Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools

Reiner Hähnle[1(✉)] and Marieke Huisman[2]

[1] Department of Computer Science, Technische Universität Darmstadt, 64295 Darmstadt, Germany
haehnle@cs.tu-darmstadt.de

[2] Faculty EEMCS, Formal Methods and Tools, University of Twente, 7500 AE Enschede, The Netherlands
M.Huisman@utwente.nl

Abstract. Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, possibly complex properties, where the verification process is based on logical inference. We follow the trajectory of the field from its inception in the late 1960s via its current state to its promises for the future, from pen-and-paper proofs for programs written in small, idealized languages to highly automated proofs of complex library or system code written in mainstream languages. We take stock of the state-of-art and give a list of the most important challenges for the further development of the field of deductive software verification.

1 Introduction

Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, possibly complex properties, where the verification process is based on some form of logical inference, i.e., "deduction". In this article we follow the trajectory of the field of deductive software verification from its inception in the late 1960s via its current state to its promises for the future. It was a long way from pen-and-paper proofs for programs in small, idealized languages to highly automated proofs of complex library or system code written in mainstream programming languages. We argue that the field has reached a stage of maturity that permits to use deductive verification technology in an industrial setting. However, this does not mean that all problems are solved. On the contrary, formidable challenges remain, and not the smallest among them is how to bring about the transfer into practical software development. Hence, the second contribution of this article is to present an overview of what we consider the most important challenges in the area of deductive software verification.

To render this article feasible in length (and to avoid overlap with other contributions in this volume) we focus on *contract-based, deductive verification of*

B. Steffen and G. Woeginger (Eds.): Computing and Software Science, LNCS 10000, pp. 345–373, 2019.
https://doi.org/10.1007/978-3-319-91908-9_18

imperative and object-oriented programs. Hence, we do not discuss model checking, SMT solvers, general proof assistants, program synthesis, correctness-by-construction, runtime verification, or abstract interpretation. Instead, we refer to the articles *Runtime Verification: Past Experiences and Future Projections*, *Software Architecture of Modern Model-Checkers*, *Statistical Model Checking*, as well as *The 10,000 Facets of MDP Model Checking* in this issue. We also do not cover fully automated verification tools for generic safety properties (see the article *Static Analysis for Proactive Security* in this issue for some aspects on these). This is not at all to say that these methods or tools are unimportant or irrelevant. On the contrary, their integration with deductive verification appears to be highly promising, as we point out in Sect. 5.2 below.

This paper is organized as follows: in the next section we walk through a non-trivial example for contract-based verification to clarify the scope and illustrate some of the important issues. In Sect. 3 we sketch the main developments in the field ca. up to the year 2000. In Sect. 4 we sketch the current state-of-art and we discuss the two main approaches to deductive verification: symbolic execution and verification condition generation. The core of the paper is Sects. 5 and 6, where we discuss the main achievements and the remaining challenges of the field, divided into technical and non-technical aspects. We conclude in Sect. 7.

2 An Example

Properties to be proven by deductive verification are expressed in a formal specification language. Ada was the first language that supported expressing formal specification annotations directly as structured comments next to the program elements they relate to [97]. As this proved to be natural and easy to use, this was followed for other programming languages. Eiffel [99] propagated a *contract-based* paradigm, where the prerequisites and obligations of each method[1] are laid down in a contract. This has the very important advantage that methods, as the central abstraction concept to structure a program, have a direct counterpart in formal specifications. Hence, specifications and programs follow the same structure. For most major imperative/object-oriented programming languages there exist dedicated contract-based annotation languages (see Sect. 5.1).

We give an example of contract-based formal specification and verification of a Java program with the Java Modeling Language (JML) and provide informal explanations of JML specification elements; more details are in [66,85]. Consider the Java method `search()` in Fig. 1 which implements binary search in a sorted integer array. Its code is completely specified, so it can be compiled and run from a suitable `main()` method.

The method contract (lines 1–7) specifies the intended behavior, whenever `search()` terminates normally. The contract's only requirement (line 2) is that the input array is sorted (in JML all reference types are assumed to be non-null by default, so this does not need to be spelled out). Sortedness is specified with the help of a *model method* `isSorted()` that is not shown. The contract

[1] We use Java terminology for what is also called procedure, function, subroutine, etc.

```
1  /*@ public normal_behavior
2  @   requires isSorted(a);
3  @   ensures ((\exists int x; 0 <= x && x < a.length; a[x] == v) ?
4  @             \result >= 0 && \result < a.length && a[\result] == v :
5  @             \result == -1);
6  @   assignable \strictly_nothing;
7  @*/
8  public static /*@ pure @*/ int search(int[] a, int v) {
9    int l = 0;
10   int r = a.length - 1;
11
12   if      (a.length == 0) { return -1; }
13   else if (a.length == 1) { return a[0] == v ? 0 : -1; }
14   /*@ loop_invariant 0 <= l && l < r && r < a.length
15   @           && (\forall int x; 0 <= x && x < l;        a[x] < v)
16   @           && (\forall int x; r < x  && x < a.length; v < a[x]);
17   @ assignable \strictly_nothing;
18   @ decreases r - l;
19   @*/
20   while(r > l + 1) {
21     int mid = l + (r - l) / 2;
22     if      (a[mid] == v) { return mid; }
23     else if (a[mid] > v)  { r = mid; }
24     else                  { l = mid; }
25   }
26   if(a[l] == v) return l;
27   if(a[r] == v) return r;
28   return -1;
29 }
```

Fig. 1. Formal JML specification of a Java binary search method

says that whenever `search()` is called with a sorted, non-null array then the call terminates and in the final state the property given in the ensures clause (lines 3–5) is satisfied. In addition, the assignable clause (line 6) says that the execution has strictly no side effects, not even creation of new objects. The contract is valid for *any* input of unbounded size that satisfies the requirements.

We take a closer look at the ensures clause: line 3 is the guard of a conditional term saying that the value `v` occurs as an entry of `a`. If true, an array index where `v` is found is returned as the result, and `-1` otherwise. We do not specify whether the result is the smallest index, but make sure that is in a valid range.

The loop invariant (lines 14–16) specifies the valid range of the pivots and says that `v` can never occur below index `l` or above `r`. To ensure termination of `search()` it is sufficient to ensure termination of the loop. This is achieved by the decreases clause (line 18), an expression over a well-ordered type that becomes strictly smaller in each iteration.

A central advantage of contract-based verification is compositionality and scalability: after showing that a method satisfies its contract, each call to that method can be replaced with its contract, instead of inlining the code. Specifically, if the callee's requires clause is satisfied at the call point, then its ensures clause can be assumed and the values of all memory locations of the caller, except the assignable ones, are preserved. We illustrate the idea with a simple client method, see Fig. 2.

```
1  //@ public invariant next >= 0;
2  private /*@ spec_public @*/ int[] indices;
3  private /*@ spec_public @*/ int next;
4
5  /*@ public normal_behavior
6  @ requires isSorted(a) && a != indices && next < indices.length;
7  @ ensures (\exists int i; i >= 0 && i < a.length; a[i] == v) ?
8  @         indices[\old(next)] == \result :
9  @         (next == \old(next) && indices[next] == \old(indices[next]));
10 @ ensures (\exists int i; i >= 0 && i < a.length; a[i] == v) ?
11 @         a[\result] == v  : \result == -1;
12 @ assignable indices[next], next;
13 @ also
14 @ public exceptional_behavior
15 @ requires isSorted(a);
16 @ requires (\exists int i; i >= 0 && i < a.length; a[i] == v);
17 @ requires next >= indices.length;
18 @ signals (ArrayIndexOutOfBoundsException) true;
19 @ assignable \nothing;
20 @*/
21 public int addIndex(int[] a, int v) {
22   int idx = search(a,v);
23   if (idx >= 0) {
24     indices[next] = idx;
25     next++;
26   }
27   return idx;
28 }
```

Fig. 2. Formal JML specification of a Java client method

Method `addIndex()` searches for value `v` in `a`. If the entry was found at index `idx`, then it is appended to the array contained in the field `indices` and returned. The specification is surprisingly complex. First of all, as specified in the exceptional termination case (lines 14–19), an `ArrayIndexOutOfBoundsException` is thrown if the array `indices` is full and the given value is found (line 16): in this case the array has to be extended (line 17). The assignable clause (line 19) is not strict, because a new exception object is created. Sortedness of parameter `a` is necessary to ensure that the contract of `search()` can go into effect.

