6.1 Introduction
Edsger Dijkstra once remarked, with tongue firmly in cheek, that he and Tony Hoare both drew their notoriety from a clever algorithm and an oft-repeated quote. In Dijkstra’s case, these were the shortest paths algorithm and the observation that ‘testing can prove the presence of bugs but never their absence.’ For Hoare, the algorithm in question was Quicksort, and the quote: ‘Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.’ Both Hoare and Dijkstra were motivated by the challenge of crafting software that was correct by construction. Dijkstra framed this challenge as one of using a finite brain and finite language to reason about infinite behaviors.\(^1\) The scale of proof construction required to handle millions of lines of

\(^1\) In his 1972 Turing Award lecture, Dijkstra says, ‘The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full
code with combinatorically complex interactions motivated the need for structure so that proofs are linear in the size of the code. Hoare logic with triples capturing a program with its precondition and postcondition introduced a *structural proof theory* for efficiently grasping the infinite within the finite. Since its invention in 1969, Hoare logic has underpinned a large number of verification systems and has spawned a large number of variants. It has also influenced the design of programming languages, particularly with respect to support for assertions that can be used both for run-time verification as well as proof.

Software has taken over as a dominant technological force in the 21st century. In the words of Marc Andreessen, ‘Software is eating the world.’ As a consequence, software bugs and security breaches have gnawed away at our faith in technology. If software is at the core of our factories, transportation systems, spacecraft, power grid, entertainment, and commerce, then we need confidence that it will work securely and reliably. This was the motivation behind Tony Hoare’s introduction of the Verified Software Initiative (VSI), which led to a cooperative international effort on the Verification Grand Challenge.

The software crisis had been identified in a 1968 NATO Software Engineering conference [Naur and Randell 1969]. By 2000, it was widely agreed in the research community and also in technology companies such as Amazon and Microsoft that mathematical methods, such as the inductive assertion logic of Hoare, were key to producing reliable software. There were a number of ongoing efforts on automating the verification process. Tony Hoare articulated that machines are fast enough and the logical theories of programs are strong enough to warrant a concerted effort toward large-scale automated verification of industrial code, under the VSI [Hoare 2002]. He was instrumental in organizing an international community of researchers. The VSI effort was kicked off at the 2005 Verified Software conference [Meyer and Woodcock 2008]. The event, hosted by Bertrand Meyer at ETH Zurich, attracted almost 100 leading researchers in program verification and included both technical presentations and discussions that helped shape the agenda for the VSI. This article summarizes significant developments following the initial VSI meeting.

A lot has changed since the launch of VSI in 2005, and some of the noteworthy milestones are:

*humility, and among other things he avoids clever tricks like the plague.* He concludes his lecture with the words: ‘We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers.’ He elaborates on this theme in his 1989 CACM article ‘On the cruelty of really teaching computing science’ [Dijkstra 1989].
1. The successful verification of realistic software components such as the CompCert C compiler [Leroy 2009], the seL4 microkernel [Klein et al. 2009], complex air-traffic collision avoidance and resolution algorithms [Butler et al. 2010], and the CakeML compiler and run-time system [Kumar et al. 2014]. These were preceded by landmark verification exercises like the Computational Logic, Inc. (CLI) stack in the 1990s [Bevier et al. 1989], but the newer projects have yielded practical verified software.

2. The POPLMark challenge [Aydemir et al. 2005], which created a movement to formalize the claims and proofs in research papers on the semantic foundations of programming languages. Many conferences have artifact evaluation that includes mechanized proofs.

3. The growth of SAT and SMT solvers, which serve as the automated inference service for a number of verification tools [Clarke et al. 2018].

4. The rise of intermediate verification languages like Boogie [Barnett et al. 2005], Why [Filliátre and Paskevich 2013], and Viper [Müller et al. 2016]; intermediate static analysis languages like CIL [Necula et al. 2002], LLVM [Lattner and Adve 2004] and Infer [Calcagno and Distefano 2011]; and intermediate verification techniques like Constrained Horn Clause solving [Björner et al. 2015], which serve as an interface layer between programming languages and back-end verification tools.

5. Theoretical and practical advances in static analysis.

In this chapter, we first give a brief overview of influential verification techniques and technologies that predate the VSI. The rest of the chapter is then devoted to major results in the three thrusts of the VSI: theory, tools, and experiments.

6.2 The Roots of the Verified Software Initiative

The science of operations, as derived from mathematics more especially, is a science of itself, and has its own abstract truth and value. Ada Lovelace

The big idea is that programs are just mathematical formulas that capture the possible executions [Elspas et al. 1972]. Some program representations, like functional programming and logic programming, are closer to logic, while the more operational representations captured by procedural programming languages have to be viewed through a logical lens such as Hoare logic [Hoare 1969]. The characterization of computation using logic opens up the possibility of employing symbolic calculation that in many cases can be effectively mechanized. We take a quick
look at the interplay between programming and calculational logic as a means of judging the correctness of programs.

