International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-024-00765-y

GENERAL

Special Section: Challenges of Software Verification

Challenges of software verification: the past, the present, the future

Pietro Ferrara[1] · **Vincenzo Arceri**[2] · **Agostino Cortesi**[1]

Accepted: 21 August 2024

Abstract
Software verification aims to prove that a program satisfies some given properties for all its possible executions. Software evolved incredibly fast during the last century, exposing several challenges to this scientific discipline. The goal of the "Challenges of Software Verification Symposium" is to monitor the state-of-the-art in this field. In this article, we will present the evolution of software from its inception in the 1940s to today's applications, how this exposed new challenges to software verification, and what this discipline achieved. We will then discuss how this chapter covers most of the current open challenges, the possible future software developments, and what challenges this will raise in software verification.

Keywords Program verification · Formal methods · Static analysis

1 Introduction

The advent of computers in the 20th century revolutionized the world. Computers allowed to automatize, speed up, and improve most of the tasks of everyday life. A computer is a piece of hardware that automatically executes a series of computations. Humans define these computations through software. Through the scientific and technological advances of the last centuries, we have nowadays a well-defined definition of software. It consists of programs written in source code (an intermediate structured language staying in the middle between the code executed by a machine and the human language) that are Turing complete [52]. Hardware instead is abstracted by the von Neumann architecture [55], that is, a central processing unit performing mathematical operations, a control unit that allows to check conditions and iterate operations, a memory where data is stored, input and output.

Software verification is a scientific field whose ultimate goal is to verify that a given software satisfies a given property. The theory of computer science proved that it is not possible to prove a nontrivial property of software by running it. Therefore, an approximation of what the program computes is needed, and several different approaches have been developed. These approaches led to various prototypal, scientific, or industrial tools. Following the classification proposed by Cousot [11], software verifiers are the ones that "are semantics-based, sound, and precise". Developing a tool that satisfies these constraints requires a relevant formalization effort (often leading to new scientific advancements) and a massive implementation effort.

Software verification is tightly coupled with software: the analyzed programs and proven properties depend on the developed software type. Therefore, it is only possible to outline the history of software verification by carefully analyzing how software evolved. In particular, different software applications led to different needs regarding the quality, efficiency, safety, and security of the software itself, leading to different software verification challenges over the decades.

The main goal of the "Challenges of Software Verification Symposium" (CSV) is to discuss both solutions and upcoming challenges in this field of research. The symposium is organized annually in Venice by the Software and System Verification laboratory of Ca' Foscari University [57]. In this introduction to the special chapter about CSV 2023, we outline the evolution of software verification since the beginning. In particular, we identified several different phases. For each phase, we discuss (i) the technologies and programming languages that appeared in that phase, (ii) what types of software were developed, (iii) what challenges this raised to software verification, and (iv) what scientific advancements

✉ P. Ferrara
pietro.ferrara@unive.it

V. Arceri
vincenzo.arceri@unipr.it

A. Cortesi
cortesi@unive.it

[1] Ca' Foscari University, Venice, Italy

[2] University of Parma, Parma, Italy

Published online: 30 August 2024

were achieved. We then outline what trends we observe today in software development and what this might imply for software verification.

Our survey aims to minimize and simplify how software and software verification evolved during the last century.

Sections 2–6 discuss different stages in software evolution and verification. Section 7 will introduce the "Challenges of Software Verification Symposium" and how the different contributions of this chapter are. Section 8 will present what, in our opinion, might be the disruptive trends of the next few years in this field. Finally, Sect. 9 will conclude.

2 The early era of software

The first question we must pose to ourselves is: When exactly did software appear? To answer such a question, we must precisely define what software means. Two milestones in the theory of Computer Science are the Turing machine [52] and the von Neumann architecture [55]. Software requires (i) a general purpose processor (that is, a Turing machine), and (ii) a memory to store data and programs (that is, a von Neumann architecture) [73].

While the idea of computer algorithms was introduced already in the 19th century by Alda Lovelace, only in 1945 did the first software, as we understand it nowadays, appear. ENIAC (Electronic Numerical Integrator and Computer) [70] is considered the first programmable computer. A few years later, Tom Kilburn wrote what is considered the first program to calculate the highest factor of 2^{18}. From then on, software took off.

2.1 Technologies and programming languages

In the early stages of software, computers were programmed at a very low level. The first programs in the 1950s were developed directly in binary code. The first compilers were developed in the 1950s, and several programming languages (PL) appeared in the following decade:[1]

- Fortran [71], a PL specifically targeting numerical and scientific computations, was invented in 1954 at IBM by John Backus, and its first compiler was released in 1957,
- LISP [76] was conceived at MIT by John McCarthy in 1958 and released in 1960, and
- COBOL [69], another PL that became mainstream and is still widely used today, was released in 1959.

The components common to most proposed approaches were nested blocks (and procedures in particular) and the lexical scoping of each block's component.