The specification case for normally terminating behavior (lines 5–12) of `addIndex()` is similar to that of `search()`: in addition we require (line 6) that `a` and `indices` are different arrays (Java arrays can be aliased) and that there is still space for a new entry (`next < indices.length`). The latter could be weakened by disjoining the condition that the value is found. The second ensures clause (lines 10–11) is almost identical to the one of `search()` (we left out the bounds on the result). The first ensures clause (lines 7–9) says that, if the value is found then its index is appended to `indices`; otherwise, `indices` and `next` are unchanged. We use the keyword `\old` to refer to a value in the prestate. This is necessary, because `next` was updated in the method.

Typically, one also specifies class invariants that, for example, capture consistency properties of the instance fields that all methods must maintain. In JML any existing class invariants are implicitly added to all requires and ensures clauses which helps to keep them concise. In the example, we maintain the invariant that `next` is non-negative (line 1). Class invariants must be established by all constructors (not shown here).

Discussion. Even our small example shows that precise contracts, even of seemingly innocent methods, can become bulky. The specification of `addIndex()` is about twice as long as its implementation. And, as pointed out, that specification could be made even more precise. But without further information about the call context it is hard to decide whether that is useful. A subtle question is whether the annotation `\result >= 0 && \result < a.length` in line 4 of Fig. 1 is needed at all: if it doesn't hold then the expression `a[\result]` is not well-defined in Java anyway. But most verifiers will not be able to deduce this by themselves, because they treat such an expression as *underspecified*. The semantics of most tools, including the JML standard, is not always unambiguous.

It is very easy to forget parts of specifications: in most cases, the first attempt will not be verifiable. While developing the example we forgot `a != indices` in line 6 of Fig. 2, a typical omission. Good feedback from the verification tool is very important here. Vice versa, some of the specification annotations should be automatically derivable, for example, the bounds. Note that reuse of specification elements is essential to obtain concise and readable annotations.

It took about one hour (for an expert) to specify and verify the example reproduced here. After finding the correct specifications, formal verification with the system KeY [3] is fully automatic and takes about 6 s on a state-of-the-art desktop, including the constructor and model methods not displayed. The most complex method, `search()`, led to a proof tree with ca. 4,000 nodes and 27 branches. Interestingly, when we loaded the verified example in OpenJML [37], we only had to rename some KeY-specific keywords such as `\strictly_nothing` (replaced by `\nothing` , and then most of the example could be verified directly. The only specification that could not be verified was the exceptional behavior specification of method `addIndex`, as OpenJML adds extra proof obligations to every array access instruction, capturing that the index should be between the bounds of the array, to ensure the absence of runtime errors.

3 History Until LNCS 1750 (aka Y2000)

The Roots of Deductive Verification. The history of deductive software verification dates back to the 1960s and 70s. Seminal work in this area is Floyd-Hoare logic [48,61], Dijkstra's weakest preconditions [42], and Burstall's intermittent assertions [31].

Floyd and Hoare introduced the notion of pre- and postcondition to describe the behaviour of a program: a Hoare triple $\{P\}S\{Q\}$ is used to express that if program S is executed in an initial state σ, such that the precondition P holds for σ, then if execution of S terminates in a state σ', the postcondition Q holds for the final state σ'. This relation is also called *partial correctness* (partial, because termination is not enforced). Any pair of states (σ, σ') for which the Hoare triple holds, must be contained in the big-step semantics [110] of S. Floyd and Hoare proposed a set of syntactic proof rules to prove the correctness of an algorithm. One classical example of such a proof rule is the rule for statement composition:

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

This rule expresses that to prove that if $S_1; S_2$ is executed in a state satisfying precondition P, if it terminates in a state satisfying Q, it is sufficient to find an intermediate assertion R, such that R can be established as a postcondition for the first statement S_1, and is a sufficient precondition for S_2 to establish postcondition Q. Rules like this break up the correctness problem of a complete algorithm into a correctness problem of the individual instructions.

Dijkstra observed that it is possible to compute the minimal precondition that is necessary to guarantee that a program will establish a given postcondition. This simplifies verification, because in this way, one does not have to "invent" the intermediate predicate that describes the state between two statements, but this can be computed. In particular, the weakest precondition wp for a statement $S_1; S_2$ can be computed using the following rule:

$$\mathsf{wp}(S_1; S_2, Q) = \mathsf{wp}(S_1, \mathsf{wp}(S_2, Q))$$

For other instructions, similar rules exist. A *Verification Condition Generator* (VCG) is a deductive verification tool that produces proof obligations expressing that the specified precondition is stronger than the weakest precondition as computed by the wp rules. For this approach to work, we require the presence of loop invariants and method contracts for all methods called in the verified code, which give rise to additional proof obligations.

VCGs in essence apply wp transformation rules backwards through the target program, starting with the postcondition to be proven. However, it is also possible to verify a program in the forward direction of its control flow. Burstall [31] proposed to combine symbolic execution with induction to show that a program implies its postcondition (see also Sect. 4).

First Deductive Verification Tools. The early program verification techniques were, to a large extent, a pen-and-paper activity. However, the limitations of doing such proofs with pen-and-paper were immediately obvious, and several groups started to develop tools to support formal verification. These efforts were all isolated, and usually still required extensive user interaction. Nevertheless, the correct application of the proof rules was checked by the system, and many obvious errors were avoided this way. It is not possible to give a complete overview of early verification systems, but we mention some representative tools and their main characteristics.

Tatzelwurm [41] was a VCG for a subset of UCSD Pascal. It accepted specification annotations in sorted first-order logic and used a tableau-based theorem prover with a decision procedure for linear integer expressions to discharge verification conditions.

Higher-order logic theorem provers were frequently used to construct a verified program verifier. The soundness of the verification technique was proven inside the theorem prover, and the program to be verified was encoded in the logic of the theorem prover, after which the verified rules could be applied. This approach was used for example in the Loop project, where Hoare logic rules were formalized in PVS (later also Isabelle) to reason about Java programs [65,67], the Sunrise project, which used a verification condition generator verified in HOL for a standard while-language [64], by Von Oheimb who formalized a Hoare logic for Java in Isabelle/HOL [126], and by Norrish, who formalized a Hoare logic for C in HOL [105].

SPARK [112] and ESC [90] were among the first tools to directly implement the weakest precondition calculus. Development of SPARK started in an academic setting, was further extended and refined in an industrial setting, and is now maintained and marketed by AdaCore and Altran. SPARK realizes a VCG for (a safety-critical subset of) Ada and is still actively developed [60]. The ESC (Extended Static Checker) tool originally targeted Modula-3, but was then adapted to Java [88]. ESC was designed with automation in mind: it traded off correctness and completeness with the capability to quickly identify possible problems in a program, thus providing the programmer with useful feedback.

