DOI:10.1145/1378727.1378742

The answer to software reliability concerns may lie in formal methods.

BY MIKE HINCHEY, MICHAEL JACKSON, PATRICK COUSOT, BYRON COOK, JONATHAN P. BOWEN, AND TIZIANA MARGARIA

Software Engineering and Formal Methods

THE SOFTWARE ENGINEERING community has devised many techniques, tools, and approaches aimed at improving software reliability and dependability. These have had varying degrees of success, some with better results in particular domains than others, or in particular classes of applications. A popular, although not uncontroversial, approach is known as *formal methods*, whereby a specification notation with formal semantics, along with a deductive apparatus for reasoning, is used to specify, design, analyze, and ultimately implement a hardware or software (or hybrid) system.

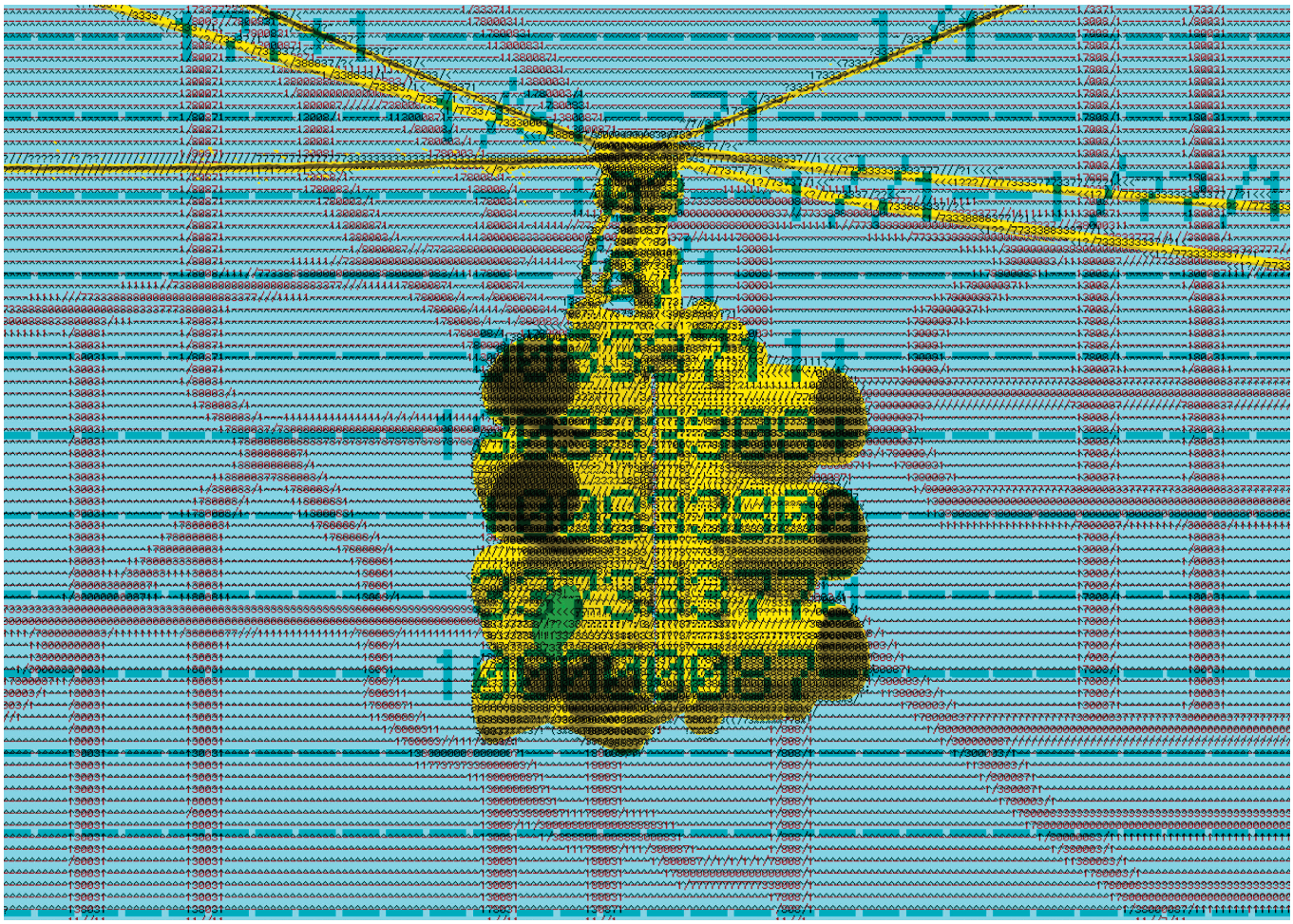
Such an approach is often thought to be difficult to apply and to require significant mathematical experience.[1, 11] Experience has demonstrated that developers without significant mathematical ability can at least understand and use formal specifications, even if greater mathematical ability and *specialization* in various areas of mathematics and engineering are needed for more challenging formal methods-related activities such as verification and refinement, and even the task of writing formal specifications themselves. As automated formal methods slowly become a natural part of the design process, we also experience higher dependability levels as a standard trait of software products.