[1] While the history of PLs is out of the scope of this paper, we recall just the main milestones to analyze what challenges the different programming patterns present to the software verification discipline.

2.2 Type of software

The existing technologies allowed the development of mostly small programs compared to the size of software we are used to nowadays, and where it was hard to reason modularly. In addition, deploying such programs was extremely hard and time-consuming. While the first computers, such as ENIAC, needed the programs to be physically developed and deployed through wiring, during the second half of the 1950s, software, as we understand it today, made its appearance. In particular, such software was (i) written in a programming language (like those mentioned above) that was humanly readable and (ii) compiled into executable binaries. The code was usually stored in punch cards, and then loaded into mainframe computers to execute it.

2.3 Challenges

In such a context, program robustness was essential: redevelopment, redeployment, and execution of the software were quite time-consuming, and computational resources were extremely expensive and inefficient. The computational model of the programming languages mentioned above was quite simple, but already Turing complete: reasoning about loops and nested blocks exposed several relevant challenges. On the other hand, these programs were extremely small, and the pace of development was extremely low.

2.4 Scientific advancements

In 1949, Alan Turing introduced the idea of proving the correctness of (a part of) a program [51]. While still quite preliminary, this work showed the need to reason about a computer program to prove some properties mathematically. In the following decade, a more rigorous and systematic approach about formal reasoning on computation was developed in the community. The most notable results in this context were produced by John McCarthy [35, 36], who developed a mathematical theory of computation between the end of the 1950s and the beginning of the 1960s.

3 The revolution of embedded software

The Apollo program is widely known as the program that brought humanity to the moon in 1969. However, it represented a turnaround in the field of software development. In particular, the landing modules contained software that managed the modules and their interaction with the physical environment. This software had considerable size, and this experience led to the birth of the term "software engineering". The development and deployment of this software was as previously described, that is, it was concretely wired and required several weeks to be deployed.

3.1 Technologies and programming languages

The C programming language [27] was introduced in 1972 [66], and it is still the most popular programming language for embedded software nowadays. C is a general-purpose programming language applied to almost all computer systems. From an embedded system point of view, the main advantage of C is that it provides a low-level view of the computer architecture, allowing direct access to the memory through free pointers. Overall, C programs are quite efficient and are well-suited for embedded systems where resources are typically quite constrained.

3.2 Type of software

The software of the Apollo program was just the first disruptive application of what is now embedded software. This type of software has been around for over half a century, and its technologies evolved. By embedded software, we mean software that (i) is embedded inside a physical device, (ii) interacts with the physical environment through sensors (reading data from the environment) and actuators (modifying the device or the physical environment in some ways), and (iii) is safety critical. Since the 1980s, embedded software has become pervasive in everyday life, and nowadays, almost all devices contain some embedded software. In addition, they are usually connected to the Internet, leading to the so-called Internet-of-Things (IoT).

3.3 Challenges

The growing size of software, coupled with the fact that safety-critical software might cause huge damages and deaths, raised severe challenges regarding the reliability of software. Since an error (also known as a bug) might lead to a disaster, it became hard to reason deeply about software because of its dimensions, which challenged the scientific communities towards developing formal techniques that help prove software's safety. The computational model provided by popular programming languages (and C in particular) made it hard to precisely reason about how the memory is managed: since a program can freely access it, ensuring that a memory access is well formed (e.g., it accesses an area of memory previously allocated and belonging to the program) became hard.

3.4 Scientific advancements

In parallel with the Apollo mission, the scientific community invested relevant efforts in the formalization of novel mathematical approaches to formally reason on the program. At the end of the 1960s, Floyd introduced assertional reasoning on programs [18], while Hoare proposed the axiomatic semantics [21] (also known as Floyd–Hoare logic). These approaches still require some forms of manual annotation/proof about the program, but they are quite more automatized than approaches proposed in the previous decades. In addition, they allow expressing and proving any property expressed in the logic adopted to specify assertions about the program.

Data [28] and control [1] flow analyses made an analogous step in a similar direction. These approaches to software verification focus on very specific properties, and apply some forms of approximation when reasoning about the programs. Since the 1970s, they have been widely applied by all compilers [30, 31].

4 The rise of micro and personal computers

The first microcomputers, as opposed to mainframes, appeared in 1975, and they allowed having a computer on a common desk. This opened the door to computers for individuals, which in later years were called personal computers (PCs). This revolution moved computational capabilities from research laboratories to everyday life, opening the door to endless new opportunities.

4.1 Type of software

First, the appearance of operating systems, and Unix in particular in 1971, allowed individuals to use computers. This step was needed to allow the development of desktop applications that provided some functionalities to common users, such as word processors and spreadsheets. Generally speaking, PCs allow installing different types of software that interact with the hardware and, therefore, the user through the operating system. Therefore, PCs brought, as a logical consequence, packaged software, that is, generic software that was not targeting a specific customer or hardware. Before the advent of the Internet, PCs were standalone and did not communicate with each other.