For a function $f$, one could assert a property of $f$ of the form $P(x) \Rightarrow Q(x, f(x))$, where $P$ is a predicate on the input $x$ and $Q$ is a predicate relating the input $x$ with the output $f(x)$. With Hoare logic, a program statement $S$ maps a pre-state $s$ to a post-state $s'$ or it diverges. A Hoare triple $\{P\} S \{Q\}$ captures the claim that a program $S$ executed on an initial state satisfying the precondition assertion $P$ can either diverge or terminate in a state satisfying the postcondition assertion $Q$. The assertions $P$ and $Q$ contain both logical variables that refer to identical values in both the precondition and postcondition assertions, and program variables whose assignments are modified by $S$ in the transition from the pre-state to the post-state. For example, if the program $S$ consists of the single statement $x := x + 1$, then we might have a precondition of the form $\{x \geq 0\}$ and a postcondition of the form $\{x > 0\}$. We might want to claim that the output value of $x$ is greater than the input value, and in this case we need to use a logical variable $y$ to record the prior value of $x$. The precondition then becomes $y = x$, and the postcondition reads $x > y$ yielding the triple $\{y = x\} x := x + 1 \{x > y\}$.

The use of assertions interspersed with the statements opens up several possibilities:

1. Verification condition (VC) generation can be used to generate proof obligations (the VCs) that entail the correctness claim captured by the Hoare triple. For the simple program above, this would involve proving that $x \geq 0 \Rightarrow x + 1 > 0$. As demonstrated in the work at SRI [Shostak et al. 1982] and Stanford [Luckham et al. 1979] starting in the late 1970s and early 1980s, proof obligations of this sort are easily discharged by means of a solver for satisfiability modulo theories (SMT). When the datatypes involved are finite, such as when $x$ is a 16-bit unsigned integer, the values can be binary coded to yield proof obligations that are Boolean formulas that can be verified using a Boolean satisfiability (SAT) solver. These satisfiability solvers can also be used to generate test inputs that explore specific program paths.

2. The symbolic evaluation of programs becomes feasible by unrolling the program and propagating the consequence of assertions. This kind of symbolic evaluation, combined with concrete evaluation on test inputs, can be used for concolic test generation [Godefroid et al. 2005] and bounded model checking [Clarke et al. 2001].

3. Logical abstractions can be used to model and approximate the behavior of programs on concrete inputs by their abstract counterparts. For example, the program $x := x + 1$ maps an abstract state $x > 0$ to itself. These abstractions can either be over a fixed range of predicates, like $x > 0$, or
computed dynamically as in abstract interpretation. These abstractions can be used to compute procedure summaries that characterize a sufficiently strong postcondition for a given precondition, or conversely a sufficiently weak precondition for a given postcondition.

4. We can extend the precision of the assertions to range over heap data. Naïve versions of the Hoare logic rules assume that the variables represent independent memory segments, whereas with heap data references can be shared between data structures and aliased across multiple variables. In the separation logic [O’Hearn et al. 2001] variant of Hoare logic, assertions characterize a restricted memory footprint and \( P \cdot Q \) indicates that assertion \( P \) and \( Q \) hold on disjoint footprints. The frame rule

\[
\frac{\{ P \} S \{ P' \}}{\{ P \cdot Q \} S \{ P' \cdot Q \}}
\]

allows the frame condition \( Q \) to be preserved from the precondition to the postcondition as long as it does not mention any program variables that are modified by \( S \).

5. True concurrency can be incorporated by merging the effects of parallel statements that operate on disjoint memory regions. Interleaving concurrency can be incorporated by ensuring that environment actions preserve local assertions. In summary, the structural proof theory of Hoare logic opens up the possibility of the structural exploration of program properties using powerful tools like satisfiability solvers, constraint solving, abstraction, and fixpoint calculation.

Many of the building blocks mentioned above already existed prior to Hoare’s initiation of the Verified Software Grand Challenge. We summarize some of the verification technologies below. A number of ambitious verification systems were constructed in the 1970s, building on Jim King’s idea of a verifying compiler [King 1970]. These include AFFIRM [Gerhart et al. 1980], Gypsy [Ambler et al. 1977], FDM [Kemmerer 1980], and HDM [Silverberg et al. 1979]. The Boyer–Moore family of theorem provers [Boyer and Moore 1979, 1998] was developed based on techniques for induction and rewriting for proving properties of functional programs. Temporal logic [Pnueli 1977] was introduced as a formalism for reasoning about reactive program behavior. Abstract interpretation [Cousot and Cousot 1977] was introduced as a powerful static analysis technique for deriving program invariants. The 1980s saw the development of explicit model checking techniques, program synthesis techniques for functional and reactive programs [Manna and Waldinger 1980, Clarke and Emerson 1981, Manna and Wolper 1984], and automated and interactive theorem provers such as Nqthm [Boyer and Moore 1998], RRL [Kapur et al. 1986],...

The 1990s also saw some spectacular failures due to computing error such as the Intel FDIV bug and the aborted Ariane 5 launch. These motivated developments in hardware and software verification and static analysis.

