Software that meets its Intent

Marieke Huisman[1], Herbert Bos[2] Sjaak Brinkkemper[3] Arie van Deursen[4]
Jan Friso Groote[5], Patricia Lago[2], Jaco van de Pol[1], and Eelco Visser[4]

[1] University of Twente, Enschede, The Netherlands
[2] Vrije Universiteit Amsterdam, The Netherlands
[3] Utrecht University, The Netherlands
[4] Delft University of Technology, The Netherlands
[5] Eindhoven University of Technology, The Netherlands

Abstract. Software is widely used, and society increasingly depends on its reliability. However, software has become so complex and it evolves so quickly that we fail to keep it under control. Therefore, we propose *intents*: fundamental laws that capture a software systems' intended behavior (resilient, secure, safe, sustainable, etc.). The realization of this idea requires novel theories, algorithms, tools, and techniques to discover, express, verify, and evolve software intents. Thus, future software systems will be able to verify themselves that they meet their intents. Moreover, they will be able to respond to deviations from intents through self-correction. In this article we propose a research agenda, outlining which novel theories, algorithms and tools are required.

1 Introduction

Problem. Software is everywhere: the estimated number of connected devices today ranges in the tens of billions and will be up in the hundreds of billions very soon. While society increasingly depends on its reliability and trustworthiness, the inconvenient truth is that software is very fragile. Software is so complicated that we have no clear idea of what a program may or may not do. And when something is working, we are afraid to touch it. For instance, many critical infrastructures do not update their software for fear of breaking things, even in the case of crucial security updates. As it stands we lack real control over software in virtually all areas. Both criminals and intelligence services employ this to hack into our computers. This is today. Tomorrow will be worse.

Solution. In 1942, the science fiction writer Isaac Asimov formulated a number of laws for robots. For instance, no matter how complex they became, they should never injure a human being, or through inaction, allow a human being to come to harm. In other words, the laws should govern the robot's behavior, independent of its normal functionality or complexity.

Today, some 75 years later, we take up this idea and apply it to software: by defining simple laws or "*intents*" for software, we want to govern a program's behavior, independent of the implementation of its functionality: it should not

crash, it should resist attacks, it should adhere to its functional specification. It should revert buggy updates, it should keep sensitive data private, etc.

Moreover, the intents should be met under any condition – even if the software evolves, or the environment changes. Where needed, the software should correct itself automatically to fulfil the intents. And whenever the intents change, the software has to adjust itself. In case conflicting intents for software are discovered, ethical considerations should be made to resolve the conflict.

This requires a research agenda, to investigate methods for discovering the dynamic intents of living software, techniques to specify and verify intents, and new mechanisms to make software self-correcting, *i.e.*, to make it adapt itself to deviations from the current intents. This requires the concerted efforts from a broad range of software technologies, combining formal analysis, experimental software laboratory research, and empirical research on deployed software. Developing and evaluating the methods also requires intimate collaboration with a number of embedded software application domains, like health care, robotics, automotive, and science experiments.

Software is becoming so versatile, all-embracing and intelligent that it is inconceivable that we can control all detailed aspects of its behavior. But by using intents we can guarantee that software adheres always to its essential purpose, avoids unnecessary harm, and operates within the boundaries of the law. With the same aim as Asimov's robot laws, intents guarantee that software can be used safely and reliably.

2 The Software Reliability Challenge

Software runs the world. Software plays an integral role in our daily lives. There is an 'app' for everything, from the energy grid that powers our infrastructure to the social media that mediate our social interactions. It is hard to recall what life was before all that software, and it is even harder to imagine what we would do without. Society would grind to a halt if the software running it would break down. This dependence on software *will keep growing* in the years to come. Gartner Inc. estimates that the number of connected devices today ranges in the tens of billions and that this will grow to hundreds of billions very soon [42].

The enormous growth in software [88] has led to many advances. Office software and information systems have revolutionized information processing in all sectors, from finance to retail to government. Software is now a crucial and often complex component in many *embedded systems* such as medical surgery robots; control systems for aviation, traffic, and nuclear equipment; and robot companions for elderly people. Software is also having a transforming impact on *science*: scientific experiments are often replaced by much faster, cheaper and safer software simulations, and scientific discoveries often result from automated analysis of large amounts of measurement data.

These successes stem from the fact that *computers outperform humans* in many tasks: computers are faster and more reliable; they can perform many tasks simultaneously, and can process astounding quantities of data. Already,

there are many cases where software 'knows' more about the tasks at hand than the humans involved, and this development has only just started.