4.2 Technologies and programming languages

In this period, object-oriented programming languages appeared. The main novelty of this approach was that it provided various primitives to encapsulate the code and allow for the polymorphic behavior of the code. The main goal was to expose only a minimal interface of the program, abstracting away implementation details to structure the software and allow its proper reuse. Smalltalk [79] was the first object-oriented programming language proposed in 1972, while a decade later C++ [67] appeared. C++ is still one of the most popular programming languages today. Moreover, the rise of expert systems generated the request to separate the concepts of logic and control in programming, giving rise to declarative programming languages [54].

4.3 Challenges

The size of software increased over time. This led to the problem of modular reasoning: since it was no longer possible to reason about the program as a whole, dividing it into distinct capsules was needed. This challenge was the main reason for the introduction and adoption of object-oriented programming languages, but it later raised further challenges regarding how to automatically reason about such programs.

On the other hand, an error in desktop applications was not as relevant as in embedded software. In fact, while a failure in safety-critical embedded software might have catastrophic consequences, a crash in a desktop application might lead, at most, to the loss of some data. Therefore, the robustness of PC software was considered less relevant than that of embedded software.

4.4 Scientific advancements

The foundations of automatic software verification were laid during these decades. In particular, the formal approaches previously introduced were further developed and generalized.

In particular, Edsger Dijkstra reformulated the Floyd–Hoare logic in the predicate transformer semantics [15]. This gave the basis for developing software verification through theorem proving in the following decades.

At the end of the 1970s, the theory of abstract interpretation [12, 13] was introduced by Patrick and Radhia Cousot. This theory generalized existing program analyses (like the control mentioned above and data flow analyses) into a generic mathematical framework that allows (i) defining the concrete semantics of a programming language, (ii) defining different types of abstractions, and (iii) proving the soundness of the abstractions with respect to the concrete semantics.

A few years later, model checking [8, 9, 16, 39] was introduced. This approach checks if a finite-state model of a given (usually physical) system satisfies a given specification. Such an approach was also widely applied to software [26, 45, 47].

Even if the logic and constraint languages, which had generated great expectations for their mathematical cleanliness, have not had the desired success, they have provided a generous workbench for the design of mathematically well-founded static software analysis techniques [42].

5 Moving into the new millennium: Internet and web applications

The advent of the Internet opened the door to the development of software in distributed systems on a large scale. While previously PCs were disconnected nodes, each running some locally installed software, PCs connected to the Internet exchanged data with remote machines, with software running on both sides.

5.1 Type of software

The Internet led to the development of the so-called Web applications. The main idea is that most of the software runs on a machine (the server) while another machine (the client) accesses the application through some specific software (the browser). Usually, both software comprise more and more code written by third parties (libraries) that implement some standard generic functionalities.

5.2 Technologies and programming languages

During the 1990s, all the currently most popular programming languages were introduced. In particular, Java [74] and C# [68] appeared in the second half of the 1990s (1996 and 2000, respectively). While providing different technological stacks, both these approaches developed several tools for Web applications, such as servlets and applets in Java and ASP.NET in C#. These programming languages were mostly used on the server side since this software implements the application's business logic, and therefore, it requires advanced constructs to encapsulate implementation details.

Another programming language that gained popularity in this context is PHP [77], a scripting language targeting explicitly Web development server-side. JavaScript [75] adopted a similar approach client-side. These technologies usually combine the implementation of the Web application's business logic with its graphical user interface.

5.3 Challenges

While until desktop applications most of the verification challenges were focused on finding runtime errors that might lead to software failures, computer systems that comprised different machines communicating with each other opened the door to new scenarios. In particular, one of the main concerns was the presence of vulnerabilities, that is, logic weaknesses that might be exploited by an attacker for various purposes.

In this context, a full ecosystem of professionals and institutions aimed at classifying vulnerabilities appeared. The main vulnerabilities involved some injections (e.g., the attacker's input might flow into SQL queries without being properly escaped, thus allowing the attacker to modify the data freely), cross-site scripting, and data leakages.

The field of embedded software experienced an evolution towards more secure and reliable solutions in this period.

In particular, certification authorities enforced various regulations about adopting safety-critical embedded software. This process started a couple of decades after embedded software appeared, and different certifications targeted different industrial sectors. For instance, ISO/DIS 26262 [59] regards the functional safety of road vehicles, and the sixth part regards how automotive software should be developed. In this context, the standard recommends the adoption of software verification techniques. Similar guidelines exist in RTCA DO-178C for embedded software in airborne systems and in IEC 62304:2006 [60] about medical device software.

5.4 Scientific advancements

First, the practical applications of the theories developed in the previous decades led to several industrial applications of software verification techniques. In particular, the safety-critical embedded software field experienced a proliferation of tools based on formal methods. Tools like Grammatech [72] CodeSonar, Polyspace [78], and ASTREE [14] (mostly based on abstract interpretation techniques) were widely applied to prove that safety critical software was runtime error-free. Other tools, like BLAST [20], applied model checking to software verification instead. In contrast, other relevant efforts led to different program verifiers and programming languages (such as Spec# [5]) based on SMT solvers.