Since 2005, there has been remarkable progress toward the vision of the Verified Software Grand Challenge building on the foundations of prior work. It is difficult to estimate the degree to which specific developments in the field have been directly inspired by Hoare’s vision of a verified software ecosystem of unified foundations, integrated tools, and interoperable verified software components. We believe that much of the work has been compatible with this original vision and we can see the faint outlines of an emerging verified software ecosystem. The theoretical advances include the development of logics for reasoning about heap-manipulating programs, concurrent programs, and hyper-properties. Tool development was significantly accelerated through the availability of high-performance SAT and SMT solvers, intermediate verification languages like Boogie and Why that supported automated front-end verification tools for practical programming languages, and extensive background and domain libraries for verification, and symbolic execution engines. The rise of competitions for verification techniques and tools has spurred a number of innovations in verification technology. We have seen large and ambitious long-term verification efforts directed at microkernels
[Klein et al. 2009], file systems [Chen et al. 2015], air-traffic control systems [Butler et al. 2010], compilers [Leroy 2009], cryptographic operations and protocols [Barthe et al. 2014], and cyber-physical systems [Loos et al. 2011]. In the next three sections, we survey some of the key results achieved during the first fifteen years of the VSI in the areas of theory, tools, and experiments.

6.3 Theory

When the VSI was kicked off in 2005, the theory of program verification was already well understood. Floyd [Floyd 1967] and Hoare [Hoare 1969] laid the foundations for deductive program verification, Dijkstra's predicate transformer [Dijkstra 1975] provided a way to automate the computation of verification conditions, and Owicki and Gries [Owicki and Gries 1976] as well as Jones [Jones 1981] generalized deductive program verification to concurrent programs. In the area of algorithmic verification, model checking [Clarke et al. 1983] offered an approach to verify temporal properties fully automatically, initially for finite-state systems and later, via symbolic model checking [Burch et al. 1990], also for infinite-state systems. Abstract interpretation [Cousot and Cousot 1977] offered a general framework for reasoning about abstractions of program executions, which is the basis of most static program analyses. Many of these reasoning techniques leverage theorem provers to discharge verification conditions or to compute program abstractions. With the Curry–Howard isomorphism, the foundation of interactive theorem provers was long known, and Nelson and Oppen's cooperating decision procedures [Nelson and Oppen 1979] provided the basis for modern SMT solvers.

Similarly, there was a rich theory on how to specify program properties. Modeling languages such as VDM [Jones 1980] and Z [Abrial et al. 1980] provided a way to specify the high-level design of systems in an implementation-independent way. Assertion languages such as the Larch family of languages [Guttag et al. 1993] and Eiffel [Meyer 1991] allowed programmers to express properties of code, typically using preconditions, postconditions, and invariants and to connect those to design-level specifications following Hoare’s seminal work on data abstraction [Hoare 1972]. Behavioral subtyping [Liskov and Wing 1994] explained how to apply these ideas to languages with subtype polymorphism, in particular, object-oriented languages. There was also a rich theory on temporal specifications, most prominently in LTL [Pnueli 1977], CTL [Clarke and Emerson 1981], and TLA [Lamport 1994], the specification formalisms used by most model checkers.

Despite this rich foundation, pursuing the VSI required substantial advances in the theory of program verification. In this section, we give an overview of some major developments that happened during the course of the VSI. Many other interesting developments had to be omitted for lack of space.
Heap reasoning. Modular verification of heap-manipulating programs is challenging because, in general, any piece of code may read and modify any heap data structure via a chain of references. Consequently, callers of a method require precise information about the effects of the callee on the heap in order to determine which heap properties are preserved by the call and which ones have changed. Whereas the latter can be expressed by the callee's postcondition, the former is more difficult: a postcondition cannot list all heap properties that were not modified by the method without breaking modularity and information hiding. The problem of proving which heap properties are unaffected by an operation—the so-called frame problem—is the main challenge of heap verification.

The first modular verification techniques for heap-manipulating programs appeared shortly before the start of the VSI. One approach to framing was to combine familiar effect specifications using modifies-clauses with a notion of hierarchical ownership, which allowed specifications to abstract over implementation details and facilitated verification by restricting the possible aliasing among objects [Müller 2002]; this approach was for instance taken in the Spec# verification system [Leino and Müller 2004], which was first demonstrated at the kick-off event of the VSI in Zurich in 2005. This approach proved to be too inflexible because many common data structures require forms of aliasing that are not permitted by hierarchical ownership.

An alternative approach was proposed by O’Hearn, Reynolds, and Yang in their seminal work on separation logic [O’Hearn et al. 2001]: Heap locations are resources whose ownership is held by method executions and is transferred between executions upon calls and returns. Which resources to transfer is prescribed by the callee’s pre- and postconditions employing dedicated assertions such as separation logic’s points-to assertion. As long as a method execution owns any given location, no other method can own (and consequently modify) it, which admits an elegant frame rule. However, at the beginning of the VSI separation logic was still in its infancy: there were no abstraction mechanisms to specify properties of unbounded sets of memory locations and it was unclear how to automate the generation of proof obligations for this new logic.