However, the tremendous growth of software also has a downside. All software systems contain errors that cause them to behave in unintended ways [41, 62]. Studies show that even tested and deployed software has on average between 1 and 16 errors per 1000 lines of code [70, 71]. As a consequence, dormant software faults persist even in mature code bases [20]. With code bases that often exceed a million lines of code, modern software systems easily contain thousands of bugs, many of which may bring down the entire system, or worse, produce wrong results that go undetected. Most software errors are introduced by developers accidentally violating the (informal) expectations on a system. However, errors may also be introduced consciously to actively fool the users [89], which makes error detection even more challenging. Almost daily, we are exposed to news reports on the consequences of failures and other malbehavior of software.

Thus, even though many people think they can depend on the *reliability* of software, the truth is that that is just an idealized dream. Software has become *too complex*, and it *evolves* so quickly, that we are not really in control. Using artificial intelligence and autonomous learning techniques, software can adapt itself to its environment to outwit people, operating in a better way than any human could ever have programmed. This makes it scientifically very challenging to predict all possible emergent behaviors of a large, complex, and evolving software system, and thus to ensure that it is and will remain free of flaws, or to guarantee that it will always avoid foul play in obtaining its objectives.

Meanwhile, engineers keep developing new systems, for which new software has to be developed, while the software of old systems grows due to maintenance and adding new features. The resulting complexity is abused by various parties, such as criminals and intelligence services, actively searching for flaws in software systems in order to exploit them for their own advantage. There are even cases where software developers designed "features" in order to mislead customers and users, e.g. to circumvent environmental emission restrictions in cars. As a result, as a society we cannot guarantee the security and integrity of data, ultimately threatening the identity of our citizens. Moreover, as devices become increasingly *connected* and exchange information, the privacy of their users is at stake.

The root cause of this software reliability crisis is that the policies that software systems implement are buried in low-level code, which makes it hard to establish what exactly those policies are, whether they are implemented correctly, and whether these are indeed the policies that we intended. This problem is exacerbated by the rapidly increasing complexity of software systems. This would be akin to a society consisting of individuals that operate by doing as they please, driven by their own particular desires.

By contrast, modern society operates by *rule of law*, imposing constraints on the behavior of its citizens and institutions. While a modern society is a highly complex system with millions of entities interacting in many ways, rule of law keeps this system in check by imposing clear and objective criteria for judging the legality of the behavior of all these entities. To ensure that we can keep

depending on software and to prevent it from running wild, software should be governed by rule of law, just like our society!

3 Software Intents

Software laws, which we will call *software intents*, explicitly describe the essential limits on and expectations on the *behavior* of a software system. Software intents often cover functional requirements, but they should also cover non-functional requirements (e.g., intended performance, response time), and security properties (e.g., resilience against attackers). A crucial property of software intents is that they describe high-level requirements that reflect the goals of a software system, not (just) the low-level effects of a particular realization of those goals on a particular computing substrate.

Common examples of software intents are: Software should not crash; When software crashes nevertheless, it should do so gracefully; Software should not cause any harm to its users, or to any other human being; Software should be resistant to attacks; Software should satisfy its specifications; and: Software should not violate the privacy of its users. For a telling example, consider the recent case of the Japanese satellite Hitomi, which tumbled out of control five weeks after launch, because of a software error [96]: in an attempt to stop itself from spinning, the control system fired a thruster jet in the wrong direction, causing an acceleration, instead of a slow down. An implicit intent for a satellite is that it should stay in its orbit. By making this intent explicit, and providing support to ensure that this intent is fulfilled at all times, the error could have been detected and corrected.

Software intents serve multiple purposes. First, each software system should explicitly *declare* its intents, which serves as a promise and explanation what it intends to do. Second, intents should be usable to *verify* that a software system is indeed realizing its intended behavior (and no others). Finally, intents should facilitate the *evolution* of software systems, by ensuring that software maintenance is intent preserving. Technically, *intents* can be viewed as a form of redundancy, which will allow to increase software reliability fundamentally, *i.e.*, reduce the chance of software errors to less than 10^{-10}.

Therefore, we propose a novel research agenda, revolving around the *paradigm of software intents*. This agenda should address at least the following areas:

- *Expression of Intent:* How to discover and define software intents?
- *Verification of Intent:* How to ensure that software systems *always* behave as intended?
- *Evolution of Intent:* How to ensure that software systems keep behaving as intended, also in the future?

