cyberagentur

ECOSYSTEM FOR TRUSTWORTHY IT

LOS 1: FORMAL VERIFICATION OF COMPLEX SOFTWARE SYSTEMS – A STUDY

Bernhard Beckert,
Oliver Denninger,
Jonas Klamroth,
Max Scheerer,
Jörg Henß

February 2023

Herausgeberin:
Agentur für Innovation in der Cybersicherheit GmbH

Disclaimer

Die hier geäußerten Ansichten und Meinungen sind ausschließlich diejenigen der Autorinnen und Autoren und entsprechen nicht notwendigerweise denjenigen der Agentur für Innovation in der Cybersicherheit GmbH oder der Bundesregierung.

Diese Studie wurde durch die Agentur für Innovation in der Cybersicherheit GmbH beauftragt und finanziert. Eine Einflussnahme der Agentur für Innovation in der Cybersicherheit GmbH auf die Ergebnisse fand nicht statt.

Impressum

Herausgeberin: Agentur für Innovation in der Cybersicherheit GmbH
Große Steinstraße 19, 06108 Halle (Saale), Germany
E-Mail: kontakt@cyberagentur.de
Internet: www.cyberagentur.de
Twitter: https://twitter.com/CybAgBund

Erscheinungsdatum: 12.07.2023
Redaktion: Abteilung Sichere Systeme, Referat Sichere Hardware und Lieferketten

— Formal Verification

of Complex Software Systems – A Study

Bernhard Beckert, Oliver Denninger, Jonas Klamroth,
Max Scheerer, Jörg Henß

February 2023

FZI Forschungszentrum Informatik

Contents

1. Introduction

Formal software verification methods can prove both functional and non-functional properties of software systems. Unlike methods such as testing or debugging, they can guarantee that a software system does not exhibit behavior violating a given specified property. Thus, they achieve a higher level of trust. Formal verification of software has a long history in research, but has never made the transition to broad industrial application. This study presents the state of the art in formal verification research for software systems and provides recommendations for improving the state of the art and its industrial application.

The focus of this study is on the vision of full verification of complex software systems, including the operating system kernel, drivers, system services, middleware services, and applications. In the past, this vision has been addressed by projects such as VeriSoft [Beckert and Moskal 2010] and DeepSpec [Appel et al. 2017]. However, it's still an open question how to scale verification to this level.

Chapter 2 introduces functional and non-functional properties of software systems. Proving these properties is the goal of formal methods. Chapter 3 presents the current state of the art on formal methods. Selected tools are introduced and their maturity for industrial use is evaluated. Chapter 4 discusses the limits of scalability of the presented methods and tools. Chapter 5 recommends actions to improve the state of the art and to increase broad industrial application.

2. Dependable Software Systems

In this chapter, we discuss dependable software systems. More specifically, we present the notion of dependability in software systems based on Sommerville's definition [Sommerville 2011]. We chose Sommerville because his definition of dependability is widely used in the literature and also in teaching. Afterward, we present a classification of formal properties used in formal verification. Finally, we illustrate how formal properties (belonging to individual property classes) are used to verify dependability properties of software systems. Dependability classes as represented here are not formally defined, and thus their definitions or even the selection of dependability classes may vary in the literature. As we will discuss later, this is however no obstacle to formal verification efforts as those dependability properties are never verified directly but rather indirectly.

2.1. Functional and Non-Functional Requirements

Before we discuss the details of dependable software systems, we have to distinguish two fundamental concepts, namely *Functional* and *Non-Functional Requirements*. The very first phase of each software development process is the requirements engineering phase, where a collection of requirements are extracted that a software system has to satisfy. Basically, requirements can be divided into two main categories, namely functional and non-functional requirements. Functional requirements capture all requirements which describe the desired operations and functions provided by the software system, e.g., the software system must provide login functionalities. In contrast, non-functional requirements (also referred to as quality requirements) refer to requirements of qualitative nature, e.g., performance or security. A non-functional requirement could, for example, require that the response time of a software system must always be less than a particular threshold or that down-times must be low.

2.2. Properties of Dependable Software Systems

We consider dependability of software systems according to Sommerville [Sommerville 2011] who defined as a property that reflects the degree of trustworthiness one assigns to a software system. Hereby, trustworthiness refers to the confidence a user associates with the software system; that is, whether the system behaves as expected and does not malfunction. Basically, Sommerville divides dependability into four principal dimensions, namely Availability, Reliability, Safety, and Security (see Figure 2.1). The individual dimensions are defined as follows:

- **Availability:** System availability is defined as the probability of a system to be up and running to deliver its provided functionalities on request.
- **Reliability:** Contrary to availability, system reliability is defined as the probability that the system provides its functionalities as expected or as defined in the system specification.

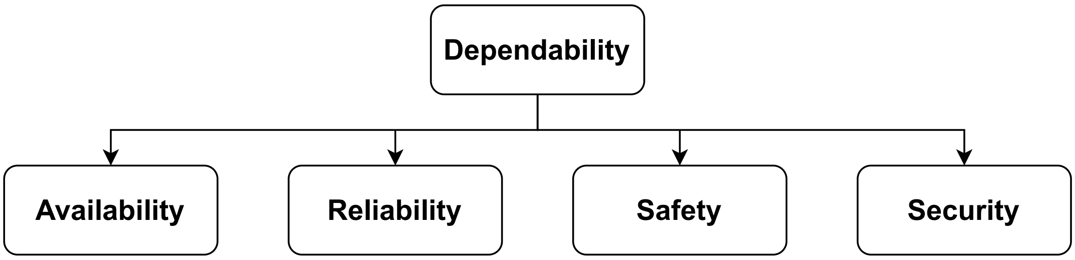

Figure 2.1.: Dependability properties [Sommerville 2011].

- **Safety:** System safety refers to the ability to operate without catastrophic failure, i.e., the system will not cause damage to people or its environment.

- **Security:** System security is considered as the ability of a system to protect itself against accidental or deliberate intrusion.