A key issue, however, is the need for those applying formal methods to be able to abstract and to model systems at an appropriate level of representation—that is, to develop solid design principles and apply them to software development. This is particularly true when proving properties of more complex systems involving significant concurrency and interaction among components. When we wish to prove properties, it is often easier—and necessary—to prove them at a more abstract level, exploiting the idea of *abstract interpretation*.

This article contains contributions from three world-renowned experts in the fields of software engineering, abstract interpretation, and verification of concurrent systems: Michael A. Jackson, Patrick Cousot, and Byron Cook. Their contributions are based on their keynote speeches at the IEEE's Fifth International Conference on Software Engineering and Formal Methods, held in London, Sept. 10–14, 2007. The aim of the conference series is to bring together practitioners and researchers in the fields of formal methods and software engineering with the goal of exploiting synergies and furthering our understanding of specialization, abstraction, and verification techniques, among other areas.

Declining Dependability Levels

Computer-based systems are pervasive and now influence almost every facet of our lives. They wake us in the morning, control the cooking of our food, entertain us in the guise of media players, help in avoiding traffic congestion, control or identify (via GPS navigation



systems) the location of the vehicles in which we travel, wash our clothes, and even give us cash from our bank accounts (sometimes!).

Computers are being used increasingly in applications where they can have great influence over our very existence.[2] They control the flow of trains in the subway, signaling on railway lines, even traffic lights on the street. The failure of any of these systems could cause great inconvenience and conceivably result in accidents in which lives are lost. As they control the actual flight of an aircraft, cooling systems in chemical plants, feedback loops in nuclear power stations, and so on, we can see that they all account for the possibility of great disasters if they fail to operate as expected.

More and more, these systems are *software intensive*, meaning that software is the major component and that much of the functionality is achieved via software rather than hardware implementations. This raises questions over the *reliability* (the measure of the ability of system to continue operating over time[12]) and the *dependability* (the property that reliance can justifiably be placed on the service it delivers[12]) of software.

A Software Crisis

This has become a major issue for the software engineering community. While hardware dependability has increased continually over the years, and with mean time to failure (a measure of dependability) for the most reliable systems now exceeding 100 years,[13] software has not kept up with this pattern and indeed has been exhibiting declining levels of dependability.[10]

This can be attributed to many factors, including:

- A naïve belief that anyone who can write software can write *good* software.
- A mistaken belief that running a few representative test cases indicates that the software is "correct" or adequate.
- Failure to understand that realizing a good *design* is more important than producing vast quantities of code and that the goal of software engineering is not only to produce code but also to produce trustworthy solutions to problems that can eventually be implemented in a programming language.
- Failure to realize that making changes to software—and in particular unnecessary, uncontrolled, and careless changes—can have an effect on its appropriateness and validity, its correct operation, and can make it less efficient, or in extreme cases obsolete.

Of course, software is *intended* to change, and must be able to change. If we were to write software that we would never change after deployment (to meet changing requirements, an evolving environment, or to correct errors or unimplemented or incorrectly implemented requirements), then we would be better off implementing everything in hardware; but this is neither technically possible nor financially or spatially feasible.

Michael Jackson: Specializing in Software Engineering

Software-intensive systems are intended to interact dependably with the human and physical problem world. Execution

ILLUSTRATION BY JOE MAGEE

of the software produces and uses real-time information about the world, and monitors and partly controls its behavior to provide required functionality and to satisfy required constraints.

