases in the power of the supporting technology, and by a pragmatic adjustment of goals. The first applications of formal methods to see industrial adoption are highly automated static analyzers that shield the user from direct contact with the underlying formal method.

For example, Coverity and several other companies provide static analyzers for C and some other programming languages, and Microsoft uses similar tools internally. These analyzers are mostly used for bug finding (i.e., they are tuned to minimize false alarms and cannot verify the absence of bugs) but still find many errors in widely used and well-tested software: for example, Coverity found an average of 0.434 bugs per 1,000 lines of code in 17.5 million lines of C code from open source projects [11]. The Spark Examiner from Praxis is tuned the other way: it may generate false alarms but does not miss any real errors within its scope—in particular, it can verify the absence of runtime exceptions in Ada programs. It has found errors in avionics code that had already been subjected to the testing and other assurance methods for the highest level of FAA certification (DO-178B Level A) [12]. T-Vec provides tools that can generate the tests required by the FAA [13], while Leirios focuses on model-based test generation [14].

The static driver verifier (SDV) of Microsoft uses many of the advanced techniques described earlier to verify absence of certain "blue screen" bugs in device drivers for Windows [15]; a related tool called Terminator can verify that drivers will not "hang" forever. The Astrée static analyzer uses abstract interpretation to guarantee absence of floating point errors (underflow, overflow etc) in flight control software for the Airbus A380 [16], while AbsInt uses abstract interpretation to calculate accurate estimates for worst-case execution time and stack usage [17]. The SCADE toolset from Esterel, also used by Airbus and for other embedded applications, has plugins for analysis and verification by bounded model checking and k-induction [18]. Rockwell-Collins has developed a prototype toolchain with similar capabilities for the Simulink/Stateflow model-based development environment from Mathworks and reports substantial improvements in effectiveness and reductions in cost in its process for reviewing requirements [19].

V. Future Prospects

Most of the applications of formal methods adopted in industry are fully automatic, highly specialized to the analysis of specific built-in properties, and focus on individual program units. I believe that high degrees of automation and specialization are essential for industrial use, but that future opportunities lie with user-specified properties and analysis of the interfaces between program units and between programs and their environments. The main impediment to this, and to analysis of more revealing and significant properties, is persuading programmers to specify them—and historically, they have been reluctant to do so because they saw little reward for their effort. I believe that as more programmers and more industries see benefits from tools of the kind mentioned in the previous section, so they may become willing to make the investment to annotate their programs and model-based designs with specifications and guidance for formal analysis.

System faults often arise at the interfaces between components. Extended type annotations for interfaces would allow formal analysis of limited—but better than current—checks that components respect their interfaces. Stronger checks require specification of how the interface is to be used (e.g., a protocol for interaction); *typestate* [20] and *interface automata* [21] provide ways to do this. Formal methods can then attempt to verify correct interface interactions, or can generate monitors to check them at runtime or test benches to explore them during development (rather like the bus functional models used in hardware). Analysis that is fully compositional is an important challenge for the future.

There is general consensus that the most significant problems in software development are due to inadequate requirements, especially where these concern what one component or subsystem may expect of another. Formal methods can help by calculating the properties that a component assumes of its environment. People are good at describing how things work but not so good at imagining how they can go wrong; using models to describe components and formal analysis to explore and calculate properties of their interaction combines the strengths of man and machine. Exploration of reachable states in the composition of a component with a model of its environment provides a way to generate information for requirements analysis, and exploration of the consequences of modeled faults or of scenarios leading to hazards can provide insights for safety analysis.

These more ambitious applications of formal methods require some advances in the supporting technology. Chief among these are improvements in quantifier elimination and nonlinear arithmetic in SMT solvers, and more powerful methods for automated generation of invariants. Large and novel problems will require cooperation among multiple specialized tools and a suitably open and loosely-coupled "tool bus" would provide a way to organize these federations [3].

In closing, I hope this brief survey encourages you to join me in looking forward to a not-too-distant future where automated formal methods are standard in every software development environment and their benefits are widely recognized and valued.

Acknowledgments: My colleagues R. DeLong, N. Shankar, and A. Tiwari helped me formulate and clarify this exposition, whose preparation was funded by NSF CNS-0644783.

References

[1] *Runtime Verif'n home*, http://www.runtime-verification.org/.
[2] N. Shankar, Ed., *IFIP Working Conference on Verified Software: Theories, Tools, and Experiments*, Zurich, Switzerland, Oct. 2005, available at http://vstte.inf.ethz.ch/papers.html.
[3] J. Rushby, "Harnessing disruptive innovation in formal verification," in *Fourth International Conference on Software Engineering and Formal Methods (SEFM)*, D. V. Hung and P. Pandya, Eds. Pune, India: IEEE Computer Society, Sep. 2006, pp. 21–28.
[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 2006.
[5] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *4th ACM Symposium on Principles of Programming Languages*. Los Angeles, CA: Jan. 1977, pp. 238–252.
[6] H. Saïdi and S. Graf, "Construction of abstract state graphs with PVS," in *Computer-Aided Verification, CAV '97*, Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254. Haifa, Israel: Jun. 1997, pp. 72–83.
[7] C. Flanagan and S. Qadeer, "Predicate abstraction for software verification," in *29th ACM Symposium on Principles of Programming Languages*. Portland, OR: Jan. 2002, pp. 191–202.
[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer-Aided Verification, CAV '2000*, Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Chicago, IL: Jul. 2000, pp. 154–169.
[9] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: A new algorithm for property checking," in *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE)*. Portland, OR: ACM Press, Nov. 2006, pp. 117–127.
[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, Lecture Notes in Computer Science, vol. 2648. May 2003, pp. 235–239.
[11] *Scan home page*, http://scan.coverity.com/.
[12] A. German, "Software static code analysis lessons learned," *Crosstalk*, Nov. 2003, available at http://www.stsc.hill.af.mil/crosstalk/2003/11/0311German.html.
[13] *T-Vec home page*, http://www.t-vec.com/.
[14] M. Utting and B. Legeard, *Practical Model-Based Testing*. Morgan Kaufmann, 2006.
[15] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft," Microsoft Research, Tech. Rep. MSR-TR-2004-08, Jan. 2004.
[16] *Astrée home page*, http://www.astree.ens.fr/.
[17] *AbsInt home page*, http://www.absint.com/.
[18] *SCADE Design Verifier*, http://www.esterel-technologies.com/products/scade-suite/design-verifier.
[19] S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl, "Proving the shalls," in *International Symposium of Formal Methods Europe, FME 2003*, Lecture Notes in Computer Science, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805. Pisa, Italy: Mar. 2001, pp. 75–93.
[20] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 157–171, Jan. 1986.
[21] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*. ACM Press, 2001, pp. 109–120.