Note that there are further possible properties one can associate with dependability such as Repairability, Survivability, Error Tolerance, and Maintainability. However, in this study, we focus on the principal dimensions of dependability. Moreover, we focus on reliability, safety, and security. Availability of a system is oftentimes achieved by architectural means (e.g., redundancy) rather than with formal methods. We still argue that certain aspects of availability may be subject to formal verification. Consider for example the integration of a load balancer in order to handle a high number of requests. This is an architectural decision, however it still would be necessary to verify the functional correctness of this load balancer in order to get the guarantees required in the formal setting.

This is a perfect example of how the different dependability classes are not easily separable from one another. At the very least, reliability cannot be achieved if the system is not safe or secure, but there are less clear examples of how these classes can overlap. Another typical example is memory safety. A system with code containing buffer overflows is potentially unsafe, but this can also lead to a security breach. So, verifying the absence of buffer overflows can be considered both safety and security related.

2.3. Verifying Formal and Informal Properties

In this section, we present different property classes into which formal properties (e.g., *Spatial Separation* [Baumann et al. 2011]) are classified. In principle, we distinguish between two different types of properties, namely *Informal-* and *Formal Properties*. Informal properties include desirable properties or requirements for an entire system; formal properties describe (code-near) properties that are required for an informal property to be true. That is, a high-level informal property is realized by (low-level) formal properties that need to be verified for the informal property to be true. For example, the memory (or spatial) separation property is shown in [Baumann et al. 2011] by verifying *Function Contracts* and *Invariants* (including *Ownership*). Generally, a property (whether formal or informal) has a property class. A property class groups properties with the same underlying concern. In the following, we present several property classes based on [Beckert and Hähnle 2014] and discuss formal and informal properties afterwards.

Finally, note that in the literature, no general classification or distinction between properties and property classes is made. However, it is important to distinguish properties because some describe *what* to verify (i.e., informal properties) and others describe *how* to verify (i.e., formal properties).

2.3.1. Property Classes

In [Beckert and Hähnle 2014], several property categories or classes are presented, namely *Safety*, *Liveness*, *Uniform*, *Generic*, *Lightweight*, *Functional Correctness* and *Relational* properties. Each property class includes various manifestations of properties considered as instance of a property class, e.g., spatial separation property is an instance of the uniform property class. In the following, we briefly describe the distinct property classes taken from [Beckert and Hähnle 2014].

- **Safety and Liveness:** Safety and liveness properties are interconnected in that they are both defined over abstract (state based) system models. Safety properties are defined to check that a critical system state is never reached; liveness properties verify that the system will finally reach a desired system state. There are numerous examples of safety and liveness property instances (commonly described by temporal logic) since the definition of a state is fairly domain-specific. Note that the above safety definition differs from the safety definition of [Sommerville 2011] in that the former is formally defined and is therefore subject to formal verification. Moreover, the notion of safety for verification mostly relates to parts of the system (i.e., components or sub-systems); on the contrary, safety according to Sommerville always refers to a property of the entire system. However, bare safety properties do not guarantee the safety of a system according to Sommerville because when considering safety at the system level, other properties such as liveness or relational properties must be taken into account as well. Instead, they subsume a collection of properties associated with safety in the formal verification community.

- **Generic and Uniform** Generic and uniform properties refer to the absence of typical errors such as buffer overflows or division by zero, and are thus fundamental for functional as well as non-functional requirements such as security aspects.

- **Lightweight:** Lightweight properties refer to simple and less expressive properties. Examples include type checks and simple Boolean expressions (such as assertions) without considering quantifiers or other higher-order logic features.

- **Relational:** Relational properties relate the results of different system calls or different versions of the system to each other to verify specific functional requirements.

- **Functional Correctness:** Functional correctness means that a system satisfies its functional specification for all runs, i.e., all possible inputs and initial states.

Each property class contains numerous properties or instances which are either formal or informal. Moreover, an informal property is verified by proving a set of formal properties. More specifically, while informal properties describe desirable system-level properties, formal properties describe specific (code-near) properties that determine how the informal property can be verified. In the following sections, we present informal and formal properties. We discuss which property classes are formal or informal, and thus categorize whole sets of properties as formal or informal. Finally, we provide an overview of established and fairly known properties.

2.3.2. Informal Properties

Informal properties describe desirable properties of a (sub-)system. Moreover, informal properties are defined in a declarative manner, i.e., they can be understood as a system requirement. However, the declarative nature of these properties complicates the verification process. Instead of directly verifying an informal property, one makes use of several formal properties (embedded in a strategy) to prove the informal property. For example, an operating system kernel (preferably) implements a *Separation Kernel* architecture for which the properties *Data Separation*, *Information Flow*, *Fault Isolation*, and *Time Separation* (which are all examples of informal properties) have been verified. However, the four properties cannot be proved directly, but only by considering formal properties that are systematically included in a proof strategy.

We consider all properties that represent instances of the classes uniform, safety, liveness and relational as informal properties. This results mainly from the observation that in literature properties of said classes are never verified directly but by using formal properties. Table 2.1 provides an overview of informal properties well-known in the verification community.

2.3.3. Formal Properties

While informal properties are typically high-level or system-level properties, formal properties tend to be closer to the code of the system. The verification of informal properties requires formal properties that specifically determine how an informal property can be proved. Note however that the amount of formal properties needed to realize one informal property is not related to its complexity. The fact that informal properties are realized through formal properties is illustrated in Figure 2.2.