During the course of the VSI, there was tremendous progress on heap reasoning. The community has developed dozens of separation logics, ever expanding the expressiveness in terms of both programs and properties that can be verified. Some key results include the introduction of data abstraction using predicates [Parkinson and Bierman 2005], higher-order separation logic [Birkedal et al. 2005], and many concurrent separation logics, which we discuss below. Moreover, various algorithms for automating separation logic proofs were proposed, including a form of symbolic execution [Berdine et al. 2005] (used in the VeriFast [Jacobs et al. 2011]

Dynamic frames [Kassios 2006] is a heap verification technique in which the read and write effects of operations are specified as sets of locations. These sets are expressed as heap-dependent functions or ghost state. Therefore, their value can change dynamically, as the name of the technique suggests. A property is then framed around a call by proving that the read effect of the property is disjoint from the write effect of the called method. The objects within a frame may be freely aliased and different frames are not required to be disjoint. Therefore, dynamic frames are well suited to verify structures with complex sharing. The dynamic frames technique is the basis of the Dafny verifier [Leino 2010].

Concurrency. The notion of ownership extends elegantly to concurrency: If each memory location is owned exclusively by a method execution, it is not possible for two threads to access the same location concurrently. Concurrent separation logic [O’Hearn 2004] uses this observation to verify race-free concurrent programs. Synchronization is supported by allowing locks and other synchronization primitives to own memory. Acquiring a lock transfers ownership to the acquiring thread and releasing the lock transfers them back to the lock.

Concurrent separation logic naturally supports concurrent reading by employing fractional ownership [Boyland 2003] but prevents concurrent writing. While this is often desirable, there are many concurrent data structures that do not use synchronization, for instance, implementations of locks and lock-free data structures such as the Treiber stack. To handle this form of fine-grained concurrency, the community developed an abundance of concurrent separation logics [Feng et al. 2007, Vafeiadis 2007]. A major theme in these logics, first proposed by Dinsdale-Young et al. [Dinsdale-Young et al. 2010], is to associate memory regions with a transition system that prescribes the legal actions on the memory region. This transition system enables rely–guarantee reasoning [Jones 1981] by constraining both a thread and its environment. The transitions are often guarded by (exclusive or shared) resources, which allows fine-grained control over the transitions a thread may make.

Reasoning about fine-grained concurrency is further complicated by the emergence of weak-memory architectures, which sacrifice sequential consistency in favor of performance. Weak memory models support multiple access modes for shared variables with different synchronization guarantees. Recent separation
logics support some of them [Vafeiadis and Narayan 2013], but more work is required to develop program logics for the full complexity of weak memory. Reasoning about weak memory is also a major theme in model checking [Atig et al. 2011, Alglave et al. 2013] and static analysis [Kusano and Wang 2017].

Rich properties. Despite the tremendous progress in developing and advancing the theory of program verification, there are still major challenges ahead for the verification of entire systems. We give three examples in the following.

(1) An important class of properties are hyperproperties, which relate multiple program executions. Typical use cases include the verification of secure information flow, determinism, and monotonicity. Hyperproperties can be verified using relational logics [Benton 2004] or by reducing them to trace properties of so-called product programs (for instance, through self-composition [Barthe et al. 2011]). However, more research is needed to develop sufficiently expressive and automated verification techniques for hyperproperties. In particular, relational logics are not automated by off-the-shelf verification tools, and product programs have limitations when the different executions of a program are not aligned, for instance, when they have substantially different control flow. Moreover, verification techniques for hyperproperties of concurrent code typically impose strong restrictions to ensure that the timing of an execution does not affect the thread schedule.

(2) Program verification is often used to prove security (rather than correctness) properties of systems. Since many systems rely on cryptography, for instance, to authenticate accesses, it is important to verify both the correctness of crypto-primitives and their application in crypto-protocols. Verifying cryptographic operations is fundamentally different from standard correctness verification: instead of proving possibilistic properties (is it possible to reach a given state), verified crypto proves probabilistic properties (how likely is it for an attacker to obtain a secret key, assuming certain mathematical hardness results). Verification of cryptographic operations and protocols is now possible due to substantial theoretical advances [Barthe et al. 2009] and has been applied to verify entire cryptographic systems [Protzenko et al. 2019]. However, due to their intricate math, the resulting proofs are still difficult to automate.

(3) Many modern systems include machine learning, for instance, to have computer vision, to classify data, or to tune parameters. One of the main challenges for the verification of such systems is that it is unclear what properties to specify and prove. For instance, how should one specify a classifier that distinguishes pictures of cats from pictures of dogs? Existing work on
verifying machine learning applications, thus, focuses on properties other than functional correctness. For instance, Gehr et al. [Gehr et al. 2018] verify the robustness of neural networks by showing that small perturbations in the inputs lead to only small perturbations in the results. More work is needed to fully understand how to prove complex correctness and security properties of machine learning components.

6.4 Tools

The VSI has always emphasized the importance of tool development for the overall success of the project as well as the necessity to work on a wide variety of tools that strike different trade-offs between expressiveness, automation, and soundness. Over the past fifteen years, the community has made tremendous progress across the whole spectrum. We highlight some of the main developments in the following.