Another early implementation of the weakest precondition calculus was provided in the B Toolkit [113] that realized tool support for the B Method [1]. The B Method is based on successive refinement of a sequence of abstract state machines—weakest precondition reasoning is used to establish invariants, preconditions, and intermediate assertions for a state machine. The B Method is one of the industrially most successful formal methods (see [93] for an overview), however, it is not a deductive software verification approach and, for this reason, not discussed further.

The KIV system [51] was the first[2] interactive program verifier based on dynamic logic, an expressive program logic that can be viewed as the syntactic closure of the language of Hoare triples with respect to first-order connectives

[2] The first verification system based on dynamic logic is reported in [95], but it was based on an axiomatic calculus and had no further impact.

and quantifiers [54]. It formalizes Burstall's [31] approach as a dynamic logic calculus whose rules mimic a symbolic interpreter [55]. Induction rule schemata permit complete symbolic execution of loops. KIV is still actively being developed, and much effort has been put into automation, and an expressive specification language, using higher-order algebraic specifications [45]. It has been used for verification of smart card applications and the Flashix file system.

ACL2 (A Computational Logic for Applicative Common Lisp) is a program verification tool for Lisp [78]. As other members of the Boyer-Moore family of provers, it has a small trusted core, and all other proof rules are built on top of this trusted core and cannot introduce inconsistencies. Its main proof strategies are based on induction and rewriting. The ACL2 prover is actively developed. It has been used to verify properties of, for example, models of microprocessors, microcode, the Sun Java Virtual Machine, and operating system kernels.

STeP, the Stanford Temporal Prover, used a combination of deductive and algorithmic techniques to verify temporal logic properties of reactive and real-time systems. It features a set of verification rules which reduce temporal properties of systems to first-order verification conditions and implements several techniques for automated invariant generation [19].

4 From LNCS 1750 to LNCS 10000

A Deductive Verification Community. After the year 2000, we see a gradual change from tools developed in isolation to a community of deductive software verification tool developers and users. Within this community, there is active exchange and discussion of ideas and knowledge. Effort has been put into standardizing specification languages, notably JML, now used by most contemporary tools aiming at verification of Java. Further, the VS-Comp and VerifyThis[3] program verification competitions have been established, where the developers and users of various deductive verification tools are challenged to solve program verification competition problems within a limited time frame [68]. After the competition, participants present their solutions to each other, which leads to substantial cross-fertilization.

Deductive Verification Architectures. As mentioned above, there are two main approaches for the construction of deductive verification tools: VCG and symbolic execution. Tools based on VCG use transformation rules to reduce an annotated program to a set of verification conditions whose correctness entails correctness of the annotated program. Tools that use symbolic execution collect constraints on the program execution by executing the program with symbolic variables. If the collected constraints can be fulfilled and imply the annotations at each symbolic state, then the annotated program is correct. Both approaches can be formalized within suitable program logics.

Kassios et al. [77] report that symbolic execution tends to be faster than VCG, but the former is sometimes less complete and occasionally suffers from

[3] http://www.verifythis.org/.

path explosion. However, the completeness issue seems to derive from the specific architecture of the symbolic execution tool that was used in their study, which relies on an inherently incomplete separation of heap reasoning and arithmetic SMT solving. Path explosion, however, is clearly an issue for symbolic execution of complex target code [39]. It was recently shown that it can be mitigated with symbolic state merging techniques [117].

Long-Running Deductive Verification Projects. Several tools whose development started around the year 2000 still exist currently, or evolved into new tools. We sketch the development of some of these tools.

Work on the KeY tool [3] started in 1998 [53] and it has been actively developed ever since. Like KIV, KeY is based on symbolic execution formalized in dynamic logic, but it extends the KIV approach to contract-based verification of Java programs and uses loop invariants as a specific form of induction that is more amenable to automation. KeY is not merely focused on functional verification, but complements it with debugging and visualization [3, Chap. 11] or test generation [3, Chap. 12]. It covers the complete JavaCard language [102] and was used to identify a bug in the Timsort algorithm [39], the standard sorting algorithm provided in the Oracle JDK, Python, Android, and other frameworks.

The development of ESC/Java [88] was taken over by David Cok and Joe Kiniry, resulting in ESC/Java2 [38]. Initially, their goal was to bring ESC/Java up-to-date, as well as to provide support for a larger part of JML and more Java features. ESC/Java2 is not actively developed as a separate tool anymore, however, it formed the foundation for the static verification support in the OpenJML framework [37]. Over the years, the proving capabilities of the static verification support in OpenJML have been strengthened. Like ESC/Java, it still prioritizes a high degree of automation, but soundness is not traded off anymore. OpenJML offers not merely support for static verification, but also for runtime verification.

The original ESC/Java development team around Rustan Leino moved into a different direction. In 2004, they presented Spec#, a deductive verification tool for C# [11], which reused much of the philosophy of ESC/Java. In parallel to the development of Spec#, the team also designed Boogie, as an intermediate language for static verification [10]. Boogie is a very simple programming language, for which it is straightforward to build correct verification tools. To provide support for more advanced programming languages, it is sufficient to define an encoding into Boogie. Boogie is used as the intermediate verification language for various programming languages, including Java (in OpenJML), Java bytecode [86], and C# (in Spec#). After the work on Spec# and Boogie, Leino took a slightly different approach: instead of developing a verification tool for an existing programming language, he designed Dafny, which is a programming language with built-in support for specification and verification [89], and in particular supporting dynamic frames [76].

Another widely used intermediate language is Why3 [24] which nowadays is used as a backend for SPARK 2014, the current version of SPARK/Ada [81], and Frama-C, a tool for the verification of C programs [80], specified with the JML-like language ACSL. Its original version (Why [47]) has been used as a backend

for Krakatoa [98] (for Java programs) and Jessie (for C programs). Frama-C provides more than mere deductive verification: it also supports runtime verification, and it contains analysis tools such as a slicer and a tool for dependency analysis. Much attention is given to the combination and interaction between these tools, for example how testing can be used automatically to understand why a proof fails [109]. Intermediate languages in the context of model checking are discussed in the article *Software Architecture of Modern Model-Checkers* in this issue.

A final example is the Infer tool [32], which supports fully automated deductive verification techniques to reason about memory safety properties of C programs. Infer uses separation logic, an extension of classical Hoare logic, which is especially suited to reason about pointer programs. The development of separation logic resulted in the creation of a series of research prototype tools (Smallfoot, Space Invader, Abductor) as a way to automatically analyze memory safety of programs. As the focus of Infer is on a restricted set of properties, specifications are not required (but it is possible to obtain the specs that infer derives from the analysis). Infer is integrated in the Facebook code inspection chain, and is used as one of the standard checks before code changes are committed.

All tools mentioned above have their specific strengths and weaknesses. However, they share that they target the verification of realistic programming languages, and have made substantial progress in this direction. Several of the tools mentioned above are used in undergraduate teaching (both at Bachelor and Master level). Importantly, this does not happen only at the universities of the tools' own developers, but also at other universities where lecturers find it important to teach their students state-of-the-art techniques that can help to improve software reliability.

There exist many verification case studies, where unmodified (library) code was annotated and verified, and often bugs were discovered, see e.g. [39,79,102,111,116]. Despite those success stories, there is a growing realization that post-hoc verification and, in particular, specification, remains difficult and challenging, and that there always is a trade-off between the verification effort and the level of reliability that is required for an application. A result of this realization is that we see a shift of emphasis from proving correctness of an application to bug-finding and program understanding.