Such systems pose a particular challenge, arising from the interaction between the quasi-formal nature of the software and the nonformal nature of its human and physical problem world outside the computer. The software in execution can be regarded as a formal system: for all but the most extremely critical systems the computer behavior can be assumed to conform to the program semantics. The nonformal problem world, in contrast, has many parts—human, natural, and engineered—whose properties and behavior are far less reliable. For a dependable system, there must be an adequate formal model of the problem world, and the software must be designed to reflect the assumption that the world conforms to this model. In executing its program, the computer always behaves as if the world model is valid; if the world deviates from the model, the system will fail in some way. The problem world, being nonformal, has unbounded possible behaviors and properties that can invalidate any formal model. Fault-tolerant techniques in which the model is weakened in suitably chosen respects can mitigate the difficulty, but they cannot eliminate it: the world can still invalidate the weakened model. How, then, can a dependable system be designed?

Development of the formal model is, above all, an engineering task. In the established engineering branches, the challenge is met by experience accumulated in each particular product class and captured in a *normal design discipline*. The aeronautical engineer W. G. Vincenti explains normal design: "The engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task." Family cars, for example, are the product of an industry-wide normal design discipline. Different manufacturers' products are strikingly similar both in their external appearance and in their internal workings because their designers adhere closely to the current normal design. This normal design embodies the accumulated knowledge of the models—both of the product itself and of its environment or problem world—that is adequate for a desired level of dependability: which deviations from a model are sufficiently improbable to be ignored; which are implicitly handled by the normal design configuration; and which must be explicitly considered in the calculations and checks mandated by the discipline of the normal design practice.

The engineer lacking an applicable normal design discipline must inevitably engage in *radical design*. In Vincenti's words: "In radical design, how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development." In short, radical design has a low expectation of dependability: it can be a good choice only in purely experimental work or where novelty and excitement are of the highest importance and dependability is little valued.

A normal design discipline does not arise easily or by chance. It rests on a culture of *specialization*, in which a community of organizations and individuals is devoted to gradual, long-term evolution of normal design in a particular product class and of the scientific and engineering knowledge it embodies. The prerequisites for this culture of specialization are two: the continuing existence of specialized communities, with their research and production facilities and their journals and conferences; and the desire of individual highly talented engineers in each generation to dedicate their professional careers to working within just one specialization. This infrastructure of a specialized community and the advances needed to evolve a successful normal design discipline are not achievable or sustainable by floating populations of itinerant generalists.

Software engineering and computer science already have many specializations: some in complexity theory, concurrency, real-time systems, programming languages, model checking, program proving, and distributed computing; and some in system software components such as compilers, database systems, virus checkers, and SAT (Boolean satisfiability problem) solvers. These specializations, however, are not enough. To build dependable software-intensive systems, we must emulate the established engineering branches, developing many more specializations that are sharply focused on narrow classes of end products and their component subsystems. Only the product-oriented specialist can achieve dependability by bringing together the contributions of specialists in all the different aspects, parts, and dimensions of the design task.

What particular specializations, then, are needed? We cannot answer this question in advance, or in any systematic or authoritative way. In a society that insists on dependable systems, with a software-engineering culture that values and motivates specialization, particular specializations will emerge and evolve in response to opportunities and challenges as they are recognized.

In sum, dependability in an engineering product must be based on a normal design discipline. Only in a product-oriented normal design discipline can all the necessary knowledge—tacit and explicit—be brought together to build a dependable system. This normal discipline in turn must be the product of a long-term community of engineers specializing in systems of the particular product class. For a critical software-intensive system, specialization is not optional.

Patrick Cousot: The Role of Abstract Interpretation in Formal Methods

Formal Verification Methods. In computer science and software engineering, *formal methods* are mathematically based techniques for the specification, development, and verification of software and hardware systems. They establish the satisfaction of a required property (called the *specification*) by a formal model (called the *semantics*) of the behavior of a system (for example, a program and its physical environment). The *semantic domain* is a set of all such formal models of system behaviors.

A *property* of the system is a set of semantic models that satisfy this property. The *satisfaction* of a specification by a system (more precisely by its semantics), which can equivalently be defined as the proof that its strongest property is a property of the system, is called its *collecting semantics*.

An example is the *trace semantics* of a programming language. The semantics of programs in the language describes all possible program executions as a set of traces over states chosen in a given set of possible states. Two successive states in a trace correspond to an elementary program computation step. An example of a property is the termination property stating that all execution traces should be finite.