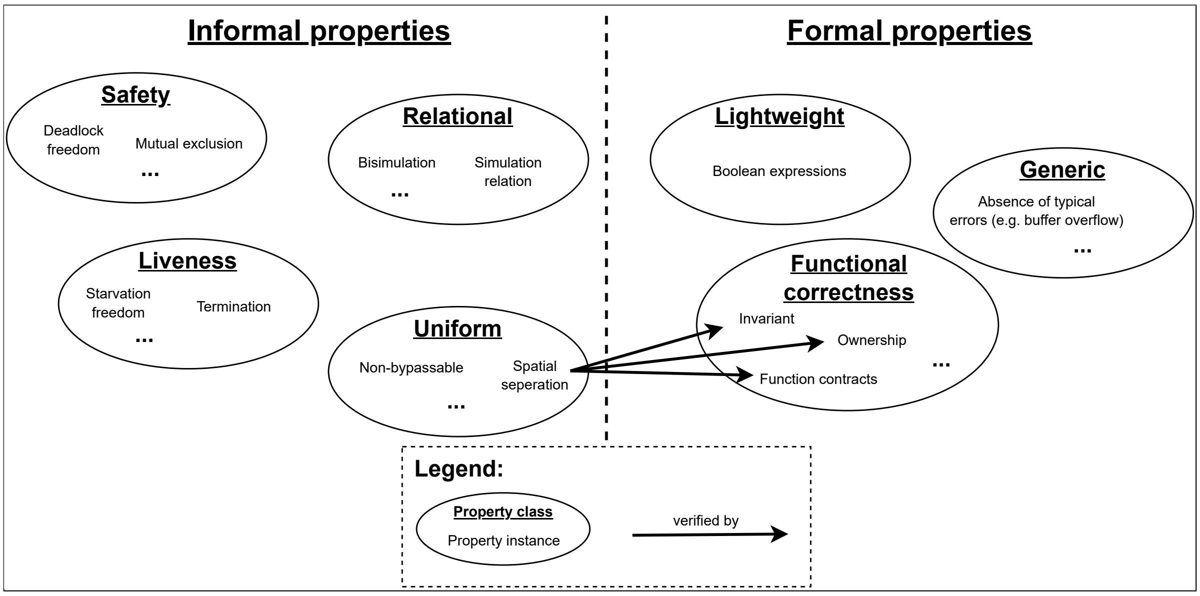

Figure 2.2.: Verifying property class instances with formal properties.

As an example, recall the work of [Baumann et al. 2011]; here the authors verified spatial separation (informal property) of a kernel memory manager by using several formal properties, namely *Function Contracts*, *Invariants*, and *Ownership*.

We consider properties belonging to the classes generic, lightweight and functional correctness as formal properties. In the following we discuss three formal properties that are commonly known and used to verify informal properties, namely *Function Contracts*, *Invariants*, *Ownership*, and *Coupling Invariant*

- **Function contract:** Function contracts are attached to a function or method by defining pre-

Informal properties		
Class	**Instance**	**Description**
Safety	Mutual exclusion	No two processes execute in critical sections simultaneously [Alpern and Schneider 1985].
Safety	Deadlock freedom	No deadlocks occur.
Liveness	Starvation freedom	A process makes progress infinitely often [Alpern and Schneider 1985].
Liveness	Termination	Completion of the final instruction, i.e., the program terminates [Alpern and Schneider 1985].
Liveness	Guaranteed service	Every service request is satisfied eventually [Alpern and Schneider 1985].
Relational	Simulation relation	One system is a refinement of another [Beckert and Hähnle 2014].
Relational	Bisimulation relation	Two systems exhibit the same behavior [Beckert and Hähnle 2014].
Relational	Non-interference	The system does not reveal information about the initial value [Beckert and Hähnle 2014].
Uniform	Data separation	A partition (which holds several applications) is implemented as a separated resource [Zhao et al. 2017].
Uniform	Information flow	Information that flows from one partition to others have an authenticated source and authenticated recipients [Zhao et al. 2017].
Uniform	Fault isolation	The resulting damage by a fault is limited [Zhao et al. 2017].
Uniform	Temporal separation	Components can share the same physical resource in different time slices without affecting each other [Zhao et al. 2017]
Uniform	Non-bypassable	Security functions cannot be circumvented [Vanfleet et al. 2005].
Uniform	Evaluatable	Security functions are small and simple such that correctness can be proved [Vanfleet et al. 2005].
Uniform	Always-invoked	Security functions are invoked at any time [Vanfleet et al. 2005]
Uniform	Tamper proof	Security functions and their data cannot be modified without authorization [Vanfleet et al. 2005]
Uniform	Authentication	Certain events are always happening in a specific order [Chaki and Datta 2009]
Uniform	Secrecy	Refers to the inability of an attacker to calculate a specific message [Chaki and Datta 2009]
Lightweight	Code assertion	Boolean expressions of the target programming language [Beckert and Hähnle 2014]

Table 2.1.: Examples of established and well-known informal properties for each property class.

and postconditions. A precondition describes the required state (or properties of the state) of the system that must hold before executing the method. A postcondition describes the properties that must be satisfied after executing the method (starting in state where the preconditions hold true). Additionally, some languages allow defining frameconditions in function contracts that describe which part of the memory may be modified or read by that function.

- **(General) Invariant:** An invariant defines assertions or properties that must hold true throughout the execution of the system, e.g., $x > 0$ in any state (or at least a previously defined set of states). Invariants can be manually specified and are therefore arbitrarily complex, but can also include properties such as type checks. A very complex example of type checks is **Ownership**, which refers to conditions where an object of a program is tied to exactly one owner; only the owner is allowed to access the object. Ownership properties are especially important when verifying security properties.
- **Coupling invariant:** Coupling invariants describe properties specific to relational verification where two executions (possibly of the same system) are set into relation. Possibly the most important instance of relational verification in this work is *Refinement*. Refinement is a verification method (discussed in more detail in Section 3.1.3) in which the functionality of the system is specified (e.g., by first-order logic) and gradually transformed by a series of refinement steps into an executable system model. Crucially, each refinement step preserves the behavior of the refined system model such that the functionality of the system is guaranteed. Coupling invariants spell out the relationship between runs of the more abstract and the more concrete system and are an important tool to prove correct refinement.