SMT solvers. Much of the recent progress in practical program verification is owed to the stunning improvement of SMT solvers. SMT solvers power many program verification tools, for instance, by discharging the verification conditions of a deductive verifier, by performing predicate abstraction in a model checker, or by generating test inputs for concolic testing. At the kick-off of the VSI, Simplify [Detlefs et al. 2005] was still the dominant SMT solver. More recent solvers, most notably Z3 [de Moura and Björner 2008] and CVC4 [Barrett et al. 2011], offer much better performance, support more theories, and provide improved quantifier instantiation strategies, all of which are crucial for program verification. This development was made possible by algorithmic improvements, for instance, by extending conflict-driven clause learning to infinite domains such as integers and by using model-based theory combination instead of the classical Nelson–Oppen combination method.

As can be witnessed in the annual SMT-COMP event, competition among automatic theorem provers is fierce. While Z3 and CVC4 are currently the most widely used solvers in program verification, other tools excel as well in important sections of the competition. Among them is Yices [Dutertre 2014], a powerful SMT solver for quantifier-free formulas, as well as Vampire [Kovács and Voronkov 2013], a first-order prover that has recently been extended to support theories.

Intermediate verification languages. Analogous to the architecture of compilers, many modern verification tools are implemented as a sequence of translation steps from programs and specifications to proof obligations in a suitable logic. A front-end encodes the input program and specification into an intermediate verification language, whereas a back-end automates the proof search, typically using SMT solvers. This architecture has significant advantages compared to monolithic
designs: the intermediate language decouples the front-end (which encodes the semantics of the input program) from the back-end (which automates the proof search). Thus, back-ends can be re-used across many verifiers, which drastically reduces the effort of building verifiers and facilitates the use of a portfolio of reasoning approaches.

This architecture was pioneered by the Boogie [Leino 2008] and Why3 [Filliâtre and Paskevich 2013] intermediate verification languages, which received the 2019 CAV Award. Boogie is a simple imperative programming language; programs consist of a background theory (declaring types, functions, and axioms to provide the mathematical vocabulary for programs and specifications) as well as a sequence of procedures (each equipped with a contract). Boogie provides a modular verification condition generator, but also supports verification by means of (bounded) inlining. It is used by a variety of program verifiers including Corral [Lal et al. 2012], Dafny [Leino 2010], SMACK [Carter et al. 2016], Spec# [Barnett et al. 2011], SymDiff [Lahiri et al. 2012], and VCC [Cohen et al. 2010]. Why3 is a first-order functional language where functions are equipped with contracts and mutable state is encoded using mutable fields in records. Why3 targets a wide range of provers and is used, for instance, in Frama-C [Blanchard et al. 2019] and Krakatoa [Filliâtre and Marché 2007].

Both Boogie and Why3 express specifications in first-order logic but have no direct support for specialized logics such as separation logic with its support for framing and concurrency (see Section 6.3). To fill this gap, the Viper infrastructure [Müller et al. 2016] natively supports separation logic features such as (fractional) permissions, predicates, separating conjunction, and magic wands, which facilitate the encoding of rich separation logics. Viper provides back-ends for verification condition generation (via a translation into Boogie) and symbolic execution. It powers, among others, the Nagini verifier for Python [Eilers and Müller 2018], the Prusti verifier for Rust [Astrauskas et al. 2019], and the VerCors verifier for Java [Blom et al. 2007].

Automated verification. Program verification tools fall into three major groups, based on the trade-off they strike between expressiveness, automation, and the size of the trusted code base. Interactive verifiers often offer the full expressive power of higher-order logic for programs and specifications, but require users to provide detailed proofs whose validity is checked by the tool. At the other end of the spectrum, model checkers and static analyses aim at full automation, but typically focus on specific program properties. Automated verifiers attempt to strike a balance: they are able to verify complex properties such as functional correctness with good automation (using SMT solvers), but require programmers to provide
annotations such as loop invariants. We highlight some noteworthy automated verifiers here and discuss the other two categories below.

During the course of the VSI, automated verifiers have developed from early prototypes into mature, powerful tools with many applications in industrial software development, research, and teaching. For instance, Dafny [Leino 2010] is the result of co-designing a programming language and an automated verifier. It provides an expressive class-based language with many features that simplify the development of specifications and proofs, such as inductive and co-inductive datatypes, induction and co-induction, lemmas, comprehensions, and calculational proofs. Dafny compiles to several other languages including C# and JavaScript, which allows verified Dafny code to interact with other components. Dafny is used in several large-scale verification projects, in particular Ironclad Apps [Hawblitzel et al. 2014], IronFleet [Hawblitzel et al. 2015] and Armada [Lorch et al. 2020]. It is also used in numerous classes to teach program verification.