A major result in 2005 was the development of a formally verified optimizing compiler called CompCert [33]. Software verification guarantees that such a compiler will produce executable code that respects the semantics of the source code while applying several optimizations on the produced binary code.

Another relevant effort was focused on the formalization and development of different types of generic analyses that might be interesting in different contexts for software verification [2, 24, 29, 32, 43, 46, 48]. In parallel, several libraries and tools [53, 64] provided an implementation of the most popular software verification frameworks through static analysis.

Last but not least, Tony Hoare launched the Software Verification grand challenge in 2003 [22]. The main idea of this challenge is to focus long-term research efforts in the field of software verification towards "the construction and application of a verifying compiler that guarantees correctness of a program before running it". This led to a community effort targeting the application of various software verification techniques to mainstream programming languages to build up verifiers that could be used in industrial contexts.

6 The present: distributed applications

During the 2010s and 2020s, software became more and more pervasive. Smartphones and other mobile personal devices have become part of everyday life. Web applications were adopted to perform several business-critical activities, such as online banking, and privacy-sensitive functionalities, such as geographical tracking. Embedded software evolved into IoT by connecting physical devices to the Internet and allowing them to communicate with each other, with other mobile devices, and with back-end services. Last but not least, starting in the early 2000s, various types of social networks appeared. Nowadays, almost anyone worldwide regularly uses these Web platforms, mostly for entertainment purposes.

6.1 Type of software

Various software architectures appeared, from service-oriented architectures to microservices and serverless applications [40]. The main goal of these distributed systems is to provide services that are (i) scalable (they can respond to an increasing number of requests over time), (ii) elastic (they can support sudden peaks of requests), and (iii) robust (if a node fails, the system is resilient). These architectures require to couple code with its deployment, since different software units are deployed into independent components that collaborate to offer users different services.

Blockchain introduced a different type of distributed architecture. While the first idea of blockchain was designed in the 1980s and 1990s [19], the applications and popularity of this technology rose after 2008 when Bitcoin, the first cryptocurrency, was invented based on such a technology. However, the first generation of blockchain did not contain any software. Instead, smart contracts were introduced by the Ethereum blockchain in 2015. The main goal of this type of software is to "automatically execute, control, or document events and actions according to the terms of a contract or an agreement" [80].

6.2 Technologies and programming languages

Most of the technologies that appeared recently evolved from previous programming languages into technologies coupled with a deployment system. For instance, Java Enterprise Edition, and later Jakarta and Java Spring, coupled the Java programming language with servers (e.g., Tomcat) to deploy a software system that executes the code to answer HTTP requests.

Scripting languages like Python and JavaScript became mainstream in this context. In particular, technologies like Node.js and Angular are nowadays probably the most popular

solutions for developing back- and front-end Web applications, respectively. For similar purposes, the Python ecosystem was enriched with various libraries (such as FastAPI and Django). While it is impossible for now to depict a definitive scenario of these technologies, most of them enrich the original programming language with various types of annotations to specify service-oriented functionalities of the software.

Several programming languages (such as Solidity [81]) were introduced in the context of smart contracts. Tightly bound with libraries to manage blockchain transactions, these programming languages were usually quite limited to prevent behaviors not allowed in a blockchain context, like nondeterminism. Another approach was to use mainstream programming languages (like the Go programming language in Hyperledger Fabric) with some restrictions and checks about what they could do.

6.3 Challenges

The splitting of applications into many [micro] deployable units led to further complexity from a software verification point of view.

First of all, each unit might be developed in a different programming language and rely on different technological stacks. For instance, the Sock Shop [65] represented a demo application well underlying the intrinsic technological variety in microservices' applications. It mixes Java, Go, JavaScript code, and Spring, Go kit, and Node.js technologies. Rigorous software verification requires relevant efforts to define the semantics of a single programming language and technology. Such a variety, therefore, represents a huge challenge to this scientific field.

In addition, since each unit is deployed independently, a global view of the software system's status might be missing. Different versions of different software system services might coexist and interact. On the one hand, focusing on a unique version of the whole system would not allow one to verify the different possible deployments. On the other hand, assuming nothing about what other microservices do would be too conservative in practice.

The law advancements pose the last challenge that might be an opportunity for software verification. For instance, the European General Data Protection Regulation [23] imposes various measures in order to protect sensitive data in information systems. While the legal approach is more focused on the process than on the technical tools applicable in such a context, it usually suggests the adoption of technologies that help improve software quality from different points of view. This context is quite similar to what happened to the certification of safety-critical embedded systems in different markets a few decades ago, as explained in Sect. 5.3.

Smart contracts usually deal with business-sensitive operations since a relevant part of this technology has been applied to develop and maintain cryptocurrency. In addition, once released, they usually persist in the blockchain and cannot be modified. Therefore, verifying that this software is correct and does not contain security vulnerabilities is of primary interest to avoid potential disasters, such as the Ethereum DAO attack [82].