5 Achievements and Challenges: Technical

5.1 Specification Languages

Deductive verification starts with specifying *what* should be verified, i.e., what behaviour we expect from the implementation. This is where the specification language comes into play.

In essence, expected program behaviour is described in the form of a method contract: a precondition specifies the assumptions under which a method may be called; a postcondition specifies what is achieved by its implementation, e.g., the

computed result, or its effect on the global state. Eiffel was the first mainstream programming language that featured such method specifications [100].[4]

Achievements. For the deductive software verification community, the design of JML, the Java Modeling Language [66,84], has been a major achievement. Figures 1 and 2 in Sect. 2 illustrate typical JML specifications. JML features method contracts, similar to Eiffel, but in addition provides support for more high-level specification constructs for object-oriented programming languages, such as class invariants, model elements, and history constraints [94]. One of the important design principles of JML is that its notation is similar to Java. Properties in JML are basically Java expressions with Boolean types, and only a few specific specification-only constructs such as quantification, and implication have been added. As a result, JML specifications have a familiar look and feel, and can easily be understood. JML is also used as a specification language for other formal validation techniques, such as test case generation, and runtime assertion checking, which further increases its usability in the software development process.

JML is a rich specification language; complex specifications can be expressed in it. It provides extensive support for abstraction in the form of a fully-fledged theory of model specification elements, based on the idea of data abstraction as introduced by Hoare [62]. The principles behind this are old, but JML turns it into a technique that can be used in practice. Abstraction allows a clear separation of concerns between specification language and implementation [33], and increases portability of specifications.

The design of JML has been influential in the design of other specification languages for deductive verification, such as the ANSI/ISO C Specification Langage (ACSL), which is used in the Frama-C project [80], and the Spec# specification language for C# [11] and its spin-off Code Contracts [96].

Challenges. A central problem of deductive verification is that specifications cannot be as declarative and abstract as one would like them, in order for verification proofs to succeed. Specifications become polluted with intermediate assertions and implementation properties that are necessary as hints for the verification engine. This becomes problematic in the verification of large code bases and is exacerbated by usage of off-the-shelf libraries. To improve the situation, we believe attention should be given to address the following two challenges:

S.1 Provide specifications for widely-used APIs. At the very least, these should describe under which circumstances methods will (not) produce exceptions. For specific APIs, such as the standard Java collection library, also functional specifications describing their intended behaviour are required. This task is work-intensive and has little (direct) scientific reward. It is, therefore, difficult to find funding to conduct the required work, see also challenge F.1.

[4] However, Eiffel contracts were intended for runtime (rather than static) verification.

S.2 Develop techniques to infer specifications from code in a (semi-)automated manner. Many specification details that have to be spelled out explicitly, actually can be inferred from the code (as illustrated in the example of Sect. 2). There is work on specification generation [63,101], but it is not integrated into deductive verification frameworks (see challenge I.9).

5.2 Integration

Integration aspects of formal verification appear on at least three levels. On the most elementary level, there is the software engineering aspect of tool integration and reuse. Then there is the aspect of integrating different methods and analyses to combine their complementary strengths. Finally, there is the challenge to integrate formal verification technology into an existing production environment such that added value is perceived by its users. We discuss each aspect in turn in the following subsections.

Tool Integration and Reuse. Software reuse is still considered to be a challenging technology in Software Engineering[5] in general. Therefore, it is not surprising that this is the case for formal verification in particular. The situation is exacerbated there, due to the complexity of interfaces and data structures.

Achievements. One success story of tool reuse in deductive verification is centered around Boogie [10] (see also Sect. 4), an intermediate specification and verification language and VCG tool chain, most often complemented by the SMT solver Z3 [40] as its backend. Boogie is a minimalist language, optimized for formal verification. It is used as a backend in several verification tool chains, including Chalice [87], Dafny [89], Spec# [11], and VCC [36]. More recently, also the intermediate verification language Silver [104], which has built-in support for permission-based reasoning, reuses Boogie as one its backends. In addition, it also comes with its own verification backend, an SE-based tool called Silicon. Interestingly, Silver in turn, is used as a backend in the VerCors platform [8] for reasoning about concurrent Java and OpenCL programs. Similarly, but with less extensive reuse, the WhyML intermediate verification language is used in the verification systems Frama-C [80] and Krakatoa [47]. Recently, a translation from Boogie to WhyML was presented [5] that links both strands. The state-of-art on tool integration in the model checking domain is discussed in the article *Software Architecture of Modern Model-Checkers* in this issue.

Challenges. Intermediate verification languages are good reuse candidates, because they are small and have a clear semantics. In addition, compilation is a well-understood, mainstream technology with excellent tool support. This makes it relatively easy to implement new frontends. On the other hand, tool reuse at the "user level", for example, for JML/Java or ACSL/C is much harder to achieve and we are not aware of any significant case.

[5] There is a whole conference series devoted to this topic, see https://en.wikipedia.org/wiki/International_Conference_on_Software_Reuse.

I.1 Equip frontend (JML, Java, ACSL, C, ...) as well as backend (Boogie, Silver, Why, ...) languages with precise, preferably formal, semantics. In the case of complex frontend languages this involves identifying a "core" that must then be supported by all tools.
I.2 Equip formal verification tools with a clear, modular structure and offer their functionality in well-documented APIs. This is a work-intensive task with few scientific rewards and, therefore, closely related to Challenge F.1.
I.3 Establish and maintain a tool integration community, to foster work on reuse and increase its appreciation as a valuable contribution.[6]

Method Integration. Arguably, one of the largest, self-imposed stumbling blocks of formal methods has been the propagation of monolithic approaches. At least in deductive verification, it became very clear within the last decade that software development, formal specification, formal verification, runtime verification, test case generation, and debugging are not separate activities, but they have to be done in concert. At the same time, formal specifications have to be incrementally developed and debugged just as the pieces of code whose behavior they describe. This is now commonly accepted in the community, even if the infrastructure is not there yet; however, there are encouraging efforts.

Achievements. It is impossible to list exhaustively the flurry of papers that recently combined formal verification with, for instance, abstract interpretation [117], debugging [58], invariant generation [82], software IDEs [92], testing [109], to give only a few examples.

Most deductive verification tools (as well as proof assistants) provide an interface to SMT solvers via the SMT-LIB [12] standard. There is growing interest in formal verification from the first-order theorem proving community where tools can be integrated via the TPTP standard [119]. There is also work towards the exchange of correctness witnesses among verifiers [17].

An interesting recent trend is that specialized verification and static analysis tools are being equipped with more general techniques. For example, the termination analysis tool AProVE [50] as well as the safety verification tool CPAchecker [18] both implement a symbolic execution engine to improve their precision. We observe that boundaries between different verification subcommunities that used to be demarcated by different methods and tools are dissolving.

Challenges. In addition to the tool integration challenges mentioned above, on the methodological level, questions of semantics and usability arise. To mention just one example, there is a plethora of approaches to loop invariant generation, see e.g., [46,63,114]. All of them come with certain limitations. They tend to be

[6] Relating to formal methods-based software tools in general, the journal *Software Tools for Technology Transfer (STTT)*, as well as the conference *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, were established as dedicated venues to foster tool integration and maturation. The article [118] discusses the history and the challenges of this endeavor, see also I.7.

driven by the technology they employ, not by applications and they are designed as stand-alone tools. This makes their effective usage very difficult.