Formal verification methods are very hard to put in practice because both the semantics and the specification of a complex system are extremely difficult to define. Even when this is possible, the proof cannot be automated (using a theorem prover, a model checker, or a static analyzer) without great computational costs. Considering all possible systems is even harder. Hence it is necessary either to work in the small (for example, model checking) or to consider approximations of large systems (for example, abstract model checking).

Abstract Interpretation. Abstract interpretation[5] is a theory of sound approximation of mathematical structures, in particular those involved in the description of the behavior of computer systems. To prove a property, the *abstraction* idea is to consider a sound overapproximation of the collecting semantics, a sound underapproximation of the property to be proven, and to make the correctness proof in the abstract.

For automated proofs, these abstractions must be computer-representable, so they are not chosen in the mathematical concrete domain but in an *abstract domain*. The correspondence is given by a *concretization function* mapping abstract properties to corresponding concrete properties. Formal verification being undecidable, the abstraction may be *incomplete*—that is, it produces *false alarms*, a case when a concrete property holds but this cannot be proved in the abstract for the given abstraction, which must therefore be refined. The *abstraction function* is the inverse of the concretization function. It maps concrete properties to their approximation in the abstract domain. Applied to the collecting semantics of a computer system, it formally provides an abstraction of the properties of the system. Abstract verification methods consist of designing an abstraction function that is coarse enough so that the abstract collecting semantics is computer-representable and effectively computable and is precise enough so that the abstract-collecting semantics implies the specification.

MICHAEL JACKSON

Development of the formal model is, above all, an engineering task. In the established engineering branches, the challenge is met by experience accumulated in each particular product class and captured in a normal design discipline.

Abstract interpretation formalizes the intuition about abstraction. It allows the systematic derivation of sound *reasoning methods* and *effective algorithms* for approximating undecidable or highly complex problems in various areas of computer science (semantics, verification and proof, model checking, static analysis, program transformation and optimization, typing, or software steganography). Its main current application is on the safety and security of complex hardware and software computer systems.

Verification by Static Analysis. Static code analysis is the fully automatic analysis of a computer system by direct inspection of the source or object code describing the system with respect to the semantics of the code (without executing programs, as in *dynamic analysis*). The proof is done in the abstract by effectively computing an abstraction of the collecting semantics of the system. Examples of successful static analyzers used in an industrial context are aiT (www.absint.com/ait used to compute an overapproximation of the worst-case execution time[8]) and Astrée (www.astree.ens.fr used to compute an overapproximation of the collecting semantics to prove the absence of runtime errors[6]).

Their growing success is a result of their being useful (such as, they tackle practical problems), sound (their results can be trusted), nonintrusive (end users do not have to alter their programming methods (for example, by producing the specification and abstract semantics that can be directly derived from the program text), realistic (applicable in any weird industrial environment), scalable (to millions of lines of code as found in actual industrial code), and conclusive (producing few or no false alarms).

The design of language-based semantics and abstractions is extremely difficult but possible on a well-defined family of programs (for example, synchronous, time-triggered, real-time, safety-critical, embedded software written or automatically generated in C for

Astrée). The abstraction of Astrée is designed by conjunction of many elementary abstractions that are individually simple to understand and implement. In the case of a false alarm, each abstraction can be adjusted in cost and precision by parameters and abstraction directives (whose inclusion in the code can be automated). If the false alarm cannot be solved by the existing abstractions, new abstractions can be easily incorporated to extend and refine the conjunction of abstractions. The abstract invariants can therefore be strengthened in a few refinement steps until no false alarm is left.[7]

Byron Cook:
Verification of Concurrent Systems

Many critical applications involve high degrees of concurrency, whereby a number of activities progress in parallel with each other. Concurrency can create surprisingly complex interactions among the concurrent components, making the verification problem inherently more complex.

The traditional methods of specifying and automatically reasoning about computer systems are not sufficient for use with concurrent systems. They do not allow for the side effects caused by other concurrent components, the occurrence of multiple simultaneous events, or the synchronization required between processes to ensure data integrity, and so forth. Specialized tools will normally be required for effective verification.

As an example, consider thread termination. Concurrent programs are often designed such that certain functions executing within critical threads must terminate. Such functions can be found in operating systems, Web servers, and email clients. Until now, no known automatic program termination prover supported a practical method of proving the termination of threads.