Table 2.2 provides an overview of common examples of formal properties (including generic properties). Note that we deliberately do not discuss generic properties. The term generic property generally refers to common code bugs, such as buffer overflows or null pointer exceptions. Proving functional correctness, i.e., that a system behaves as specified, often naturally ensures the absence of typical bugs (generic properties). Conversely, it is often much easier to verify generic properties because they do not require formal specifications specific to the software, but this does not give any guarantee of functional correctness.

Since generic properties cover many real-world bugs, generic property verification tools are quite valuable. In 2022, over 800 buffer overflows were discovered in major software[1].

2.4. Verifying Dependability Properties

In this section, we illustrate how formal properties (belonging to a property class) are used to verify the dependability properties of software systems. Dependability properties are complex high-level properties that are rather difficult to be fully verified. However, each dependability property can be broken down into simpler properties [Sommerville 2011], e.g., reliability is supported by verifying safety as well as functional correctness. Such simpler properties are associated with informal properties, which in turn are verified by a set of formal properties. In Table 2.3, we list several examples of works from literature which verify dependability properties (according to Section 2.2) by using informal and formal properties.

Reliability and safety are two closely related system properties. While safety properties of a system are highly domain-specific (i.e., depending on the definition of a state), reliability properties

[1] Buffer overflow CVEs: `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow`

Formal properties		
Class	**Instance**	**Description**
Functional correctness	Function contracts	See Section 2.3.3
Functional correctness	(General) Invariant	See Section 2.3.3
Functional correctness	Ownership	See Section 2.3.3
Functional correctness	Frame condition	The set of elements that may or may not be modified during state transition [Przigoda et al. 2018].
Functional correctness	Coupling invariant	See Section 2.3.3
Generic	Null pointer exception	Occurs when variables are accessed that do not point to any object.
Generic	Index out of bounds	Invalid index when accessing elements of an array.
Generic	Buffer overflow	Accessing memory outside the reserved limits of a buffer.
Generic	Stack overflow	Stack size grows beyond the memory limits.

Table 2.2.: Examples of established and well-known formal properties for each property class.

are indirectly addressed by proving correctness of critical parts of the system. In Table 2.3 a few examples are shown which verify safety and reliability properties.

2.4.1. Verifying Security

Just as reliability and safety, security refers to a complex dependability property that can be barely fully verified. The term security is not even formally defined to an extent that a verification in the formal sense would be possible. Instead of verifying security as such, one has to focus on proving simpler sub-properties. More specifically, verifying security is often tied to defining an adversary model, which describes the capabilities of an adversary. Based on this model, a set of measures can be defined that prevent the adversary from doing harm. Crucially, this means that attacks not covered by the adversary model may have no measures to prevent them and are thus potentially still possible. One major concern when verifying security are side channel attacks in which the adversary derives information indirectly from observations about the system (e.g., the response time). To account for that aspect of security, additional verification methods proving bounds for resource consumption have to be implemented. A collection of approaches that verify security properties (or rather sub-properties) by considering various informal properties and formal properties, respectively, are shown in Table 2.3.

Reference	**Dependability property**	**Property class**	**Property**	**Formal properties**	**Tool (Language)**
[Hawblitzel et al. 2017]	Safety	Safety	n.a.	-	IronFleet (Dafny)
		Liveness	-	-	
		Relational	Simulation relation	Coupling invariant	
[Baumann et al. 2011]	Security	Uniform	Spatial separation	Function contract, Invariant, Ownership	VCC (C)
[Butterfield et al. 2014]	Security	Uniform	Spatial separation	Invariant	Isabelle (HOL)
		Relational	Simulation relation	Coupling invariant	
		Generic	Correctness	Function contract	
[Penix et al. 2005]	Security	Liveness	Temporal separation	-	SPIN (Promela)
[Heitmeyer et al. 2006]	Security	Uniform	Spatial separation	Invariant	PVS
		Relational	Simulation relation	Coupling invariant	
		Generic	Correctness	Function contract	
[Dam et al. 2013]	Security	Uniform	Spatial separation	n.a.	Hol4
		Relational	Bisimulation relation	Function contract	
		Relational	Non-interference	n.a.	
[Abbassi and Joua 2014]	Reliability	Safety	Transactional properties	Invariant	Event-B

Table 2.3.: Example references verifying dependability properties. Note that the properties of data separation, information flow, and fault isolation form spatial separation properties [Zhao et al. 2017].

3. Formal Methods for Software Verification

In this chapter, we present the current state of the art in formal methods for software verification and the tools available. First, we explain the most common formal methods used to verify programs. Then, we explain some foundations of verification tools, followed by how we conducted our research. Finally, we present a list of tools categorized according to these methods and their properties.

3.1. Overview Verification Methods

The following general descriptions of the four main verification methods are derived from a 2014 state of the art study on software verification [Beckert and Hähnle 2014].

3.1.1. Deductive Verification

Under Deductive Verification, we subsume all verification methods that (1) use an expressive (at least first-order) logic to state that a given target system is correct with respect to some specification and that (2) use logical reasoning (deduction) to prove the validity of such a statement. Perhaps the best-known approach along these lines is Hoare logic [Hoare 1969], but it represents only one of three possible architectures.

The first one being a general deductive framework in which the semantic of any programming or specification language can be modeled. Typical examples of such frameworks are Isabelle and Coq. Based on a formalisation of language semantics, nearly arbitrary properties of programs can be verified. The downside to this high expressiveness is the typically high amount of manual effort needed to formally define the semantics of a language, as well as to conduct proves. However, impressive case studies have been carried out with this approach, including formalisations of the floating-point logic of x86 processors, a nontrivial fragment of the Java language, the C language, and an OS kernel.

The second approach exploits program logics, where the semantic of a language is directly embedded into a logic and deductive rules. This approach has the advantage that language specific features can naturally be exploited when conducting proofs and the language semantics do not have to be defined by the user as they are inherently included in the logic. Examples of tools in this category include KeY and KIV.