Another area from whose integration deductive verification could benefit is machine learning, specifically, automata learning (see also the article *Combining Black-Box and White-Box Techniques for Learning Register Automata* in this issue).

I.4 Calls to auxiliary tools must return certificates, which must be re-interpreted in the caller's correctness framework. This is necessary to ensure correctness arguments without gaps.
I.5 The semantic assumptions on which different analysis methods are based must be spelled out, so that it is possible to combine different approaches in a sound manner. Some work in this direction has been done for the .NET static analyzer Clousot [35], but such investigations should be done on a much larger scale.
I.6 There is a plethora of possible combinations of tools and methods. So far, method and tool integration is very much *ad hoc*. There should be a systematic investigation about which combinations make methodological sense, what there expected impact is, and what effort their realization would require.
I.7 A research community working on method integration should be established.

Integration with the Software Production Environment. It is very difficult to integrate software verification technology into a production environment. Some of the reasons are of a non-technical nature and concern, for example, usability or the production context. These are explored in Sect. 6 below. Another issue might be the lack of coverage, see Sect. 5.3. In the following, we concentrate on processes and work flows.

Achievements. Our guiding question is: How can formal software verification be usefully integrated into a software development process? The emerging integration of verification, test generation, and debugging aspects into single tool chains, as described above, is an encouraging development. We begin to see deductive verification tools that are intentionally presented as enhanced software development environments, for example, the Symbolic Execution Debugger (SED) [56] based on Eclipse or the Dafny IDE [91] based on MS Visual Studio.

Several verification tools support users in keeping track of open proof obligations [59,80,91] after changes to the code or specification. This is essential to support incremental software development, but not sufficient. To realize versioning and team-based development of *verified* software, it is necessary to generalize code repositories into proof repositories [30]: a commit computes not merely changes, but a minimal set of new proof obligations that arise as a consequence of what was changed.

Another issue is that most verification attempts fail at first. It requires often many tries to render a complex verification target provable [39]. It is crucial to provide feedback to the user about the possible cause behind a failed proof.

Systems, such as KeY [3], can provide symbolic counter examples, and SED [56] computes symbolic backward slices from failure nodes in symbolic evaluation trees. The system StaDy [109] goes beyond this and uses dynamic verification to analyze failed proofs. The StaRVOOrS framework [34] generates optimized runtime assertion monitors for the unprovable parts of a specification.

In the context of commercial software production one can question whether functional verification is a worthwhile and realistic goal in the first place. Arguably, for safety- and security-critical code, as well as for software libraries used by millions, it is, but probably not for any kind of software. However, this does not mean that formal verification technology is restricted to the niches mentioned above, because there are many relevant formal verification scenarios, in addition to functional verification, notably: bug finding (discussed in Sect. 4) [15], information flow [44], and symbolic fault injection [83].

Challenges. The nature of software development is mostly incremental and evolutionary, and this must be accounted for by formal verification technology when used in commercial production. This is not the case at the moment.

Perhaps the biggest obstacle in functional verification is the lack of detailed enough specification annotations in the form of contracts and loop invariants. Without contracts, in particular for library methods, deductive verification does not scale. For some verification scenarios less precise annotations will do, but in general this is a huge bottleneck [13].

I.8 Implement proof repositories that support incremental and evolutionary verification and integrate them with verification tools.
I.9 Integrate automated specification generation techniques into the verification process.

5.3 Coverage

To make sure that deductive verification tools are practically usable, they need to support verification of a substantial part of the programming language. This means that for every construct of the programming language, verification techniques need to be developed (or at least, clear boundaries have to be provided, detailing what is covered, and what is not). Moreover, once the verification techniques are there, all variations of the programming language construct need to have tool support. Developing suitable verification techniques is typically a scientific challenge, but ensuring that a tool supports all variations of a language construct is mainly an engineering issue. If a language construct is not supported, preferably the tool design is such that it gracefully ignores the non-understood construct, and warns the user about this.

Achievements. State-of-the-art tools for deductive verification currently cover a very large part of the sequential fragment of industrially-used languages. To mention a few: OpenJML [37], KeY [3] and KIV [45] for Java, Frama-C [80], VeriFast [72] and Infer [32] for C, AutoProof [120] for Eiffel, and SPARK [60]

for Ada. These tools are mature enough to verify non-trivial software applications, and to identify real bugs in them, as discussed in Sect. 4. However, for more advanced language features such as reflection, and recent features such as lambdas in Java, verification technology still has to be developed (and thus, is currently not supported by these tools).

To provide tool support for a realistic programming language entails verification techniques such as reasoning about integer types (including overflow) [16], reference types, and exceptions [70]. Some of these, for example, support to reason about exceptions, became mainstream and are built into all modern deductive verification tools. In contrast, precise reasoning about integers, including overflow, often clutters up specifications and renders verification much harder. Therefore, many deductive verification tools abstract away from it, or provide it as an optional feature.

There is active research to investigate how to extend support for deductive verification to concurrent software. This opens up a whole new range of problems, because one has to consider all possible interleavings of the different program threads. Pen-and-paper verification techniques existed already for a long time [75,107], however, tool support for them remained a challenge.

The advent of concurrent separation logic [28,106] gave an important boost, as it enabled modular verification of individual threads in a (relatively) simple way. This has given rise to a plethora of new program logics to reason about both coarse-grained and fine-grained concurrency, see [29] for an overview. Also variations of separation logic for relaxed memory models have been proposed [121,124]. However, most of these logics still lack tool support.

In parallel to the theoretical developments, the basic ideas of concurrent separation logic, extended with permissions [25,26] started to find their way into deductive verification tools. Existing tools such as VeriFast [72], VerCors [9,20] and VCC [36] support verification of data race-freedom for different programming languages, using both re-entrant locks [6] and atomic operations as synchronisation primitives [7,71]. Current investigations focus on the verification of functional properties of concurrent software by means of abstraction [23]. In addition to Java and C, the VerCors tool set also supports reasoning about OpenCL kernels, which is using a different concurrency paradigm [22]. Also the KeY verifier provides some support to reason interactively about data race freedom of concurrent applications [103]. This approach can be used in addition to VeriFast and VerCors, and is in particular suitable to trace the source of a failing verification.

There also exist alternative verification techniques for concurrent software that use a restricted setup to achieve their goals. In particular, Cave [123] automatically proves memory safety and linearizability using an automated inference algorithm for RGSep, a combination of rely-guarantee reasoning and separation logic [125]. Just as the Infer tool mentioned above, it achieves automation by restricting the class of properties that can be verified. Another alternative line of work is to investigate more restricted concurrency models that allow near-sequential verification techniques. This is the approach advocated in ABS [74] which supports cooperative multitasking with explicit scheduling points [43].

Challenges. The main challenges with respect to coverage go into two different directions: one is to cover more aspects of the programming languages already supported; the other is to cover new classes of programming languages.

C.1 Precise verification of floating point numbers is essential for many algorithms, in particular in domains such as avionics. There is preliminary work [108], but a full-fledged implementation of floating point numbers in deductive verification systems is not yet available. A promising recent breakthrough is an automatable formal semantics for floating points numbers [27] which found its way also into the SMT-LIB and the SMT competition.

C.2 Tool support for verification of concurrent software is still in its infancy. We need further developments in two directions: (1) automated support of functional properties of fine-grained concurrency, which does not require an overwhelming amount of complex annotations, and can be used by non-experts in formal verification, and (2) verification techniques for relaxed memory models that resemble realistic hardware-supported concurrent execution models.

C.3 Reasoning techniques for programs that use reflection are necessary for application scenarios such as the analysis of obfuscated malware, or of dynamic software updates.