While continuous change, reliability and trustworthiness regard any type of software system, they are especially critical in embedded software where intents derived from safety, ethics and performance concerns play an essential role. Therefore, this research agenda can only be effective in close collaboration with a

selected number of embedded software application domains. Clear candidate domains are robotics, in industry, autonomous automotive, or home care situations, other health care applications, and science experiments in physics and biology. Here, reliability, safety, ethics, and performance play a prominent role.

3.1 Expression of Intent

The first challenge is *how to discover the implicit intents* of software systems. Generally, people have all kinds of expectations on software, but are incapable of making that precise. What are the fundamental properties that should continue to hold throughout the lifetime of a software system, no matter how it evolves? And how can we be sure that all relevant intents have been formulated? Intents might be specified upfront, but another interesting challenge is to discover intents for existing (legacy or open source) systems by observing their use, or mining user or developer data like logs, chats, code repositories, bug trackers, test suites, etc.

A particular interesting question is whether different intents can be *combined* or when intents are *conflicting*. In case of conflicting intents, we need to understand how to resolve these conflicts, which quickly becomes an *ethical question*, as it requires prioritizing one intent over the other. For example, in principle all user data should remain private, but if the user is a suspected terrorist, the privacy restriction might be released in favor of the society's security.

The next challenge is *how to express software intents*. There can be many sources of intents: they can be linguistic expressions of intent by human developers or users; they can be legal restrictions, imposed by laws on safety, privacy and security; or formal specifications, models, test cases etc. Thus, intents may be formal, but they do not have to be. How should suitable *languages of intent*, covering different classes of intents, be defined? For example, as a language of intent for a specific application domain, or as a language of intent for a particular class of properties? Rather than requiring a single language, we stress the usability and understandability of the languages, so that when an intent is expressed, all stakeholders have a common understanding of how the software should behave. Thus, these languages of intent will provide a common framework for all developers and users of software to express what they believe to be the intended program's behavior. Moreover, the advantage of having a well-defined language is that its underlying *semantics of intents* can be defined precisely and formally. This makes it possible to demonstrate that intents are compatible, and ultimately that software satisfies its intent.

3.2 Verification of Intent

There are many different ways to verify intents on software. This can be done at *design time* using logic-based techniques, which can provide high-correctness guarantees. It is possible to do this by testing applications in a *software laboratory* setting before deployment. But it can also be done after deployment,

by *monitoring applications* in real-life situations in the specific environment in which the software is run.

Design-time verification (or static verification), is done on the basis of models of the program code or on the code itself. Instead of executing the program, it analyses all the possible program executions (for example, by exploring the complete state space of a program, or by representing program executions at a suitably abstract level in a symbolic way). Over the last decade, significant progress has been made to apply design-time verification to large-scale examples [27, 53, 69]. However design-time verification still requires a significant amount of user intervention. For design-time verification of intent-based software, the major research challenges that need to be addressed are:

1. to *fully automate* this process by automatically generating all the information that is necessary to capture all possible program behaviors in a suitable way;
2. to investigate how design-time verification can be *generalized* to other classes of intents than traditional functional specifications; and
3. to apply design-time verification when one only has control over parts of the software system only.

Testing, or run-time verification, considers concrete program executions, and verifies in a software laboratory whether software does not violate the intended behavior of the program. Because of the lab situation, one has full control over the program, and can rerun it, applying smart (or dangerous) test cases. The main challenges are:

1. to guide the testing process to test cases that might lead to discovering potential intent violations, or can support certifying their absence;
2. while at the same time optimizing the time and effort put in the testing process, possibly taking into account a suitable risk assessment.

Finally, **monitoring** is needed to verify that the software fulfils its intent even after software deployment. Verification and testing typically assume a particular environment in which the software is run, but after deployment the environment might change in unforeseen ways. Monitoring provides a first step, when a violation of intent is detected or predicted, to adapt the behavior of the system, or if necessary to completely interrupt it before any real harm can be done. The major challenges are to understand how the monitoring can be done effectively for all classes of intents:

1. without significant *overhead on the performance* of the software; and
2. in the presence of *great uncertainty* due to the dynamic environment in which the software is running.