The last category of deductive verification tools are those based on verification condition generators. These are based on rewriting rules that encode the language semantics. These rules are used to deduce a set of first-order assertions from a given program, which can then be passed on to some automated reasoning backend. Thus, this approach is the one with the highest degree of automation. However, it is hard for users to influence the proof search, check for intermediate results (such as partial and failed proof attempts) and relate the response of a reasoning backend back to the source code. Examples of this kind of approach are Why3 and Dafny.

All these approaches rely on extensive auxiliary specification (additional specification like loop invariants). Similar to lemmas in mathematical proofs, auxiliary specifications are needed to split proofs in manageable sub-proofs. Without them, full functional correctness of large and complex systems cannot be verified. While general verification frameworks have the highest expressiveness when specifying properties, they normally also require the most amount of manual work. This is a typical tradeoff in software verification.

3.1.2. Model Checking

For the Model Checking approach, a (software) system is modeled as a finite state automaton where each state is a possible variable assignment. Given a property (formalised, e.g., in propositional temporal logic), model checkers try to prove that each possible trace of the state automaton meets the property. The main issue with this approach is the exponentially growing number of possible states, which can get prohibitively large when modelling realistic systems (known as the state explosion problem). In order to address this challenge, several approaches have been proposed that make it possible to deal with real sized systems (even infinite automatons in some cases). One of these approaches is to limit the scope of considered executions by some given bound (bounded model checking). The advantage of model checking is that it can be very well automated. Typical examples of tools using this approach include CBMC and nuXmv.

Lately, there has been a trend in the literature to also subsume verification tools and methods that use reasoning technology, such as SMT and propositional satisfiability (SAT) solving under the term "model checking".

3.1.3. Refinement and Code Generation

Refinement is the approach in which a system is modeled at a very high level and then gradually refined in possibly many steps down to the desired final system (typically source code). For each of these refinement steps, it has to be proven that the original properties are maintained. Thus, the resulting final refinement still has the originally specified properties. This approach is often implemented using high-level specification languages based on set theory. The probably most prominent example of the approach is the Event-B language and its tools.

A closely related approach is to verify desired properties in a dedicated abstract programming language that is specifically designed to facilitate verification and to then generate code in some established programming language. This approach has the advantage that the verification effort is lifted to a suitable language/system in which specification and verification is usually comparatively easy while the ultimately resulting code is still in a standard programming language. Having code in a standard language allows a seamless integration into existing (legacy) systems. Typical examples of approaches like this include Dafny and F*.

3.1.4. Abstract Interpretation

Abstract Interpretation is based on the idea to introduce abstract domains for the variables of a system (the domain of a variable contains its possible values). These abstract domains are chosen to be sound and finite. The approach guarantees that a property proven using the abstract domains holds for the original domains as well, while the abstract domain being finite enables easier/faster analysis of the problem as it becomes decidable. This can be viewed as a general approach to reason in finite domains about infinite state systems. This technique allows for the very fast analysis of comparatively large systems. However, completeness is reduced, i.e., not all valid properties that can be proven for the concrete domain can be proven using an abstract domain as well. Several tools use this approach for program analysis, including Astrée and Frama-C.

3.2. Foundations of Tooling

In the previous section, we have presented the main methods of formal software verification. For each of these methods, there is a wide range of tools available implementing the methods for all varieties of programming languages and properties. However, there are some common building blocks on which the majority of the most successful tools are based. In this section, we discuss these building blocks.

3.2.1. SMT and SAT Solvers

Beckert et al. call efficient automated reasoning a "ubiquitous sub-task of hard- and software verification" [Beckert and Hähnle 2014]. In the majority of currently well established tools, this sub-task is carried out by SAT or SMT solvers. SMT solvers do not only provide the easiest way to automate reasoning, but also most of them support a standardized input language (SMTLIB2 [Barrett et al. 2016]), which allows SMT solvers to be used interchangeably. As such, they are used as a backend to verification tools. The typical workflow is that a verification tool processes the input (specification and program) and internally calls an SMT solver to discharge the resulting proof obligations (which are formulas not containing program elements any more). Due to the widespread use of these solvers for program verification, discussing them here is inevitable. Some of the most well established SMT solvers are Z3 [Moura and Bjørner 2008], CVC5 [Barbosa et al. 2022], Yices [Dutertre 2014], MathSAT [Cimatti et al. 2013], and Alt-Ergo [Conchon et al. 2018]. SMT solvers as a stand-alone tool are not verification tools, but most fully automated verification tools would be inconceivable without them. As such, we consider them to be one of the most important building blocks of formal software verification tools. Note also that any improvements to these building blocks (e.g., improved SAT solvers based on quantum computers (see Section 4.3) are direct improvements to any tool that builds on them.

Similarly to fully automatic reasoning as a backend, interactive theorem provers provide the capabilities to reason manually, often times over even more complex properties. Well known examples of such systems are Coq, Isabelle and Lean among others. However, all of those tools provide a more direct way to reason about programs than SMT or SAT solvers do, which is why we listed them explicitly in the section on tools (Section 3.4) and do not discuss them here in more detail.

3.2.2. Intermediate Languages and Verification Frameworks

Another commonly used building block for verification tools is the concept of an intermediate language or platform. This concept is familiar from programming languages that rely on an intermediate representation instead of direct compilation (e.g., LLVM IR [Lattner and Adve 2004] and Java bytecode). For verification purposes, this means that instead of translating a program together with its specification directly into some logic representation, the problem statement is first translated into an intermediate language. This has two main advantages: (1) translation into an intermediate language is often much easier because it is closer to a programming language, (2) formulating a verification problem in an intermediate language allows any solver capable of dealing with that intermediate language to be applied. Examples of well-known intermediate languages are Boogie [Barnett et al. 2006], Viper [Müller et al. 2016], and WhyML [Filliâtre and Paskevich 2013].