F* [Swamy et al. 2016] aims at combining the automation provided by SMT solvers with the expressiveness of interactive verification. It provides a dependently typed functional programming language. F* can be parameterized with different monads, for instance, to provide predicate transformers for higher-order programs or stateful programs with different state models (e.g., encoding dynamic frames or a region-based heap model). In contrast to most purely SMT-based verifiers, F* provides more control over the proof search, which is useful for intricate verification problems. Verified F* code can be extracted to F#, OCaml, and other languages. The F* system is used, for instance, in Microsoft's project Everest [Delignat-Lavaud et al. 2017], which develops verified TLS implementations including the underlying cryptographic primitives.

Prusti [Astrauskas et al. 2019] is a recent verifier for Rust, which leverages Rust's ownership type system to simplify verification and reduce the annotation overhead for programmers. Starting from a well-typed program in safe Rust, Prusti automatically extracts a core proof in separation logic. This core proof contains all assertions (such as predicate definitions, function specifications, and loop invariants) required to prove memory safety in separation logic. They also enable framing, which makes it easy to extend the core proof to richer properties such as functional correctness. In particular, programmers can express specifications using simple assertions in Rust without being exposed to the intricacies of a complex program logic.

**Interactive verification.** Interactive verification tools typically embed program logics into interactive theorem provers such as Coq [Bertot and Castéran 2004], Isabelle [Paulson 1994], or PVS [Owre et al. 1992]. Besides offering the expressive power of
higher-order logic, these tools are foundational in the sense that their proofs rest on a small number of logical axioms and can, thus, be checked by a small, trusted proof checker. In particular, soundness of the program logic is typically proved within the theorem prover, which increases confidence in the verification results. Interactive verification tools achieve a certain degree of automation by using specialized proof tactics, but require substantially more user input than SMT-based verifiers.

The Verified Software Toolchain (VST) [Cao et al. 2018], developed within the DeepSpec project, is an interactive verification tool for a subset of concurrent C based on Coq. VST uses a higher-order separation logic, which is proved sound with respect to the C semantics used by the CompCert compiler [Leroy 2009]. Automation is provided by the Floyd tactics library. VST offers a formalism called interaction trees to specify interactions of a system with its environment, for instance, to verify a distributed system.

**Automatic verification.** Fully automatic verifiers use a broad range of techniques to prove program properties without requiring any guidance from the user beyond the properties to be checked. In the following, we highlight tools that illustrate this variety.

Corral [Lal et al. 2012], a verifier based on Boogie, uses verification condition generation and SMT solving. To avoid the need for procedure specifications and loop invariants, Corral performs bounded verification through stratified inlining. That is, it unrolls loops and inlines procedure calls up to a given depth, which is increased after each successful verification attempt until an error is found. This approach cannot give guarantees for unbounded executions, but will detect violations of safety properties eventually. Corral is one of the verifiers used by Microsoft’s Static Driver Verifier [Lal and Qadeer 2014].

In the area of model checking, a particularly noteworthy development during the course of the VSI was IC3 [Bradley 2011] and its variant PDR [Eén et al. 2011]. These algorithms provide an efficient way of checking reachability properties by incrementally learning inductive invariants. A key difference compared to other symbolic model checking techniques is that IC3 and PDR do not unroll loops, but instead learn new facts locally. IC3 is, for instance, used by SeaHorn [Gurfinkel et al. 2015].

Another form of bounded checkers that emerged during the course of the VSI are automatic test case generators based on concolic testing (also called dynamic symbolic execution). These tools alternate concrete and symbolic executions of the code to be checked and use an SMT solver to obtain test inputs that will take paths in the concrete execution that have not been previously explored, thereby increasing branch coverage. One of the key strengths of concolic testing is the ability to fall
back on concrete values if the SMT solver is unable to solve a symbolic constraint. This allows concolic testing tools to handle complex constraints involving operations that are difficult to reason about symbolically (such as hash functions). There are several powerful concolic testing tools, for example, SAGE for x86 assembly [Godefroid et al. 2012] and KLEE for C [Cadar et al. 2008]. Concolic testing checks bounded executions, but can provide sound guarantees for code that does not contain input-dependent loops [Christakis and Godefroid 2015].

Last but not least, there are several new static analyzers. Facebook’s Infer tool [Calcagno and Distefano 2011] analyzes Java programs and detects potential null pointer dereferences and memory leaks; it has recently been extended to also support C, C++, and Objective-C. Infer performs a compositional shape analysis based on separation logic and bi-abduction. Zoncolan [Distefano et al. 2019] is another static analysis developed at Facebook, which performs abstract interpretation to detect security vulnerabilities in very large HACK (a dialect of PHP with gradual typing) code bases. Zoncolan tracks dataflows in the program and compares them to a set of rules that indicate potential security or privacy violations. This allows developers to customize the analysis by adding and removing such rules.

### 6.5 Experiments

The third pillar of the VSI, besides theory and tools, is to evaluate the developed verification techniques and tools through experiments, with the goal of demonstrating the capabilities of verification technology and to identify open problems. This pillar has materialized in two major directions. First, the community has organized regular competitions for many kinds of verification approaches; these competitions are useful to compare techniques and tools, serve as a source of benchmark problems, and help motivate developers to push the limits of their tools. Second, several projects have developed entire systems that are formally verified. These projects are especially useful to demonstrate to other communities that verification is no longer a theoretical endeavor but has become a viable option to build provably correct and secure systems. We summarize some key achievements in both directions in this section.