3.3 Evolution of Intent

A fact of life (which makes it interesting) is that both the intents that software should satisfy as well as the environment in which it must run are continuously

changing. If the environment changes, the software should live up to its intents, and if needed, adjust itself automatically. In case the intent itself changes, for example because of a change in user requirements, or a new law imposing new rules that the software must satisfy, the software has to be adjusted to live up to its intents. In traditional software engineering, the software has to be re-engineered. Because of the enormous size of software this is a costly, error prone and time consuming affair. Avoiding this requires novel techniques such that the software can correct itself autonomously, and then provide guarantees that it behaves according to its (new) intent again. Exciting new research shows that it is possible to achieve this on small programs, with search-based techniques, [58, 73]. However, turning this into a scalable technique will require a major effort.

The first challenge is to investigate how software can *establish* itself that it is still fulfilling its intents, especially when there are changes in the environment. The software *should take responsibility for its own intent.* The next challenge is to understand the different techniques for *efficient correction* of the software in case the intents are violated. Due to the size and complexity of software, repairing software at one place can have an unwanted and unexpected impact on other parts of its behavior. Efficient methods are required, to show that local software reparations do not adversely affect the rest of the code. The challenge is to show that the intent is fulfilled again without a complete re-verification. Finally, the last major challenge is to understand if and how these correction techniques can be applied autonomously, i.e., to make the software *self-correcting.* For this, techniques are required that select the appropriate corrections for the diagnosed source of the problem automatically, and then apply these corrections.

Just as modern spellcheckers effectively apply auto-correction to natural language texts by pattern recognition, in our vision future software systems should

1. *detect software patterns* that violate their own intents; and
2. *identify and apply correction patterns automatically* to make software respect its intents again.

4 Current State-of-the-Art

The quest for reliable software is not new, but started right after the conception of programmable machines. In this section, we provide a broad overview of the current state of the art on the topics covered by the proposed research agenda.

4.1 Expression of Intent

Requirements engineering is traditionally used to understand and model the software's intended behavior. Requirements engineering typically focuses on identifying, modeling and analyzing the goals of the stakeholders [90]. Usually, every stakeholder has a different perspective on what the software should do [37, 79]. These *viewpoints* can be independent or partially overlapping. Various proposals exist to resolve the inconsistencies between viewpoints [36, 5]. A common approach is to prioritize the requirements.

A systematic approach to incorporate ethical and legal questions into requirement prioritization is still largely missing. From a legal perspective, the formulation of norms and regulations on software is slowly gaining traction, especially in the field of security and privacy [15, 54]. Recently, [24] formalized intents about a wide range of security requirements. Lago et al. started a library of intents in the domain of cloud-based software [77, 61]. Despite their long history, the adoption of goal modelling techniques is still low. Software requirements are often written in natural language [91], so they cannot be linked directly to the other activities of software development.

At the other side of the spectrum, we have formal languages. For instance, modal logics specify external system behaviour [8, 48]. They operate on idealized system models rather than on actual code. For program verification, several program annotation languages have been introduced [51, 59]. These approaches are precise, but limited in expressivity. They are based on formal semantics, detailing the intended behavior of substantial parts of widely used programming languages, like C [60, 34] and sequential and concurrent Java [52, 6]. Other formal specification notations, like B and VDM, operate at the system level [3, 38].

Formal notations are precise and often very expressive, but hard to use, even by insiders. Domain-specific languages (DSLs) [87, 40] provide a good middle ground between informal natural language and low-level formal logics. DSLs provide domain-specific notation, analysis, verification, and optimization. A key complication is the cost of their implementation and maintenance, as studied by Van Deursen et al. [85]. This can be alleviated by the development of programming environments for the domain-specific programming languages [39, 35]. A limitation of the current generation of language workbenches is the lack of support for guaranteeing the safety and reliability of languages, which is crucial for the software intent paradigm. This requires integrating verification of language properties in language workbenches [92, 68, 75].

4.2 Verification of Intent

In industry, testing is currently the prevalent way to discover faults, and to assess the quality of software. It became the most expensive part of system development, calling for test automation to reduce the costs. However, testing is poorly understood: while testing does reveal many software faults, it is currently unknown how much testing is sufficient to gain some justified level of confidence. Also, testing is often focused at small components (unit testing) while the actual challenges arise at system integration. An advanced test automation technique for control-intensive software is model-based testing [83]. This provides a sound and complete theory for deriving, executing, and evaluating test cases from a model of a system. However, systematic research on computing the next best optimal test case is still missing. For data generation, search-based testing has been investigated [80], which uses meta-heuristic search for steering random test suites towards optimizing a test fitness function. For both technologies, there is a gap between the intents of the system and the used test models.