6.4 Scientific advancements

The challenges previously raised by Web applications pushed for the formalization and development of novel analyses starting from information flow [41]. In particular, taint analysis [50] (roughly, a relaxation of information flow analysis that omits implicit flow) was widely applied to detect security vulnerabilities. While not all approaches in this field can be interpreted as software verification (since most of them are unsound or based on catching some of the vulnerabilities instead of proving the correctness of the software [10]), this was probably the most relevant attempt of the software verification community in this field. Using similar approaches, software verification later focused on mobile applications [4] and different types of programs written in JavaScript [25], Python, and other programming languages. In addition, some tools [56, 61] provided implementation of core algorithms of static analysis that could be applied to different programming languages, with the final goal of developing a multi-language software verifier.

A clearly perceived trend by the software verification community is that modern programming languages comprise more and more dynamic features to allow the development of complex programs in the distributed system environment. For this reason, features like multithreading, reflections, and annotations are usually ignored by sound static analysis since their rigorous treatment would lead to too complex formalization and implementation or too imprecise analyses. The concept of soundiness [34] was then introduced to allow the scientific community to state the exact boundaries of software verifiers clearly.

A proliferation of results was also experienced in verifying smart contracts [49]. While several different approaches to software verification have already been applied to all the most popular blockchain platforms, this effort is still ongoing and evolving quickly.

7 Challenges of software verification symposium 2023

A first edition of the "Challenges of Software Verification Symposium" was held on May 20, 2022 [62]. In the morning, Ca' Foscari University awarded Prof. Patrick Cousot a PhD in Computer Science Honoris Causa. Instead, the symposium took place in the afternoon with 15 invited talks. An extended

version of some of the talks was later published in SpringerNature [3].

Following the great success of this event, the "2nd Challenges of Software Verification Symposium" was organized on May 25–26, 2023 [63]. The scope of the symposium covered theoretical results in the field of software verification, their practical applications, novel and innovative tools, and their impact of software verification in software engineering and DevOps practices. The symposium comprised 18 half-an-hour-long invited talks with speakers from various countries (Italy, France, Germany, the United States, Israel, Norway, and Belgium). These talks covered all the main topics of software verification and its open challenges. The sessions comprised theoretical and practical aspects of static analysis, abstract interpretation, software engineering, and security. About 50 scientists participated to the event.

This chapter contains the scientific contribution of six invited papers that formalize and extend the results presented during some of the CSV 23 talks. These articles cover all the main recent challenges of software verification, and some of them pose the basis for future ones.

In particular, Olivieri and Spoto [38] tackle smart contracts (as discussed at the end of Sect. 6.3) in the context of software verification. In particular, it clarifies the notion of blockchain-oriented software, and it provides an overview of the verification challenges it raises.

Jensen et al. [17] advances the state-of-the-art of information flow analyses (that we introduced in Sect. 6.4) by extending Hoare logic (mentioned in Sect. 3.4) to binary PER-based predicates for relating observationally equivalent states.

Monniaux [37] presents several solutions and still open challenges in verified static analysis. This represents a further application of the concept of the formally verified compiler (discussed in Sect. 5.4): like a compiler, the implementation of a sound static analyzer needs to be formally verified in order to ensure that the properties it verifies are effectively satisfied by the analyzed software.

Since the invention of the abstract interpretation theory (recalled in Sect. 4.4), various abstract domains have been formalized, mostly to approximate numerical information computed by programs. In this context, relational and nonrelational domains appeared for different reasons. A third level (weakly relation domains) was invented to obtain efficient and precise abstractions. Seidl et al. [44] explores how this idea can be generalized to other types of (not necessarily numerical) information.

Bodei et al. [6] present the result of the formal analysis of an automotive software module designed to ensure confidentiality, integrity, and authentication at the same time for traffic exchanged over CAN protocol (that is, among different physical components of a car). Such effort represents the follow-up of the results obtained a couple of decades ago on safety-critical embedded software discussed in Sect. 5.4. Nowadays, vehicles are usually connected to the Internet through the infotainment system. Therefore, the CAN protocol might cause unsafe and insecure traffic. Nowadays, applying software verification to such protocols is of primary relevance to ensure the physical safety of modern vehicles.

Brodo et al. [7] apply formal methods to reaction systems, a qualitative computational formalism inspired by biochemical reactions in living cells. This work shows that formal methods initially developed purely for software verification can be far beyond this field, finding interesting and practical applications in other scientific disciplines.

8 The future: what is next?

While knowing what will happen in the future is impossible, and even guessing it might be very dangerous, we believe some trends that will revolutionize the software development world are already established. The software verification community already started to focus its efforts on these topics.