As another example, concurrent programs are usually written in languages that assume but do not guarantee memory safety (such as, pointer dereferences never fail, and memory is never leaked). Again, until now, no known automatic program verifier supported the proving of memory safety for concurrent programs.

Recent advances now allow us to address these problems (such as, proving properties such as memory safety and termination of concurrent code. These advances have led to the recently added support for concurrency in the Terminator termination prover[15] and the SLAyer shape analysis engine.[14]

BYRON COOK

Concurrency can create surprisingly complex interactions among the concurrent components, making the verification problem inherently more complex.

The common theme of these advances is thread modularity: existing sequential-program provers can be used to prove the correctness of concurrent programs if appropriate abstractions can be found to represent the other threads in a concurrent system. In the case of termination proving, the abstractions describe the direction of change (called the variance) of values caused by the other threads in the system. We might know, for example, that the value of the variable x cannot go up in the other threads, thus allowing us to prove the termination of a loop in the thread of interest that decrements x during each loop iteration. In the case of memory-safety proving, the abstractions describe the association between locks and data structures: we can prove the memory safety of the thread of interest by assuming that certain data structures are safe to modify only when the associated locks are acquired.

The difficulty with thread-modular techniques is that the space of abstractions is so large that finding the right one for a given program verification problem is difficult. Heuristics must be developed. Furthermore, through the use of experimental evaluations, these heuristics must be shown to work in the common case. We have developed and evaluated such heuristics. In the case of termination proving, we have found that by temporarily ignoring concurrency we can guess a set of ranking functions (see Patterson[13] for details), which leads to a useful candidate abstraction. Using static program analysis techniques we can prove, for example, that every instruction in the other threads does not increment x. The fact that x needs not to be incremented, as opposed to y, comes from the proof of termination via the sequential-program termination prover (see Cook[4] for the details).

In the case of proving memory safety, we use initial guesses that match locks to variables, and then use counterexample-guided techniques to refine the shape of the data structures protected by the locks (see Gotman[9] for further details).

Conclusion

Dependability in software-intensive systems can be achieved only through the application of solid design principles. That, in turn, is achieved through an understanding of the product and specialization of engineers in particular domains, product types, and techniques. Abstract interpretation can aid in reducing the complexity inherent in proving properties and correctness of complex software systems, which otherwise would not be feasible. Such an approach facilitates automated reasoning and fully automated reasoning. In particular, recent advances in verification techniques have made it possible to verify concurrent systems and prove termination by considering termination of a thread without considering concurrency per se.

In the realm of formal methods, too, we encounter radical and normal engineering, as well as the whole spectrum of intermediate phases in between. In a sense, the ratio of radical design to normal design can be taken as a measure of the maturity of a field of engineering.

In retrospect, radical engineering of, and with, formal methods has characterized the first 20 years of this field, making it an art: it was limited to a restricted scope of few users and a few high-budget, high-risk projects for which high assurance was compelling. This is no longer the case today:

- A first transition from weak to strong formal methods[17] moved the field from specification-only toward tool-based semantics analysis, making formal methods not only descriptive but also operational.
- A second transition shifted tool support from heavyweight to lightweight formal methods: from proof assistants, which still require specialist skills, education, and a good deal of intuition, to fully automatic analyses embedded in a software engineer's usual development environment.

This shift is central to enabling every software developer to use the techniques seamlessly, advancing from a radical design endeavor to everyday practice. Then it is not an art anymore, just another craft, in the same way that photography enabled everybody to portray a subject with perfect faithfulness to the subject's traits, at a very reasonable cost and without being, or resorting to, a portrait painter.

We observe that today different techniques occupy different positions on this transition axis. While termination proof of concurrent systems is still an art (a case for radical design), techniques such as type checking are enforced by the vast majority of programming environments (natural design), and many other techniques are walking the naturalization path. It is a steep and narrow path that takes decades to follow. For example, abstract interpretation was discovered in the 1970s, but it is only in recent years that we have had tools such as Astrée to enable normal developers to apply collections of sophisticated abstract interpretations as part of their daily routine within an enhanced normal engineering process. Model checking, rewarded this year with the ACM A.M. Turing Award to Edmund M. Clarke, Joseph Sifakis, and E. Allen Emerson, is another successful example of formal methods that became increasingly natural in the course of their 25+ years of history. Once naturalized, the magic of a technique (and of its gurus) vanishes, but we all profit from the achievements of the revolutionaries that enter mainstream production.