C.4 The rapid evolution of industrial programming languages (e.g., substantial new features are added to Java every 2–3 years) is a challenge for tools that are maintained with the limited manpower of academic research groups. Translation to intermediate languages is one way out, but makes it harder to provide feedback at the source level. Ulbrich [122] suggested a systematic framework for combining deductive verification at the intermediate language level with user interaction at the source level, but it has yet to be integrated into a major tool.

C.5 Deductive verification technology is not merely applicable to software, but also to cyber-physical systems, as they exhibit similar properties [52]. Computational engineers are mainly working with partial differential equations to describe their systems, and they implement these in C, MATLAB, etc. There are some results and tools for deductive verification of hybrid systems [49]. Hybrid systems have been traditionally modeled with differential equations (see the article *Multi-Mode DAE Models: Challenges, Theory and Implementation* in this issue) and automata-based techniques (article *Continuous-time Models for System Design and Analysis* in this issue). It is an open problem to find out how these different methodological approaches relate to and could benefit from each other.

6 Achievements and Challenges: Non-technical

6.1 Usability

Research in formal verification is method- and tool-driven. As a consequence, the effectiveness of a novel method or a new tool is usually simply claimed without

justification or, at best, underpinned by citing execution statistics. The latter are often micro benchmarks carried out on small language fragments. The best case are industrial case studies which may or may not be representative and in nearly all publications these are performed by the researchers and tool builders themselves, not by the intended users.

To convince industrial stakeholders of the usefulness of a formal verification approach, it is not only necessary to demonstrate that it can fit into the existing development environment (see Sect. 5.2), but also to argue that one can solve tasks more effectively or faster than with a conventional solution. This is only possible with the help of experimental user and usability studies.

Achievements. There are very few usability studies around formal verification tools. We know of an evaluation of KeY and Isabelle based on focus groups [14], while the papers [21,57,69] contain user studies or analyses on API usage, prover interfaces, and proof critics, respectively. There are a few papers that attempt to construct user models or elicit user expectations, but [57] seems to be the *only* experimental user study so far that investigated the impact of design decisions taken in a verification system on user performance.

Challenges. To guide research about formal verification so that it has impact on industrial practice, it is essential to back up claims on increased effectiveness or productivity with controlled user experiments. This has been proven to be beneficial in the fields of Software Engineering and Computer Security.

U.1 Claims about increased effectiveness or productivity attributed to new methods or tools should be backed up by experimental user studies.
U.2 Establish the paper category *Experimental User Study* as an acceptable kind of submission in formal verification conferences and journals.

6.2 Funding

To support formal verification of industrial languages in real applications requires a sustained effort over many years. As detailed in the previous sections, to specify and to reason about programs means that the semantics of the language they are written in must be fully and deeply understood, solutions for inference and its automation must be found, suitable specification abstractions must be discovered. To formulate appropriate theoretical and methodological underpinnings took decades and the process is still not complete for complex aspects such as floating point types and weak memory models (Sect. 5.3).

The road from the first axiomatic descriptions of program logics (Sect. 3) to the verification of software written in major programming language that is actually in use was long, and we are by far not at its end. It takes a long view, much patience, and careful documentation to avoid "re-invention of the wheel" or even regression. Tool building is particularly expensive and can take decades. To protect these large investments and to ensure measurable progress, long-term projects turned out to be most suitable.

Achievements. There are several long-term projects in deductive software verification that have sufficiently matured to enable industrial applicability (see also Sect. 4). We mention ACL2[7], Boogie[8], KeY[9], KIV[10], OpenJML[11], SparkPro[12], and Why/Krakatoa[13].

Challenges. Some of the long-term projects mentioned above are supported by research labs with strong industrial ties (Altran, INRIA, MSR). Unfortunately, neither the trend to embedded industrial research nor the current climate of academic funding are very well suited for this kind of enterprise. The challenge for ambitious projects, such as DeepSpec[14], is their continuation after the initial funding runs out. It is worrying that all existing long-term academic projects on deductive software verification were started before 2000. Further detrimental factors to long-time engineering-heavy projects are the publication requirements for tenured positions in Computer Science as well as the unrealistic expectations on short-term impact demanded from many funding agencies. Successful long-term research is not "disruptive" in its nature, but slowly and systematically builds on previous results. On the other hand, usability aspects of formal verification are hardly ever evaluated.

F.1 The academic reward system should give incentives for practical achievements and for long-term success (see [4] for some concrete suggestions how this could be achieved).
F.2 Large parts of Computer Science should be classified and treated as an Engineering or Experimental Science with an according funding model. Specifically, there needs to be funding for auxiliary personnel (professional software developers) and for software maintenance: complex software systems should be viewed like expensive equipment, such as particle colliders. The base level of funding should be that of an engineering or experimental science, not a mathematical science.
F.3 Grant proposals should foresee and include funding to carry out systematic experimental studies, also involving users. For example, money to reward the participants of user studies must be allocated.

6.3 Industrial and Societal Context

The best prospects for industrial take-up of deductive verification technology is in application areas that are characterized by high demands on software quality.

[7] http://www.cs.utexas.edu/~moore/acl2/.
[8] https://github.com/boogie-org/boogie.
[9] http://www.key-project.org/.
[10] http://www.isse.uni-augsburg.de/en/software/kiv/.
[11] http://www.openjml.org/.
[12] http://www.adacore.com/sparkpro/.
[13] http://krakatoa.lri.fr/.
[14] https://deepspec.org/.

This is clearly the case for safety- and security-critical domains that are regulated by formal standards overseen by certification authorities.

In many other application domains, however, timely delivery or new features are considered to be more important than quality. A contributing factor are certainly the relatively weak legal regulations about software liability. With the ongoing global trend in digitalization, however, we might experience a surge in software that can be deemed as safety- or security-critical, in particular, in the embedded market (e.g., self-driving cars [2], IoT). On the other hand, that market is partially characterized by a strong vendor lock-in in the form of modeling tools such as MATLAB/Simulink, which have no formal foundations. An interesting side effect of digitalization is the arrival of companies on the software market that so far had no major stake in software. Here is an opportunity for formal methods and formal verification, in particular, since software verification tools are as well applicable to cyber-physical systems [52,73] (see Challenge C.5).

Formal specification and deductive verification methods are expressed relative to a target programming language. New features of languages such as C/C++ or Java are not introduced with an eye on verifiability, making formal verification and coverage unnecessarily difficult.

Achievements. The latest version of the DO-178C standard [115], which is the basis for certification for avionics products, contains the *Formal Methods Supplement* DO-333 that permits formal methods to complement testing. This makes it, in principle, possible to argue that formal verification can speed up or decrease the cost of certification.

The development of the concurrent modeling language ABS [74] demonstrated that it is possible to design a complex programming language with many advanced features that has an associated verification tool box with high coverage [127], provided that analyzability and verifiability are taken into account during language design.

Challenges. In order to ensure substantial impact of deductive software verification in society and industry, a coordinated effort is necessary to influence standardization and certification activities.

ISC.1 Researchers from the formal verification area should become involved in language standardization. In general, research in the fields of programming languages and formal verification must be better coordinated.

ISC.2 Researchers from the formal verification area should become actively involved in the standardization efforts of certification authorities.

ISC.3 Specific quality assurance measures for verification tools such as test coverage, incremental testing, external validation, etc. should be developed and applied. If deductive software verification should become usable in certification activities, the software quality of the verification tools themselves is a critical issue.