Related to this is the idea of frameworks that facilitate the construction of verification tools. There are several examples of such frameworks, using different approaches to provide such a general framework. Well-known examples include the frameworks associated with the intermediate languages mentioned above, as well as others such as KLEE, which is a symbolic execution engine for LLVM on which several tools are built [Cadar et al. 2008]. And the K-Framework, which is a framework for formalizing the semantics of programming languages, can be used to construct verification tools based on that formalisation [Roșu and Serbănută 2010].

3.3. Methodology of Literature Review

The state of the art in software verification, described in the next section, is based on a literature review. To reduce the risk of missing important publications of the wide area of formal methods, we applied different investigation approaches.

Scope Before discussing the process of our literature research, we define the scope of this study – describing the field of formal methods for software, particularly methods for verification for software systems. We explicitly do *not* consider the following aspects or features of systems:

- Hybrid properties: properties with a physical component or a logic which involves continuous behavior.
- Machine-learning-based systems with components that are learned through some statistical approach (e.g., neural networks or HMMs). Verifying ML-based systems is a very active field of research in the last few years, but will not be considered for this study.
- Probabilistic properties: components with probabilistic (or non-deterministic) behavior.
- Anything below operating system level, this includes instruction set architectures and hardware.
- Bug-finding methods: we limit the scope of this work to verification methods and exclude methods like bug finding or test generation. As a rule of thumb, we will consider everything that is able to provide some kind of formal guarantee rather than to just provide warnings or to point out mistakes.

Note that we explicitly state these restrictions for properties or components of systems, rather for entire systems themselves. This allows us to consider systems partially in accordance with the defined topic. As an example, consider the control software for a drone system. We would consider the software itself with all the possible properties, be it functional correctness, safety, or security, but we would not consider how to model the hybrid dynamics of a flying object.

Systematic literature search We conducted a systematic literature search with several branches of investigation in order to build a broad knowledge base. As a starting point, we considered a survey by Beckert et al. in 2014 [Beckert and Hähnle 2014], which describes the state of the art at this point in time. This paper served as the basis for **a snowballing process**, where we collected all the papers citing that survey, resulting in a list of 62 papers. Of these papers, **we only included** those that met the following criteria (1) **peer-reviewed**, (2) **published at a conference ranked B or higher**[1], (3) **matches the defined range of topics**, and (4) **written in English or German**. The result of the snowballing and filtering process was a list of 11 papers (see Appendix A).

As a second source of scientific publications we considered **all papers published 2014 or later at one of the following conferences: FASE, TACAS, FM, CAV, IJCAR and NFM**. These conferences are to the best of our knowledge and after consideration with experts, the most influential ones in the field. We automated the process of finding all publications with the mentioned conditions and thus were able to gather a table of 2223 papers. For each paper we extracted and saved the following information: title, authors, #citations, #accesses, abstract, year of publication, conference, and link. The information for number of citations as well as number of accesses were taken from the website of Springer[2] on October 17, 2022. Out of this large body of papers **we only consider** the ones meeting the following criteria (1) **more than 50 total citations**, or (2) **more than 8 citations per year** (on average), or (3) **more than 10.000 accesses**, and (4) **matches the defined range of topics**.

This resulted in a set of 39 papers that we considered (see Appendix B). The thresholds chosen for citations and accesses were chosen in such a way that the number of papers considered for this state-of-the-art analysis was considered manageable. The 39 papers selected represented more than 3% of the recorded accesses and more than 13% of the recorded citations, while representing less than 2% of the papers considered. This further confirms our intuition that the papers selected in this way are highly relevant contributions.

As an additional source of insight, we conducted **a brief unstructured web search** on the aforementioned topics. The goal was mainly to identify relevant news about the use or spread of formal methods, especially in industry, rather than to collect more scientific publications. In this way, we tried to cover the social aspect of how formal methods are currently discussed in the media.

Last but not least, we **examined the most well established competitions** in the field of formal verification, which we consider to be SV-COMP [Beyer 2021] and VerifyThis [Huisman et al. 2020; Dross et al. 2021]. They cover different notions of formal verification. While SV-COMP is explicitly “an annual comparative evaluation of fully automatic software verifiers for C and Java programs” [Beyer 2021], VerifyThis tends to “emphasizes verification problems that go beyond what can be proved fully automatically and require instead human experts” [Dross et al. 2021]. This leads to a very different set of tools participating in these competitions.

We **limited the tools considered** to those which (1) have **a sizable community**, (2) have been **present for several years** as evidenced by both publications and releases, and (3) are **not close variants of other tools** included. Especially the last point is sometimes hard to establish, as it is normal for tools to be developed in a particular environment and to spawn slightly modified versions. We try to cover only the most relevant tool of each family, rather than including several closely related tools. In total, we collected 50 papers and 31 tools that we considered for our state-of-the-art analysis.

3.4. State of the Art in Software Verification Tools

The presentation of software verification tools begins with an overview of the tools, their supported programming languages, and their verifiable properties. This is followed by a detailed evaluation of each tool with respect to industrial use. Finally, the supported languages and general limitations are discussed, followed by a conclusion.

[1] Based on scores from the CORE Conference Ranking `https://www.core.edu.au/conference-portal`

[2] Springer Publishing: `https://link.springer.com/`

3.4.1. Overview of Tools

The following overview of software verification tools, selected according to the procedure in Section 3.3, is based on a literature review and information provided by the development community of each tool via web sites. There has been no evaluation of the tools themselves.

Table 3.1 shows a list of all tools along with the verification method, supported programming languages, and verifiable properties. Some tools found in the literature search were excluded because they are no longer maintained. The language column indicates the primary programming languages for which a tool is designed or used. This does not reflect the fact that some tools are capable of generating code for a variety of target languages. Nor does it reflect the fact that tools based on, e.g., higher-order logic can be used to define semantics for any programming language.