The first disruptive trend is the adoption of generative artificial intelligence techniques during software development. This led to AI-based code assistants suggesting code to professional developers and guessing their needed features. Gartner predicts that "by 2027, 50% of enterprise software engineers will use ML-powered coding tools, up from fewer than 5% today" [58]. While the adoption of these techniques appeared only recently, we believe this is just the next step in a trend we have observed in software development during the last decade. Before the advent of ChatGPT and similar tools, it was common for developers to google for a solution, usually landing in a Stackoverflow discussion providing the answer. At the end of this process, some potentially unsafe and unreliable code was copied and pasted into the software project. From a high-level software verification point of view, the two approaches do not differ much: some code produced from unreliable sources is almost mindlessly inserted into a software project. If, on the one hand, this opens the door to even more insecure and unsafe software, it represents an opportunity for software verification.

However, the artificial intelligence revolution we are experiencing these years goes far beyond code generation. Nowadays, all the top-tier conferences in software engineering (such as ICSE, ASE, and FSE) and programming languages (such as PLDI, POPL, and OOPSLA) contain contributions adopting some form of machine learning or artificial intelligence. While such effort rarely falls into the field of software verification as defined in Sect. 1, its fast development might lead to novel insights and contributions that will have an impact in such a context.

Other emerging trends in technology involve quantum computing, various aspects of cybersecurity, novel approaches to software development and deployment, etc.

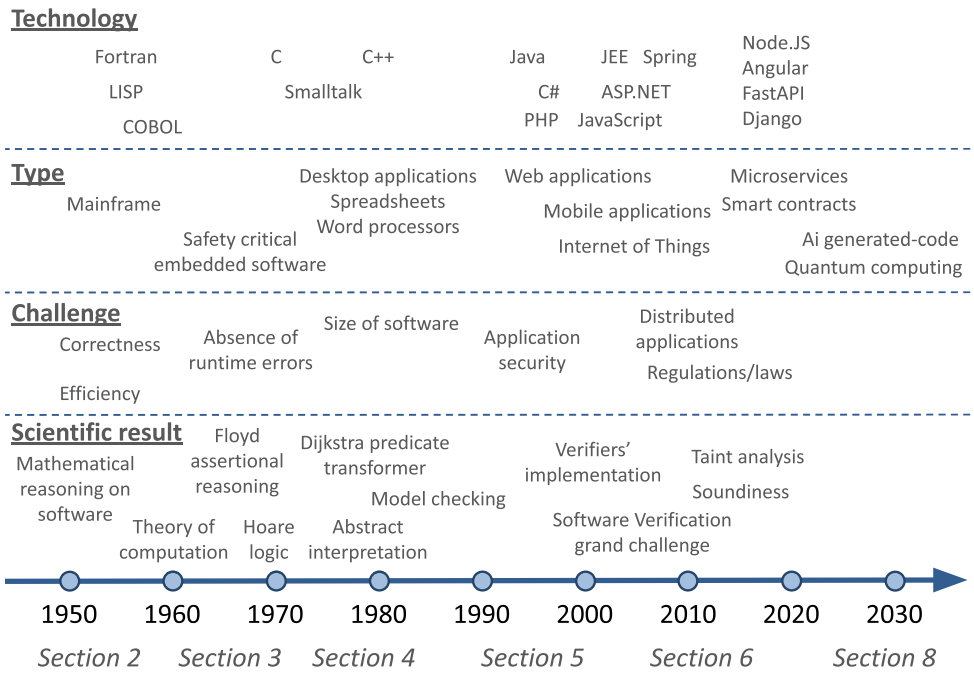

Fig. 1 Timeline of software verification challenges

While all these topics are interesting regarding software verification, it is still hard to predict which ones will experience broad adoption in the future and become relevant and the main trend topic for software verification.

9 Summary and conclusion

Figure 1 summarizes the content of this paper into a timeline. At a very high level, the idea of software, as we intend it today, appeared during the 1950s, when the first programmable mainframes appeared. The limited resources and difficulties in deploying a program emphasized efficiency and correctness. In parallel, the first attempts to mathematically formalize what software is and can do appeared.

The following decades experienced the appearance of safety-critical embedded software, where the complete absence of runtime errors is crucial. However, it took quite some time to find complete, practical solutions to this need in the scientific field of software verification. First, different formal approaches were introduced. Then, several attempts were made to make them useful in practice. Only after about three decades, software verifiers were able to prove the absence of runtime error on industrial software. In particular, this required first defining the bases of computation, then developing approaches able to soundly prove properties on software, and finally, properly implementing it.

From the 1980s onward, software became more and more pervasive. Desktop, Web, mobile, cloud, and distributed applications today allow performing a variety of tasks in everybody's life every day. While the quality of such software is crucial only sometimes, its security becomes increasingly important, and different approaches to verify the absence of the main application vulnerabilities have been developed.

Finally, during the last months we experienced the revolution of AI-generated code. While it is still hard to predict how pervasive this will be, there is no doubt that using third-party code in software projects is a common practice today. Verifying this software by focusing on the external code is still an open challenge. Last but not least, quantum computing is becoming more and more visible and hopefully useful. This might be yet another software revolution that would push software verification to deal with this innovative model of computation.