Another challenge is to establish the likelihood that software satisfies its intent after testing, or to generate optimal test cases that lower that likelihood. This challenge is tough. An overview of the field is presented in [98], providing nice and intricate theoretical testing models. Aligning these models with practical observation is a big challenge. Viewing testing as a learning activity is promising, but not yet widespread. It is most prominent in the field of *exploratory testing* [50, 95]. Our notion of intent could be used to record the knowledge obtained by exploratory testing. An algorithmic interpretation is provided by automaton learning [19, 63, 64, 30], which has been used complementary to testing [1].

Automated verification of software at design-time raises confidence in program correctness. Its two fundamental pillars are modularity and automated reasoning. *Modularity* forms the cornerstone to scale up verification technology to full systems. It is rooted in proper abstractions of program behavior. In his seminal work, Robin Milner [66] introduced bisimulation as the appropriate equivalence notion. This allowed to reason about the abstract behavior of a program in its wider context. Hoare logic [51] provides modularity at the level of program code. Pre- and post-conditions abstract away from the implementation of software functions. Other parts of the program need to deal with their specification only.

Automated reasoning has been revolutionized by SAT solving. Although SAT solving is NP-hard, a series of breakthroughs [25, 17, 81, 49, 33] led to algorithms that allowed to solve instances of propositional formulas of tremendous size. Many problems have been translated into SAT solving. An early example is the verification of PLC programs by Groote [47] that guard the safety of railway yards, bridges and sluices. More recently, SAT solvers were extended with mathematical theories, leading to SMT solvers (satisfiability modulo theories), like Z3 and Mathsat [29, 21], reasoning about data types automatically.

The three main branches in automated program verification are: (1) *Interactive proof checkers.* The Dutch mathematician De Bruijn observed in 1967 that proof checking could be mechanized. He built Automath [26], the first program to check mathematical proofs. Modern proof checkers, like Coq [11], follow the same principle. The user constructs a proof interactively, which is checked by an independent procedure, to eliminate all reasonable doubt on the validity of a theorem. This approach is versatile, but not feasible for non-trivial software, since a correctness proof is extremely detailed.

(2) *Model checkers* [8], invented by the Turing Award winners Clarke, Emerson and Sifakis, operate fully automatically in principle. They take a (model of a) program and a specification, and check the property by brute force computation. One of the most expressive model checkers is the mCRL2 toolset [48]. Tremendous progress was made by automated abstraction techniques [22, 14], the use of BDD and SAT-solving technology [12], and the use of high-performance graph algorithms for model checking, e.g. [55]. Model checkers operate on finite systems of limited size only; checking realistic software with datatypes and concurrency remains a challenge.

(3) *Automated program verifiers*, like OpenJML and KeY [23, 4] strike a balance between expressivity and automation. They directly operate on software programs, generating a sufficient set of verification conditions, which are automatically discharged by SMT solvers. Non-trivial programs can be handled, which is demonstrated by revealing an error in the popular TIMsort routine in the Java library with KeY [27]. Extending Hoare Logic [51] by Separation Logic [32] allowed handle concurrency and data structures of modern software. This led to tools that can reason about parallel Java programs, like the VerCors tool [13]. Still, program verifiers can only be used by verification experts, since they require manual annotations of pre- and post-conditions, as well as auxiliary properties like invariants and thread permissions. This doesn't scale beyond medium size programs.

Monitoring deployed software is common to all programming languages and even computer hardware. An example is checking memory accesses, to prevent violation of the intent that programs should access only their own memory. This can be supported by hardware, guaranteed by the Operating System, and is built-in in many modern programming languages (like Java and C#).

In the security context, the versatility and unpredictability of attacks on software make it hard to verify against *a priori* defined intent. Besides monitoring violation of well-defined security intents, software is often monitored globally and any unusual activity (such as extreme network activity) is reported for further investigation. Whenever a security breach is detected, this is generally transformed into an explicit intent that can be checked. Advanced security monitoring techniques are pioneered by Bos et al [82, 84]. Likewise, Lago's team has pioneered monitoring of sustainability intents [76, 78, 57].