Tool	Formal method	Language	Properties
ACL2	deductive verification (interactive)	Lisp	functional correctness, bug finding
Agda	deductive verification	Agda	functional correctness
Alloy Analyzer	refinement, deductive verification	Alloy	functional correctness, safety
AproVE	symbolic execution	Java, C, LLVM, Haskell, Prolog, TRS	termination
Astrée	static analysis	C, C++	safety
CBMC	deductive verification	C, Java	bug finding, lightweight properties
Coq	proof assistant	Coq	functional correctness, safety, security, refinement
CPAChecker	model checking	C	safety, functional correctness
Dafny	deductive verification	Dafny	functional correctness, bug finding
Event-B	deductive verification, refinement	Event-B	refinement
F*	deductive verification	F*	functional correctness
Frama C	deductive verification (batch)	C	functional correctness, bug finding
HOL4	proof assistant	HOL	functional correctness
Isabelle	proof assistant	HOL	functional correctness, safety, security, refinement
KeY System	deductive verification	Java	functional correctness, bug finding (test case generation)
KIV	deductive verification (interactive)	Abstract System Description	functional correctness, bug finding, security, refinement
Lean	deductive verification, proof assistant	LEAN	functional correctness
mCRL2	model checking	labeled transition systems	functional correctness

Table continued on the next page

Table 3.1.: Overview of all tools described in detail

Tool	Formal method	Language	Properties
Microsoft Windows Static Driver Verifier	symbolic execution	C, C++	safety, reachability
Nagini	verification condition generation, symbolic execution	Python	functional correctness
NuXMV	model checking	NuSMV language	safety
OpenJML	deductive verification, runtime verification	Java	functional correctness, safety
PVS	proof assistant	PVS	functional correctness, safety, security, refinement
SeaHorn	model checking, abstract interpretation	LLVM, C	termination, reachability
SMACK	model checking	LLVM	functional correctness
Software Analysis Workbench SAW	symbolic execution	C, Java, Cryptol	functional correctness
Spark ADA	symbolic execution, static analysis	Ada	functional correctness
SPIN	model checking	Promela, extraction from C possible	safety
TLA+	model checking	State machines	functional correctness
UPPAAL	model checking	custom timed automata	safety, liveness properties of temporal automata
VeriFast	deductive verification (batch)	Java, C	functional correctness

Table 3.1.: Overview of all tools described in detail (cont.)

A reduced list of tools that support the most relevant languages in practice, C, C++, Java, and Rust, is shown in Table 3.2. The table shows the languages and properties covered by the tools. See Section 3.4.3.2 for a discussion of important languages for basic IT systems.

3.4.2. Maturity of Tools

This section discusses software verification tools and their maturity with respect to industrial use. A mature tool should have a sufficiently large developer community so that continued availability and maintenance can be expected over several years. Furthermore, the effort to use the tool should be low in terms of technical requirements, required expertise, and execution of the actual verification. Last but not least, the tool should be able to verify relevant properties without serious limitations.

The three dimensions – community, effort, and capability – are further broken down into subdimensions such as availability, expertise, and automation for effort, and scalability and limitations

Formal method	**Tool**	**C**	**C++**	**Java**	**Rust**
Deductive Verification	CBMC	fc	fc	fc	
	Frama C/Why	fc			
	KeY			fc	
	OpenJML			fc	
	Isabelle	fc		fc	
	VeriFast	fc		fc	
Model Checking	AproVE	te		te	
	CPAChecker	sa, fc			
	SAW	fc		fc	
	SDV	sa	sa		
	SMACK	fc	fc		fc
Abstract Interpretation	Astrée	sa	sa		
	SeaHorn	fc	fc		fc

Table 3.2.: Overview matrix of tools and their supported programming language, as well as verifiable properties *(fc = functional correctness, sa = safety, te = termination)*.

for capability. Each dimension or sub-dimension consists of either multiple classes or parameters. Table 3.3 provides an overview of the maturity assessment dimensions, along with the classes and parameters.

The **community dimension** includes three classes – large, medium, and small. A **large community** consists of many stakeholders (companies, institutions, universities, etc.), each with several people frequently contributing both new functionality and maintenance of the tool suite. There are also regular publications about the tool by different authors. A **medium-size community** consists of only a few stakeholders, but they contribute regularly. Publications are mainly limited to authors who directly contribute to the content development of the tool. A **small community** is mostly supported by a single stakeholder, and there is sporadic development of the tool.

The **effort dimension** is divided into three sub-dimensions: availability, expertise, and automation.

Availability describes the effort required to get a tool up and running and to keep it running. In particular, precompiled binaries for the major operating systems (Windows, Linux, MacOS) and an up-to-date software stack are important. Typical examples of an outdated software stack are a limitation to Ubuntu 16.04 or Python 2. In general, it is preferable for a tool to be available as a standalone program. Integration as a plugin in frameworks is often advantageous. However, this can quickly create a dependency on a particular framework that users may not want to use in their software development process (in addition, if the tool is updated, the plug-in must be updated as well). The usefulness of a GUI depends heavily on the type of tool. An automated framework may only be run by scripts anyway, while a tool with a lot of manual work will benefit greatly. Whether a tool is available as open source or not is generally irrelevant for the evaluation of industrial use. However, with respect to the underlying goal of this study to build a community that advances the state of the art, open source is preferable.

The **expertise** sub-dimension includes which programming and specification languages a tool supports. If the software to be verified has to be written in a **tool-specific language**, this implies a considerable amount of effort to learn and use the language. There are tool-specific languages, such as Dafny (see Section 3.4.2.9), that can be translated into C after verification. The code can

Dimension	Classes and parameters	Effect
Community (class)	large – frequent releases/commits and publications	++
	medium – regular releases/commits and publications	+
	small – sporadic releases/commits and publications	–
Effort: Availability (parameter)	pre-compiled binaries available	+
	support for major OS	+
	up-to-date software stack	++
	standalone executable	+
	graphical user interface	○
	open source	○
Effort: Expertise (parameter)	no custom programming language required	+
	common specification language used	+
	textbook available	+
	comprehensive online manual or support available	++
Effort: Automation (class)	fully automated (push button)	++
	autoactive	+
	interactive	–
Capability: Scalability (class)	industrial use cases	++
	multiple academic examples	+
	small academic examples	○

Table 3.3.: Dimensions and sub-dimensions (classes or parameters) and their impact on the maturity assessment *(++ strongly preferable, + preferable, ○ no preference in general or tool dependent, – undesirable)*.

then be further processed in the usual tool chains used by developers. However, this only covers some parts of the development process. The use of **common de facto standards for specification languages** such as ACSL [Baudin et al. 2008] or JML [Chalin et al. 2005] is therefore advantageous. We use the term de facto standard to describe languages that are well known and used by a wide range of stakeholders, although they have never been approved by a standards body. To familiarise yourself with a tool, it is helpful to have a **textbook** that describes in detail how the tool works. **Comprehensive online documentation** with questions and answers and examples is also helpful. Especially if they include support from a community or company.