As noted 12 years ago, specialization in high-assurance systems concerns devising appropriate heterogeneous methods that adequately exploit the various application-specific characteristics of the problem.[16] Computer-aided *method engineering* is the new craft. It targets understanding and solving problems *heterogeneously* at a meta level, where whole methods and paradigms are combined. Even though this holds already for many sequential systems, it is particularly true for distributed systems, which by their nature are of a much higher conceptual complexity. Multicore architectures in our laptops and massively multicore systems as part of Web computing and cloud computing environments demand increased attention here.

This quest continues... C

References

1. Bowen, J.P. and Hinchey, M.G. Seven more myths of formal methods. *IEEE Software 12*, 4 (July 1995), 34–41.
2. Bowen, J.P. and Hinchey, M.G. High-integrity system specification and design. *Formal Approaches to Computing and Information Technology*. Springer-Verlag, London, 1999.
3. Cook, B., Podelski, A., Rybalchenko, A. Termination proofs for systems code. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 415–426.
4. Cook, B., Podelski, A., Rybalchenko, A. Proving thread termination. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007*, 320–330.
5. Cousot. P. and Cousot, R. Systematic design of program analysis frameworks. In *Conf. Rec. 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, ACM Press (1979), pp 269–282.
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. Varieties of static analyzers: A comparison with Astrée. In *Proceedings of the 1st IEEE & IFIP Int. Symp. on Theoretical Aspects of Software Engineering, TASE '07*. IEEE Computer Society Press (2007), 3–17.
7. Delmas, D. and Souyris, J. Astrée: From research to industry. *Lecture Notes in Computer Science 4634*, Springer (2007), 437–451.
8. Ferdinand, C., Heckmann, R., and Wilhelm, R. Analyzing the worst-case execution time by abstract interpretation of executable code. *Lecture Notes in Computer Science 4147*, Springer (2006), 1–14.
9. Gotsman, A., Berdine, J., Cook, B., Sagiv, M. Thread-modular shape analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 266–277.
10. Gray, J. Dependability in the Internet era. In *Proceedings of the High Dependability Computing Consortium Conference*, Santa Cruz, CA, May 7, 2001.
11. Hall, J.A. Seven myths of formal methods. *IEEE Software 7*, 5 (Sept. 1990), 11–19.
12. Laprie, J.-C., ed. Dependability: Basic concepts and terminology in English, French, German, Italian and Japanese. *Dependable Computing and Fault-Tolerant Systems, Vol. 5*, Springer-Verlag, NY, 1992.
13. Patterson, D. et al. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Computer Science Technical Report UCB//CSD-02-1175. University of California, Berkeley, CA, March 15, 2002.
14. http://research.microsoft.com/SLAyer
15. http://research.microsoft.com/terminator
16. Steffen, B., Margaria, T. Method engineering for real-life concurrent systems. *ACM Computing Surveys, Special issue*: Position Statements on Strategic Directions in Computing Research 28, 4es (Dec. 1996) 56. ACM, NY.
17. Wolper, P. The meaning of "formal": From weak to strong formal methods. *STTT 1*, 1–2 (1997), 6–8. Springer Verlag.

Mike Hinchey (mike.hinchey@lero.ie) is codirector of Lero, the Irish Software Engineering Research Center, and professor of software engineering at the University of Limerick, Ireland.

Michael Jackson (jacksonma@acm.org) is Visiting Research Professor, Centre for Research in Computing, The Open University, Milton Keynes, England.

Patrick Cousot (Patrick.Cousot@ens.fr) is a professor of computer science at the École Normale Supérieure. He is a specialist in semantics, verification, and static analysis of programs and complex systems and is the inventor of abstract interpretation.

Byron Cook (bycook@microsoft.com) is a researcher at Microsoft's Laboratory at Cambridge University, where he has been working on the program termination prover Terminator, the shape analysis engine SLAyer, and the software model checker SLAM.

Jonathan P. Bowen (jpbowen@gmail.com) is chairman of Museophile Limited. He is contracted to work for Praxis High Integrity Systems, applying formal methods for software testing. He is also a visiting professor at King's College London and an emeritus professor at London South Bank University.

Tiziana Margaria (margarita@cs.uni-potsdam.de) is chair of service and software engineering at the Institute of Informatics, Universität Potsdam, Germany. She is also president of the European Association of Software Science and Technology (EASST).