### 6.5.1 Competitions

Since 2002, the SAT Competition at the Theory and Applications of Satisfiability Testing conference has provided an annual snapshot of the state of the art in SAT solving and, thereby, documented the impressive progress in this field. Inspired by this great success, various verification communities started to organize regular competitions. The Satisfiability Modulo Theories Competition (SMT-COMP) [Barrett et al. 2005] has been organized annually in conjunction with the SMT workshop since 2005. SMT-COMP poses a rich set of challenge problems in many
different tracks to assess the capabilities of SMT solvers for different logical fragments, solving modes (e.g., incremental vs. non-incremental), and functionalities (e.g., determining unsat cores for unsatisfiable formulas). SMT-COMP attracts around 25 contestants every year and is, thus, able to provide a representative overview of the state of the art.

Similar in spirit to SMT COMP is the Competition on Software Verification (SV-COMP), which focuses on automatic verification (mostly through model checking and static analysis). SV-COMP has been organized annually since 2012, affiliated with TACAS. Similarly to SMT-COMP, SV-COMP includes benchmark problems from several different categories, grouped by the properties to be checked (e.g., reachability properties, concurrency safety) It also includes benchmarks in different programming languages including C and Java. With 28 participating tools in the most recent competition, SV-COMP provides a comprehensive comparison of automatic verification tools.

SMT-COMP and SV-COMP focus on automatic tools and assess their performance by measuring the correctness of the results as well as the run time required. In contrast, the results in automated and interactive verification depend on both the capabilities of the used tool and the skills of the participating users. Therefore, the VerifyThis Competition focuses on users, letting them freely choose the tools they want to employ. VerifyThis has been held since 2011 [Klebanov et al. 2011], usually as a live event in conjunction with ETAPS. It typically provides three benchmark problems and requires participants to develop formal specifications and verify those against an implementation (which is often provided as pseudocode). The quality of the solutions is assessed by a jury after a discussion with the team that developed them. Instead of ranking teams or tools, VerifyThis awards prizes in different categories such as ‘best overall team’ and ‘most distinguished tool feature.’ VerifyThis attracts around 10 to 15 participating teams, which need to attend physically.

Competitions are generally considered immensely useful by the verification community. Their benefits go far beyond the immediate results of comparing tools. They serve as a collection of benchmark problems, which are used for the evaluations in countless publications. Moreover, they drive progress by recognizing successful tool builders, which is especially important since much of the work that is required to advance verification tools is not easily publishable.

### 6.5.2 Verified Systems

What the community has achieved in terms of verifying real code bases goes far beyond ‘experiments’ and has exceeded the expectations we had at the beginning of the VSI. The first lighthouse projects—the formally verified operating system
seL4 [Klein et al. 2009] and the formally verified C compiler CompCert [Leroy 2009]—inspired the community to undertake ambitious verification projects. Major achievements so far include the Ironclad [Hawblitzel et al. 2014] and IronFleet [Hawblitzel et al. 2015] projects, which pioneered the verification of distributed systems; Microsoft's project Everest [Delignat-Lavaud et al. 2017], which includes (among many other results) a formally verified crypto library; and the results of DeepSpec, which integrate the foundational verification of different system layers, including the operating system kernel CertiKOS [Gu et al. 2016].

Even though these projects demonstrated clearly that the verification of entire system implementations is now feasible, it is not clear yet how best to go about such a project. Key decisions include, for instance, how to structure the verification process (e.g., top–down or bottom–up), which languages to use to express specifications and implementations, and which tools to employ (e.g., interactive or automatic theorem provers). In the discussion here, we consider approaches for the verification of implementations rather than designs or algorithms. Moreover, we focus on deductive verification, aiming at proving complex properties such as full functional correctness and security. There are also many success stories of algorithmic verification such as model checking and abstract interpretation.

Verification process. The main verification task of the CompCert compiler is to prove that each compiler pass preserves the semantics of the input program. These passes are implemented and verified within Coq; the executable implementation is then obtained using Coq’s code extraction feature. A similar top–down approach is taken by CertiKOS. seL4 combines top–down and bottom–up development by inserting a Haskell prototype between the high-level specification and the high-performance C implementation. The proof then consists of two refinement properties: the imperative implementation refines the Haskell prototype, which in turn is a refinement of an abstract specification. Ironclad and IronFleet follow a bottom–up approach. In Ironclad, both the implementation and specification of the system are written and verified in Dafny. To remove the compiler from the trusted code base, the compiled x86 code is verified against a translation of the specification. Non-interference is proved separately using SymDiff [Lahiri et al. 2012], also in a bottom–up fashion. IronFleet, on the other hand, organizes the verification in layers, connected by refinement proofs, and employs a separate argument based on reduction [Lipton 1975] to reason about concurrency. Similarly to Ironclad, project Everest expresses specifications, imperative implementations, and proofs within one programming language, here F* [Swamy et al. 2016].