Acknowledgement Work partially supported by SERICS (PE00000014 – CUP H73C2200089001) under the NRRP MUR program funded by the EU – NGEU, and by iNEST – Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 49 1.5) NextGeneration EU (ECS_00000043 – CUP H43C22000540006).

References

1. Allen, F.E.: Control flow analysis. ACM SIGPLAN Not. **5**(7), 1–19 (1970)
2. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD thesis, Datalogisk Institut, Københavns Universitet (1994)
3. Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (eds.): Challenges of Software Verification 2022. Intelligent Systems Reference Library, vol. 238. Springer, Berlin (2023)
4. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. ACM SIGPLAN Not. **49**(6), 259–269 (2014)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: an overview. In: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, pp. 49–69. Springer, Berlin (2004)
6. Bodei, C., De Vincenzi, M., Matteucci, I.: Formal analysis of an AUTOSAR-based basic software module. Int. J. Softw. Tools Technol. Transf. (2024, in press)
7. Brodo, L., Bruni, R., Falaschi, M., Gori, R., Milazzo, P., Montagna, V., Pulieri, P.: Causal analysis of positive reaction systems. Int. J. Softw. Tools Technol. Transf. (2024, in press)
8. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logic of Programs, pp. 52–71. Springer, Berlin (1981)
9. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
10. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 196–206 (2007)
11. Cousot, P.: Principles of Abstract Interpretation. MIT Press, Cambridge (2021)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252 (1977)
13. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 269–282 (1979)
14. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyzer. In: Proceedings of the 14th European Symposium on Programming (ESOP 2005), pp. 21–30. Springer, Berlin (2005)
15. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
16. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. In: Automata, Languages and Programming: Seventh Colloquium, Noordwijkerhout, the Netherlands, July 14–18, 1980, Proceedings 7, pp. 169–181. Springer, Berlin (1980)
17. Filinski, A., Larsen, K.F., Jensen, T.: Axiomatising an information flow logic based on partial equivalence relations. Int. J. Softw. Tools Technol. Transf. (2024, in press)
18. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of the American Mathematical Society Symposia on Applied Mathematics, vol. 19, pp. 19–31. Springer, Berlin (1967)
19. Haber, S., Stornetta, W.S.: How to Time-Stamp a Digital Document. Springer, Berlin (1991)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In: Model Checking Software: 10th International SPIN Workshop, Portland, OR, USA, May 9–10, 2003. Proceedings 10, pp. 235–239. Springer, Berlin (2003)
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
22. Hoare, T.: The verifying compiler: a grand challenge for computing research. J. ACM **50**(1), 63–69 (2003)
23. Hoofnagle, C.J., Van Der Sloot, B., Borgesius, F.Z.: The European Union general data protection regulation: what it is and what it means. Information & Communications Technology Law **28**(1), 65–98 (2019)
24. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)
25. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: Static Analysis: 16th International Symposium, SAS 2009. Los Angeles, CA, USA, August 9–11, 2009, Proceedings 16, pp. 238–255. Springer, Berlin (2009)
26. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys (CSUR) **41**(4), 1–54 (2009)
27. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall, New York (1988)
28. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 194–206 (1973)
29. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Compiler Construction: 4th International Conference, CC'92. Paderborn, FRG, October 5–7, 1992, Proceedings 4, pp. 125–140. Springer, Berlin (1992)
30. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pp. 224–234 (1992)
31. Knoop, J., Rüthing, O., Steffen, B.: Optimal code motion: theory and practice. ACM Trans. Program. Lang. Syst. **16**(4), 1117–1155 (1994)
32. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst. **18**(3), 268–299 (1996)
33. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**, 363–446 (2009)
34. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015)
35. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, Studies in Logic and the Foundations of Mathematics, vol. 26, pp. 33–70. Elsevier, Amsterdam (1959). https://doi.org/10.1016/S0049-237X(09)70099-0
36. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960)
37. Monniaux, D.: Pragmatics of formally verified yet efficient static analysis, in particular for formally verified compilers. Int. J. Softw. Tools Technol. Transf. (2024, in press)
38. Olivieri, L., Spoto, F.: Software verification challenges in the blockchain ecosystem. Int. J. Softw. Tools Technol. Transf. (2024, in press)
39. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: International Symposium on Programming, pp. 337–351. Springer, Berlin (1982)
40. Richards, M., Ford, N.: Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media (2020)
41. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
42. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**(1–2), 131–170 (1996)

43. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002)
44. Seidl, H., Erhard, J., Tilscher, S., Schwarz, M.: Non-numerical weakly relational domains. Int. J. Softw. Tools Technol. Transf. (2024, in press)
45. Steffen, B.: Data flow analysis as model checking. In: International Symposium on Theoretical Aspects of Computer Software, pp. 346–364. Springer, Berlin (1991)
46. Steffen, B.: Property-oriented expansion. In: Static Analysis: Third International Symposium, SAS'96, Aachen, Germany, September 24–26, 1996, Proceedings 3, pp. 22–41. Springer, Berlin (1996)
47. Steffen, B., Claßen, A., Klein, M., Knoop, J., Margaria, T.: The fixpoint-analysis machine. In: CONCUR'95: Concurrency Theory: 6th International Conference, Philadelphia, PA, USA, August 21–24, 1995, Proceedings 6, pp. 72–87. Springer, Berlin (1995)
48. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 281–293 (2000)
49. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Computing Surveys (CSUR) **54**(7), 1–38 (2021)
50. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. ACM SIGPLAN Not. **44**(6), 87–97 (2009)
51. Turing, A.M.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69. Univ. Math. Lab, Cambridge (1949)
52. Turing, A.M., et al.: On computable numbers, with an application to the entscheidungsproblem. J. Math. **58**(345–363), Article ID 5 (1936)
53. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot – a Java bytecode optimization framework. In: Proceedings of CASCON'99, p. 13. IBM Press, Raleigh (1999)
54. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4), 733–742 (1976)
55. von Neumann, J.: First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993)

Sitography

56. Antoine Mine: MOPSA - Modular Open Platform for Static Analysis. https://mopsa.lip6.fr/ (2024). [Online; accessed 05-July-2024]
57. Ca' Foscari University of Venice: The Software and System Verification group. https://ssv.dais.unive.it (2024). [Online; accessed 05-July-2024]
58. Gartner: Gartner Hype Cycle Shows AI Practices and Platform Engineering Will Reach Mainstream Adoption in Software Engineering in Two to Five Years. https://www.gartner.com/en/newsroom/press-releases/2023-11-28-gartner-hype-cycle-shows-ai-practices-and-platform-engineering-will-reach-mainstream-adoption-in-software-engineering-in-two-to-five-yearsl (2024). [Online; accessed 05-July-2024]
59. ISO: 26262-6 Road vehicles - Functional safety. https://www.iso.org/standard/68388.html (2024). [Online; accessed 05-July-2024]
60. ISO: IEC 62304 Medical device software — Software life cycle processes. https://www.iso.org/standard/38421.html (2024). [Online; accessed 05-July-2024]
61. Luca Negrini: LiSA - A Library for Static Analysis. https://lisa-analyzer.github.io/ (2024). [Online; accessed 05-July-2024]
62. The Software and System Verification group at Ca' Foscari University of Venice: 1st Challenges of Software Verification Symposium (CSV 22). https://unive-ssv.github.io/events/2022/05/20/csv.html ((2022)). [Online; accessed 05-July-2024]
63. The Software and System Verification group at Ca' Foscari University of Venice: 2nd Challenges of Software Verification Symposium (CSV 23). https://unive-ssv.github.io/events/2023/05/25/csv.html ((2023)). [Online; accessed 05-July-2024]
64. WALA: T. J. Watson Libraries for Analysis. https://github.com/wala/WALA (2024). [Online; accessed 05-July-2024]
65. Weavworks: Sock Shop : A Microservice Demo Application. https://github.com/microservices-demo (2024). [Online; accessed 05-July-2024]
66. Wikipedia: C — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language) (2024). [Online; accessed 05-July-2024]
67. Wikipedia: C++ — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C%2B%2B (2024). [Online; accessed 05-July-2024]
68. Wikipedia: C Sharp — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_Sharp_(programming_language) (2024). [Online; accessed 05-July-2024]
69. Wikipedia: COBOL — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/COBOL (2024). [Online; accessed 05-July-2024]
70. Wikipedia: ENIAC — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/ENIAC (2024). [Online; accessed 05-July-2024]
71. Wikipedia: Fortran — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Fortran (2024). [Online; accessed 05-July-2024]
72. Wikipedia: GrammaTech — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/GrammaTech (2024). [Online; accessed 05-July-2024]
73. Wikipedia: History of software — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/History_of_software (2024). [Online; accessed 05-July-2024]
74. Wikipedia: Java — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Java_(programming_language) (2024). [Online; accessed 05-July-2024]
75. Wikipedia: JavaScript — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/JavaScript (2024). [Online; accessed 05-July-2024]
76. Wikipedia: LISP — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/LISP (2024). [Online; accessed 05-July-2024]
77. Wikipedia: PHP — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/PHP (2024). [Online; accessed 05-July-2024]
78. Wikipedia: Polyspace — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Polyspace (2024). [Online; accessed 05-July-2024]
79. Wikipedia: Smalltalk — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Smalltalk (2024). [Online; accessed 05-July-2024]
80. Wikipedia: Smart Contract — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Smart_contract (2024). [Online; accessed 05-July-2024]
81. Wikipedia: Solidity — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Solidity (2024). [Online; accessed 05-July-2024]
82. Wikipedia: The DAO — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/The_DAO (2024). [Online; accessed 05-July-2024]