In the run-time verification community, attention has shifted to the dynamic verification of behavioural properties on programs [10, 31]. These techniques are often limited to safety properties, which guarantee absence of malbehavior. Like user-inserted asserts, all these forms of monitoring typically lead to a stop-the-world whenever some intent violation is detected.

4.3 Evolution of Intent

Programming by contract [65] emerged as a systematic approach. When the software intent or the environment change, it can be checked whether the environment still offers sufficient functionality, such that the software can still guarantee its renewed contract. To establish that the software itself ends up undamaged in core memory, traditional methods are self-tests, checksums and oblivious hashing. These can be circumvented, so security researchers introduced tamper-proof software, which relies on encryption [97, 7].

To localize faults, spectrum-based fault diagnosis attributes observed failures to the responsible software components [2, 74]. A different perspective is origin tracking [86], where the contribution of a certain source element to some external effect is carefully traced. A promising new localisation approach is proof carrying code [67, 9]. Here, the program correctness proof is encoded within the software.

If the intent changes, proof reconstruction might point to the relevant parts of the proof and program that must be changed.

There are a number of fields where returning to a safe situation has been investigated in depth. Examples are file storage [93], database transactions [56] and networking [18]. Generally, such techniques rely on external operators to restart systems and bring them back in a working state. A whole research line on software engineering considers self-* systems (e.g. self-adaptation, self-management, self-healing, self-configuration). Here a system must preserve its operation at run-time even in the presence of changes in environment, resources, security breaches, faults [28]. Solutions come from various areas, like software architecture, fault-tolerance, and bio-inspired computing [94, 16, 72]. These techniques typically protect against environment uncertainty.

If the software contains errors, or when the intents themselve change, software must be updated. There are several techniques to update software while it is running (hot swappable). Reliable dynamic updates and error recovery are studied in [45, 44]. There is little systematic research into the question how software can handle its own mishap completely autonomously. Only very recently, some indications that self-correcting code might work were published [43, 58, 73]. The question is whether these observations carry over to software on a larger scale. In particular, it will be challenging to compute the best update on the software that removes the problem.

The fundamental challenge in self-correcting code lies in guaranteeing that code adapts itself in such a way that it will continue to behave in the right way, even if the intent has not been formulated explicitly and completely. In the past, various forms of self-adaptation of software have been used, generally with an adverse effect. An excellent example is the programming language COMAL [46], which would correct simple programming errors by itself. Superficial typos were sometimes transformed in deep, hard to spot programming errors, undoing the beneficiary effect of type checking.

5 Perspectives

We conclude by summarizing our perspective on achieving software that meets its intents. In particular, we identify technical challenges that will need further research in the next decade.

First, intents must be discovered and expressed. The goal is that all stakeholders can express, understand, and prioritize their expectations on software. Domain specific languages should bridge the enormous gap between human intent and low-level machine readable code.

Second, intents must be verifiable, both statically during design (verification), dynamically in a laboratory setting (testing), and on deployed code (monitoring). The big challenge is to make an automation leap: human effort to annotate or instrument programs, construct proofs, derive test cases, or operate verification tools should be avoided. Technically, this requires an advance of annotation

and invariant generation. At the same time, verification should bridge the gap between detailed code and its high-level, emerging properties.

Finally, software should correct itself when it contains errors, when its environment changes, or when the intents of people or organizations evolve. Current runtime adaption and software update technology should be complemented by an auto-correction facility for code. Building in explicit and executable notions of intent make it possible in principle that software adapts itself to its intent.

These technical developments require the coordinated effort of researchers in theoretical computer science, programming languages, and empirical software engineering. When successful, they could form the basis for intents as software laws that can be verified and guaranteed. Only automatically self-correcting software will enable us to regain grip on the complex, evolving software systems that run our society. Ultimately, this development allows that software providers can be held accountable for their product.

References

1. F. Aarts, H. Kuppens, J. Tretmans, F. W. Vaandrager, and S. Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1-2):189–224, 2014.
2. R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. A new Bayesian approach to multiple intermittent fault diagnosis. In *International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 653–658, 2009.
3. J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
4. W. Ahrendt, B. Beckert, R. Hähnle, P. Rümmer, and P. H. Schmitt. Verifying object-oriented programs with KeY: A tutorial. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, (FMCO 2006)*, volume 4709 of *LNCS*, pages 70–101. Springer, 2006.
5. R. Ali, F. Dalpiaz, and P. Giorgini. Reasoning with contextual requirements: detecting inconsistency and conflicts. *Information and Software Technology*, 55(1):35–57, 2013.
6. A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11, 2014. paper 2.
7. D. Andriesse, H. Bos, and A. Slowinska. Parallax: Implicit code integrity verification using return-oriented programming. In *IEEE/IFIP IC on Dependable Systems and Networks, DSN'15*, pages 125–135. IEEE Computer Society, 2015.
8. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
9. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. *ACM Trans. on Programming Languages and Systems*, 31(5), 2009.
10. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.
11. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

12. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
13. S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In *Formal Methods*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
14. A. R. Bradley. IC3 and beyond: Incremental, inductive verification. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*. Springer, 2012.
15. T. D. Breaux, M. W. Vail, and A. I. Antón. Towards regulatory compliance: Extracting rights and obligations to align requirements with regulations. In *IEEE International Requirements Engineering Conference*, pages 46–55, 2006.
16. Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.
17. R. E. Bryant. Symbolic manipulation of Boolean functions using a graphical representation. In H. Ofek and L. A. O'Neill, editors, *22nd ACM/IEEE conference on Design automation, (DAC 1985)*, pages 688–694. ACM, 1985.
18. C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
19. S. Cassel, F. Howar, B. Jonsson, and B. Steffen. Active learning for extended finite state machines. *Formal Asp. Comput.*, 28(2):233–263, 2016.
20. T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *11th Working Conference on Mining Software Repositories*, MSR 2014, pages 82–91. ACM, 2014.
21. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - TACAS'13*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
22. E. M. Clarke, A. Gupta, and O. Strichman. SAT-based counterexample-guided abstraction refinement. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.
23. D. R. Cok. OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. In C. Dubois, D. Giannakopoulou, and D. Méry, editors, *1st Workshop on Formal Integrated Development Environment, (F-IDE 2014)*, volume 149 of *EPTCS*, pages 79–92, 2014.
24. F. Dalpiaz, E. Paja, and P. Giorgini. *Security Requirements Engineering: Designing Secure Socio-Technical Systems*. MIT Press, 1 edition, 2016.
25. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
26. N. de Bruijn. A survey of the project AUTOMATH. In *To H.B. Curry: Essays in combinatory logic, lambda calculus and formalism*, pages 579–606. Academic Press, 1980.
27. S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification (CAV 2015)*, LNCS 9206, pages 273–289. Springer, 2015.
28. R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. *Software engineering for self-adaptive systems: A second research roadmap*. Springer, 2013.
29. 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, (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

30. A. v. Deursen. Testing web applications with state objects. *Commun. ACM*, 58(8):36–43, July 2015.
31. V. Diekert and M. Leucker. Topology, monitorable properties and runtime verification. *Theor. Comput. Sci.*, 537:29–41, 2014.
32. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems(TACAS 2006)*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
33. N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, SAT'03.*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
34. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.
35. S. Erdweg, T. van der Storm, M. Vlter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. K. 0001, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.
36. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE TSE*, 20(8):569–578, 1994.
37. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *Intl. J. of Softw. Eng. and Knowledge Eng.*, 2(1):31–57, 1992.
38. J. Fitzgerald and P. G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development*. Cambridge University Press, 2009. 2nd edition.
39. M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. http://www.martinfowler.com/articles/languageWorkbench.html.
40. M. Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.
41. A. Ganapathi and D. A. Patterson. Crash data collection: A windows case study. In *DSN*, pages 280–285. IEEE Computer Society, 2005.
42. Gartner Inc. Smart cities will include 10 billion things by 2020. https://www.gartner.com/doc/3004417/smart-cities-include–billion, 2015.
43. W. Ghardallou, N. Diallo, and A. Mili. Program derivation by correctness enhancements. In *Refinement 2015*, 2015.
44. C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. Practical Automated Vulnerability Monitoring Using Program State Invariants. In *DSN*, Oct. 2013.
45. C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *LISA*, Oct. 2013.
46. I. Gratte. *Starting with COMAL*. Englewood Cliffs: Prentice-Hall, 1985.
47. J. Groote, J. Koorn, and S. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd (extended abstract). In *10th Annual Conference on Computer Assurance (COMPASS'95)*, pages 57–68, 1995.
48. J. Groote and M. Mousavi. *Modeling and analysis of communicating systems*. The MIT Press, 2014.
49. J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. *J. Autom. Reasoning*, 24(1/2):101–125, 2000.
50. E. Hendrickson. *Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing*. The Pragmatic Bookshelf, 2013.

51. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
52. M. Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
53. Y. Hwong, J. Keiren, V. Kusters, S. Leemans, and T. Willemse. Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider. *Science of Computer Programming*, 78:2435–2452, 2013.
54. S. Ingolfo, A. Siena, and J. Mylopoulos. Establishing regulatory compliance for software requirements. In *Conceptual Modeling–ER 2011*, pages 47–61. Springer, 2011.
55. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-performance language-independent model checking. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15*, LNCS 9035, pages 692–707. Springer, 2015.
56. B. Kemme, R. Jiménez, and M. Patiño-Martinínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool publishers, 2010.
57. P. Lago, S. A. Koçak, I. Crnkovic, and B. Penzenstadler. Framing sustainability as a property of software quality. *Communications of the ACM*, 58(10):70–78, 2015.
58. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.
59. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007. Dept. of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.
60. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52:107 – 115, 2009.
61. G. A. Lewis, P. Lago, and P. Avgeriou. A decision model for cyber-foraging systems. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA 2016)*, pages 51–60. IEEE, 2016.
62. R. Matias, M. Prince, L. Borges, C. Sousa, and L. Henrique. An empirical exploratory study on operating system reliability. In *29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1523–1528. ACM, 2014.
63. A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1), 2012.
64. A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automated testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, 2012.
65. B. Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer, 2009.
66. R. Milner. *Calculus of communicating systems*. Number 92 in Lectures in Computer Science. Springer-Verlag, 1980.
67. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, 1997.
68. P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.
69. A. Osaiweran, M. Schuts, J. Hooman, J. Groote, and B. van Rijnsoever. Evaluating the effect a lightweight formal technique in industry. *International Journal on Software Tools for Technology Transfer*, 18:93–108, 2016.

70. T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64. ACM, 2002.
71. T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 86–96. ACM, 2004.
72. T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 33–42. IEEE Press, 2012.
73. Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.
74. A. Perez, R. Abreu, and A. van Deursen. A unifying metric for test adequacy and diagnosability. In *Automated Software Engineering*, 2016. Submitted.
75. C. B. Poulsen, P. Neron, A. P. Tolmach, and E. Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July, 2016, Rome, Italy*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. to appear.
76. G. Procaccianti, H. Fernández, and P. Lago. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems Software*, 117:185–198, 2016.
77. G. Procaccianti, P. Lago, and G. A. Lewis. A catalogue of green architectural tactics for the cloud. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA'14)*, pages 29–36. IEEE, 2014.
78. G. Procaccianti, P. Lago, A. Vetro, D. M. Fernández, and R. Wieringa. The green lab: Experimentation in software energy efficiency. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, 2015.
79. N. Rozanski and E. Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.
80. K. Salvesen, J. P. Galeotti, F. Gross, G. Fraser, and A. Zeller. Using dynamic symbolic execution to generate inputs in search-based GUI testing. In G. Gay and G. Antoniol, editors, *8th IEEE/ACM International Workshop on Search-Based Software Testing, SBST'15*, pages 32–35. IEEE, 2015.
81. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.
82. A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of USENIX Annual Technical Conference*, Boston, MA, June 2012.
83. J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.
84. V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, USA, May 2016. IEEE.
85. A. van Deursen and P. Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.
86. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15(5/6):523–545, 1993.

87. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.
88. M. van Genuchten and L. Hatton. Metrics with impact. *IEEE Software*, 30:99–101, Jul/Aug 2013.
89. M. van Genuchten and L. Hatton. When software crosses a line. *IEEE Software*, 33:29–31, Jan/Feb 2016.
90. A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 5–19, 2000.
91. A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
92. E. Visser, G. Wachsmuth, A. P. Tolmach, P. Neron, V. A. Vergu, A. Passalaqua, and G. D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In A. P. Black, S. Krishnamurthi, B. Bruegge, and J. N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014.
93. W. von Hagen. *UNIX Filesystems: Evolution, Design, and Implementation*. SAMS, 2002.
94. D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad. A survey of formal methods in self-adaptive systems. In *Proceedings of the IC on Computer Science and Software Engineering*, pages 67–79. ACM, 2012.
95. J. A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley, 2009.
96. A. Witze. Software error doomed Japanese Hitomi spacecraft. *Nature*, 533, 2016.
97. G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE Computer Society, 2005.
98. S. Yamada. *Software reliability modeling*. Springer, 2014.