Even though existing verification projects apply very different approaches, they all have two essential insights in common. First, all of them carefully decompose
the verification task into smaller, manageable components. In some cases, this composition follows the structure of the code, in others the structure of the properties to be checked—both are far superior to a monolithic approach. Second, in all of the above projects, the implementation is either derived from the proof or developed together with the proof. It seems essential that software engineers and verification experts work hand in hand to make the implementation verifiable and to elicit the correctness arguments to be used.

Notations. Most of the above verification projects use a single notation to express implementations, specifications, and proofs. Such a homogeneous approach simplifies the application of tools and avoids subtle inconsistencies between notations. However, the chosen notations differ greatly. seL4, CompCert, and DeepSpec perform the development entirely within an interactive theorem prover such as Isabelle or Coq. This choice offers them the full expressive power of higher-order logic to express non-trivial properties such as the semantics of a programming language. Proofs are then developed mostly manually. By contrast, Ironclad, IronFleet, and Everest express their development in programming languages designed for verification (here, Dafny and F*). Specifications are expressed using contracts such as pre- and postconditions, and proofs are written in code, for instance, by using ghost procedures to encode lemmas and loops to encode induction. This approach allows verification engineers to work directly on the level of the implementation, without using cumbersome embeddings. On the other hand, using richer specification formalisms (such as TLA in IronFleet) requires encodings into the verification language.

The choice of notation is tightly coupled with the tool support, in particular, the degree of automation. Moreover, different notations put the focus of the verification efforts on different aspects: Developments within an expressive logic tend to emphasize properties more than implementations and might be best suited to prove complex properties, whereas developments within programming languages emphasize the implementations and might be better when dealing with intricate implementations. Therefore, the choice of formalism is often a trade-off between expressiveness and productivity, which needs to be resolved based on the requirements of the verification project at hand.

Tool support. Several successful verification projects have been carried out using interactive theorem provers. Besides expressiveness, these tools offer full control over the proof development. Moreover, their proofs are foundational, that is, all properties are ultimately derived from a small number of axioms such that they can be checked independently by simple, trustworthy checkers. In particular, the formal development in projects such as DeepSpec includes the soundness proof
of the program logic, which minimizes the assumptions and code base that must be trusted. seL4 is developed in Isabelle; CompCert and DeepSpec both use Coq. In contrast, Ironclad, IronFleet, and Everest use automated verifiers based on SMT solvers (Dafny and F*), resulting in higher automation; the proof overhead of interactive verifiers is usually between 10 and 20 lines of proof per line of code, whereas projects using automated verifiers report an overhead factor of 5. The relatively higher degree of automation of automated verifiers increases productivity; however, it can complicate debugging of verification errors and authoring of proofs that the tool cannot find automatically. Enhanced control over the proof effort was one of the main reasons project Everest chose F*. Most verification projects seem to prefer a homogeneous environment, where all proofs are carried out in one tool. IronFleet is an exception; it uses a dedicated verifier to prove non-interference.

A recent empirical study on the correctness of verified systems [Fonseca et al. 2017] systematically tested several verified systems. According to this study, the vast majority of detected bugs occurs at the boundaries of the verification project, in particular, at the interface to unverified code (such as the API of the operating system). Interestingly, none of the bugs were caused by an unsoundness in the verification tool itself, which might suggest that foundational proofs are desirable but not essential. Expressiveness, automation, and user interaction are, thus, the main criteria for choosing the tools to be employed in a verification project.

**The Road Ahead**

Systems have become complex and interconnected, not only with each other but with systems in the physical world for banking, commerce, medicine, entertainment, communication, transportation, energy, and much more. Complex protocols and data interchange formats form the glue supporting the connections between physically and virtually distributed systems. If software is the beating heart of these systems and services, the consequences of losing intellectual control over this software can be costly and even catastrophic. Formal specifications are needed to cover platforms, policies, and protocols. Security policies need to be vetted for unintended feature interaction. The code itself needs to be analyzed for properties beyond the absence of run-time errors such as buffer overflows and uncaught exceptions to cover information flow, deadlocks, race conditions, and timing properties. This is a world in which there is an increasing role for mechanized formal methods at all stages of the software lifecycle. On the supply side of the equation, we can expect to see formal techniques embedded into and seamlessly integrated with design tools and integrated development environments (IDEs) supporting such activities as test generation, symbolic exploration, extended static checking, verification, and synthesis. The power and scale of these
tools will expand significantly through theoretical advances as well as rigorous experiments and ambitious case studies. In the next fifteen years, we can expect software verification to become a routine part of software development for safety-critical and security-critical systems. The confluence of the increasing complexity and connectedness of the software infrastructure, fast improving technology, and a well-trained workforce are creating fertile conditions for realizing Hoare’s vision of a thriving verified software ecosystem in the decades ahead.

References


A. Biere and R. Bloem (Eds.). 2014. *Computer Aided Verification—26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014* (July 18–22,
114  Chapter 6  The First Fifteen Years of the Verified Software Project


References