7 Summary

We described the progress made in the area of deductive software verification. Starting as a pen-and-paper activity in the late 1960s, deductive verification nowadays is a mature technique and it can substantially increase the reliability of software in actual production. Advanced tool support is available to reason about the behaviour of complex programs and library code, written in mainstream programming languages. Industrial applicability of deductive verification is witnessed by several success stories.

However, there are many challenges that need to be addressed to make the transfer from an academic technique to a technique that is a *routine* part of *commercial* software development processes. We divided these challenges into two categories: technical and non-technical. Technical challenges relate to what properties can be verified, what programs can we reason about, how we can make verification largely automatic, and how we provide feedback when verification fails. Non-technical challenges relate to how we can fund all necessary engineering efforts, how we can ensure that tool developers get sufficient scientific credits, and how to convince industrial management that the extra effort needed for verification will actually be beneficial. We hope that these challenges can serve as an incentive for future research directions in deductive software verification.

Acknowledgements. We are grateful to Alastair Donaldson, Michael Leuschel, Peter H. Schmitt and Bernhard Steffen, for carefully reading our paper and for their very useful feedback. Many thanks to Richard Bubel for help with the preparation of the example in Sect. 2.

References

1. Abrial, J.-R.: The B Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Ackerman, E.: Hail, robo-taxi!. IEEE Spectr. **54**(1), 26–29 (2017)
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P., Ulbrich, M. (eds.): Deductive Software Verification-The KeY Book: From Theory to Practice. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Alglave, J., Donaldson, A.F., Kroening, D., Tautschnig, M.: Making software verification tools really work. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_3
5. Ameri, M., Furia, C.A.: Why just boogie? Translating between intermediate verification languages. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 79–95. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_6
6. Amighi, A., Blom, S., Darabi, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Verification of concurrent systems with VerCors. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 172–216. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_5

7. Amighi, A., Blom, S., Huisman, M.: Resource protection using atomics. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 255–274. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_14
8. Amighi, A., Blom, S., Huisman, M.: VerCors: a layered approach to practical verification of concurrent software. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP, Heraklion, Crete, Greece, pp. 495–503. IEEE Computer Society (2016)
9. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. Logical Methods Comput. Sci. **11**(1) (2015)
10. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
11. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
12. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)
13. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification - specification is the new bottleneck. In: Cassez, F., Huuck, R., Klein, G., Schlich, B. (eds.) Proceedings of the 7th Conference on Systems Software Verification. EPTCS, vol. 102, pp. 18–32 (2012)
14. Beckert, B., Grebing, S., Böhl, F.: A usability evaluation of interactive theorem provers using focus groups. In: Canal, C., Idani, A. (eds.) SEFM 2014. LNCS, vol. 8938, pp. 3–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15201-1_1
15. Beckert, B., Klebanov, V., Ulbrich, M.: Regression verification for Java using a secure information flow calculus. In: Monahan, R. (ed.) Proceedings of the 17th Workshop on Formal Techniques for Java-Like Programs, FTfJP, Prague, Czech Republic, pp. 6:1–6:6. ACM (2015)
16. Beckert, B., Schlager, S.: Software verification with integrated data type refinement for integer arithmetic. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM 2004. LNCS, vol. 2999, pp. 207–226. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24756-2_12
17. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, Seattle, WA, USA, pp. 326–337. ACM (2016)
18. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 195–211. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_14
19. Bjørner, N., Browne, A., Colón, M., Finkbeiner, B., Manna, Z., Sipma, H., Uribe, T.E.: Verifying temporal properties of reactive systems: a step tutorial. Formal Methods Syst. Des. **16**(3), 227–270 (2000)
20. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9

21. Blom, S., Huisman, M., Kiniry, J.: How do developers use APIs? A case study in concurrency. In: International Conference on Engineering of Complex Computer Systems (ICECCS), Singapore, pp. 212–221. IEEE Computer Society (2013)
22. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014)
23. Blom, S., Huisman, M., Zaharieva-Stojanovski, M.: History-based verification of functional behaviour of concurrent programs. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 84–98. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_6
24. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages (2011)
25. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Long Beach, California, USA, pp. 259–270. ACM (2005)
26. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
27. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: 22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, pp. 160–167. IEEE (2015)
28. Brookes, S.: A semantics for concurrent separation logic. Theoret. Comput. Sci. **375**(1–3), 227–270 (2007)
29. Brookes, S., O'Hearn, P.: Concurrent separation logic. ACM SIGLOG News **3**(3), 47–65 (2016)
30. Bubel, R., Damiani, F., Hähnle, R., Johnsen, E.B., Owe, O., Schaefer, I., Yu, I.C.: Proof repositories for compositional verification of evolving software systems. In: Steffen, B. (ed.) Transactions on Foundations for Mastering Change I. LNCS, vol. 9960, pp. 130–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46508-1_8
31. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing 1974, pp. 308–312. Elsevier/North-Holland, Amsterdam (1974)
32. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
33. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exp. **35**, 583–599 (2005)
34. Chimento, J.M., Ahrendt, W., Pace, G.J., Schneider, G.: StaRVOOrS: a tool for combined static and runtime verification of Java. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 297–305. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_21
35. Christakis, M., Müller, P., Wüstholz, V.: An experimental evaluation of deliberate unsoundness in a static program analyzer. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 336–354. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_19

36. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
37. Cok, D.: OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) 1st Workshop on Formal Integrated Development Environment, (F-IDE). EPTCS, vol. 149, pp. 79–92 (2014)
38. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_6
39. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
40. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Deussen, P., Hansmann, A., Käufl, T., Klingenbeck, S.: The verification system *Tatzelwurm*. In: Broy, M., Jähnichen, S. (eds.) KORSO: Methods, Languages, and Tools for the Construction of Correct Software. LNCS, vol. 1009, pp. 285–298. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0015468
42. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
43. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35
44. Do, Q.H., Bubel, R., Hähnle, R.: Exploit generation for information flow leaks in object-oriented programs. In: Federrath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 401–415. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_27
45. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: overview and verifythis competition. STTT **17**(6), 677–694 (2015)
46. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
47. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
48. Floyd, R.W.: Assigning meanings to programs. Proc. Symp. Appl. Math **19**, 19–31 (1967)
49. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36

50. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_13
51. Hähnle, R., Heisel, M., Reif, W., Stephan, W.: An interactive verification system based on dynamic logic. In: Siekmann, J.H. (ed.) CADE 1986. LNCS, vol. 230, pp. 306–315. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-16780-3_99
52. Kamburjan, E., Hähnle, R.: Uniform modeling of railway operations. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2016. CCIS, vol. 694, pp. 55–71. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-53946-1_4
53. Hähnle, R., Menzel, W., Schmitt, P.: Integrierter deduktiver Software-Entwurf. Künstliche Intelligenz, pp. 40–41, December 1998
54. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundations of Computing. MIT Press, Cambridge (2000)
55. Heisel, M., Reif, W., Stephan, W.: Program verification by symbolic execution and induction. In: Morik, K. (ed.) GWAI-87 11th German Workshop on Artifical Intelligence. Informatik-Fachberichte, vol. 152, pp. 201–210. Springer, Heidelberg (1987). https://doi.org/10.1007/978-3-642-73005-4_22
56. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 255–262. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_21
57. Hentschel, M., Hähnle, R., Bubel, R.: An empirical evaluation of two user interfaces of an interactive program verifier. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, pp. 403–413. ACM Press, September 2016
58. Hentschel, M., Hähnle, R., Bubel, R.: The interactive verification debugger: effective understanding of interactive proof attempts. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, pp. 846–851. ACM Press, September 2016
59. Hentschel, M., Käsdorf, S., Hähnle, R., Bubel, R.: An interactive verification tool meets an IDE. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 55–70. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_4
60. Hoang, D., Moy, Y., Wallenburg, A., Chapman, R.: SPARK 2014 and GNATprove: a competition report from builders of an industrial-strength verifying compiler. STTT **17**(6), 695–707 (2015)
61. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580, 583 (1969)
62. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**, 271–281 (1972)
63. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_7
64. Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. Comput. J. **38**(2), 131–141 (1995)
65. Huisman, M.: Reasoning about Java programs in higher order logic with PVS and Isabelle. Ph.D. thesis, University of Nijmegen (2001)
66. Huisman, M., Ahrendt, W., Grahl, D., Hentschel, M.: Formal specification with the Java modeling language. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R.,

Schmitt, P., Ulbrich, M. (eds.) Deductive Software Verification - The KeY Book. LNCS, vol. 10001, pp. 193–241. Springer, Cham (2016)
67. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, pp. 284–303. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46428-X_20
68. Huisman, M., Monahan, R., Müller, P., Poll, E.: VerifyThis 2016: a program verification competition. Technical report TR-CTIT-16-07, Centre for Telematics and Information Technology, University of Twente, Enschede (2016)
69. Ireland, A., Jackson, M., Reid, G.: Interactive proof critics. Formal Asp. Comput. **11**(3), 302–325 (1999)
70. Jacobs, B.: A formalisation of Java's exception mechanism. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 284–301. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45309-1_19
71. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Austin, TX, USA, pp. 271–282. ACM (2011)
72. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
73. Jeannin, J., Ghorbal, K., Kouskoulas, Y., Gardner, R., Schmidt, A., Zawadzki, E., Platzer, A.: Formal verification of ACAS X, an industrial airborne collision avoidance system. In: Girault, A., Guan, N. (eds.) International Conference on Embedded Software, EMSOFT, Amsterdam, Netherlands, pp. 127–136. IEEE (2015)
74. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
75. Jones, C.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
76. Kassios, I.T.: The dynamic frames theory. Formal Asp. Comput. **23**(3), 267–288 (2011)
77. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing verification condition generation with symbolic execution: an experience report. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 196–208. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_16
78. Kaufmann, M., Moore, J.S.: Design goals for ACL2. In: Third International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, pp. 92–117 (1994)
79. Kiniry, J.R., Morkan, A.E., Cochran, D., Fairmichael, F., Chalin, P., Oostdijk, M., Hubbers, E.: The KOA remote voting system: a summary of work to date. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 244–262. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75336-0_16
80. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)

81. Kosmatov, N., Marché, C., Moy, Y., Signoles, J.: Static versus dynamic verification in Why3, Frama-C and SPARK 2014. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 461–478. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_32
82. Kovács, L.: Symbolic computation and automated reasoning for program analysis. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 20–27. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_2
83. Larsson, D., Hähnle, R.: Symbolic fault injection. In: Beckert, B. (ed.) Proceedings of the 4th International Verification Workshop (Verify) in Connection with CADE-21 Bremen, Germany, vol. 259, pp. 85–103. CEUR Workshop Proceedings (2007)
84. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. Technical report 98-06y, Iowa State University, Department of Computer Science (2003). Revised June 2004
85. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual, May 2013. Draft revision 2344
86. Lehner, H., Müller, P.: Formal translation of bytecode into BoogiePL. Electr. Notes Theor. Comput. Sci. **190**(1), 35–50 (2007)
87. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007-2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_7
88. Leino, K., Nelson, G., Saxe, J.: ESC/Java user's manual. Technical report SRC 2000–002, Compaq System Research Center (2000)
89. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
90. Leino, K.R.M., Nelson, G.: An extended static checker for Modula-3. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 302–305. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0026441
91. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings of the 1st Workshop on Formal Integrated Development Environment, F-IDE, Grenoble, France. EPTCS, vol. 149, pp. 3–15 (2014)
92. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 380–397. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_22
93. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models with ProB. Formal Asp. Comput. **23**(6), 683–709 (2011)
94. Liskov, B., Wing, J.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(1), 1811–1841 (1994)
95. Litvintchouk, S.D., Pratt, V.R.: A proof-checker for dynamic logic. In: Reddy, R. (ed.) Proceedings of the 5th International Joint Conference on Artificial Intelligence, pp. 552–558. William Kaufmann, Cambridge (1977)
96. Logozzo, F.: Practical verification for the working programmer with CodeContracts and abstract interpretation. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 19–22. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_3

97. Luckham, D.C., von Henke, F.W.: An overview of Anna, a specification language for Ada. IEEE Softw. **2**(2), 9–22 (1985)
98. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. J. Log. Algebr. Program. **58**(1–2), 89–106 (2004)
99. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (1992)
100. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
101. Mohsen, M., Jacobs, B.: One step towards automatic inference of formal specifications using automated VeriFast. In: ter Beek, M.H., Gnesi, S., Knapp, A. (eds.) FMICS/AVoCS -2016. LNCS, vol. 9933, pp. 56–64. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45943-1_4
102. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B. (ed.) Proceedings of the 4th Interenational Verification Workshop in connection with CADE-21, Bremen, Germany. CEUR Workshop Proceedings, vol. 259. CEUR-WS.org (2007)
103. Mostowski, W.: Dynamic frames based verification method for concurrent Java programs. In: Gurfinkel, A., Seshia, S.A. (eds.) VSTTE 2015. LNCS, vol. 9593, pp. 124–141. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29613-5_8
104. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
105. Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998)
106. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoret. Comput. Sci. **375**(1–3), 271–307 (2007)
107. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica J. **6**, 319–340 (1975)
108. Paganelli, G., Ahrendt, W.: Verifying (in-)stability in floating-point programs by increasing precision, using SMT solving. In: Bjørner, N., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S.M., Zaharie, D. (eds.) 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, 23–26 September 2013, pp. 209–216. IEEE Computer Society (2013)
109. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? Testing helps to find the reason. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 130–150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41135-4_8
110. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60–61**, 17–139 (2004)
111. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 414–434. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_26
112. Praxis Critical Systems. SPARK–The SPADE Ada Kernel, 3.2 edition (1996)
113. Robinson, K.: The B method and the B toolkit. In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349, pp. 576–580. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0000503
114. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. **64**(1), 54–75 (2007)

115. RTCA. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, January 2012
116. Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., Reif, W.: Development of a verified flash file system. In: Ameur, Y.A., Schewe, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. LNCS, vol. 8477, pp. 9–24. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_2
117. Scheurer, D., Hähnle, R., Bubel, R.: A general lattice model for merging symbolic execution branches. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 57–73. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_5
118. Steffen, B.: The physics of software tools: SWOT analysis and vision. Softw. Tools Technol. Transf. (STTT) **19**(1), 1–7 (2017)
119. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reason. **43**(4), 337–362 (2009)
120. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 566–580. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_53
121. Turon, A., Vafeiadis, V., Dreyer, D.: GPS: navigating weak memory with ghosts, protocols, and separation. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA, Portland, OR, USA, pp. 691–707. ACM (2014)
122. Ulbrich, M.: Dynamic logic for an intermediate language: verification, interaction and refinement. Ph.D. thesis, Karlsruhe Institute of Technology (2013)
123. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_40
124. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for C11 concurrency. In: OOPSLA 2013. ACM (2013)
125. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
126. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concur. Comput.: Pract. Exp. **13**(13), 1173–1214 (2001)
127. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT **14**(5), 567–588 (2012)