The **automation** dimension comprises three classes that describe the kind of interaction needed for verification. **Fully automated** tools such as CBMC (Table 3.9) require the least effort. **Autoactive** verification tools such as Dafny (see Table 3.12) require some interaction in the form of extensive specification (sometimes iteratively refined) but run without further manual actions afterward. **Interactive** tools, such as Isabelle (see Section 3.4.2.14), rely on user input for extensive specification as well as during the proof itself. Note, however, that these tools often have powerful automation capabilities that allow many proofs to be made with very little manual interaction. Note also that, as we discussed earlier in the context of software verification, automation most often comes at the cost of a loss of expressiveness. Tools that require the most manual effort also provide the most powerful logic, and thus the ability to prove complex properties (see Section 3.5.4 for a detailed explanation and illustration).

The **scalability** dimension describes how many and which examples and use cases are known for a tool. Other aspects of scalability are difficult to evaluate, the size or complexity of the verifiable software depends very much on the property to be verified and cannot be described in a meaningful general way. The computational complexity is similar for many tools, since SMT and SAT solvers are almost always used. For the same reasons, there is no compact representation of the limitations of a tool. Limitations can be seen as the size or complexity of the software to be verified. However, limitations can also be related to the functional scope of a tool, i.e., the number of features supported by the tool. It is difficult to judge whether a tool that can verify a single property very well should be considered limited or not. Furthermore, the supported features of the target programming language can be considered a limitation of a tool. The various aspects of limitations are discussed in Section 3.5.

The column “effect” describes whether a class or parameter is preferred or not. This reflects the choice of the authors. As explained for some classes and parameters, there are sometimes good reasons for different opinions. Therefore, **we do not derive a single maturity score**.

On the following pages, each of the tools is presented with a brief description and a table that assesses the maturity of the tool.

3.4.2.1. Tool - ACL2

ACL2 is a logic and programming language with corresponding tool to model and verify computer systems. It is developed at the University of Texas and is openly available (`https://github.com/acl2/acl2`). Several books, papers as well as tutorials and binaries are available at the website. ACL2 was used to verify properties of systems by multiple major companies including IBM, Motorola and AMD. The prover for ACL2 is an interactive theorem prover thus requiring comparatively a lot of user input to conduct proofs.

Info	description	A theorem prover for high-level descriptions of systems
	method	deductive verification (interactive)
	website	https://www.cs.utexas.edu/users/moore/acl2/
	verifiable properties	functional correctness, Bug Finding
	current version	Version 8.5 (July 2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI, Plug-in (Emacs, Eclipse)
	open source	✓
Expertise	no custom programming language	✓(Lisp)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	industrial use cases

Table 3.4.: Description and maturity evaluation of tool: ACL2

3.4.2.2. Tool - Agda

Agda is a dependently typed programming language which as such can be used as a proof assistant to prove mathematical theorems. It is developed at Chalmers University (Gothenburg, Sweden) and is available as open source at GitHub (`https://github.com/agda/agda`). Agda is mainly intended for Linux and OSX systems. A binary version bundled with Emacs is however downloadable for Windows. Books as well as tutorials and extensive documentation can be found at the website.

Info	description	a programming language which also allows writing proofs
	method	deductive verification
	website	`https://wiki.portal.chalmers.se/agda/`
	verifiable properties	functional correctness
	current version	2.6.2.2 (April 2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✗
	user interface	Plug-in (Emacs)
	open source	✓
Expertise	no custom programming language	✗(Agda)
	common specification language	✗
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	multiple academic examples

Table 3.5.: Description and maturity evaluation of tool: Agda

3.4.2.3. Tool - Alloy Analyzer

Alloy is the name of an open source language as well as that of the corresponding analyzer. Closely related is the model finding tool Kodkod. There are also several other tools and plugins build around or on top of Alloy and Kodkod. Alloy has been used in various scenarios including finding security holes and designing telephone switching networks. It is mainly developed at the Software Design Group at MIT. The source code is hosted publicly at GitHub (`https://github.com/AlloyTools`). Additional resources like documentation, list of papers and binaries are available at the website of the project. With more than 1000 papers written about and using Alloy it has a sizable community.

Info	description	Fully automatic tool based on the alloy language
	method	refinement, deductive verification
	website	https://alloytools.org/
	verifiable properties	functional correctness, safety
	current version	Release 6 (November 2021)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✓
Expertise	no custom programming language	✗(Alloy)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.6.: Description and maturity evaluation of tool: Alloy Analyzer

3.4.2.4. Tool - AproVE

AproVE is a tool for reasoning about termination and complexity of programs. Based on term rewrite systems it also supports several other languages like Java byte-code, C, Haskell and Prolog. AproVE is not open source. As a Java command line program, it is compatible with all major operating system and additionally offers an Eclipse Plugin which allows using it through a GUI. Binaries as well as publications are available online, documentation as well as a textbook are however missing.

Info	description	termination proofs for several languages
	method	symbolic execution
	website	https://aprove.informatik.rwth-aachen.de/
	verifiable properties	termination
	current version	?? (December 2021)
Community	community	small
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-in (Eclipse)
	open source	✗
Expertise	no custom programming language	✓(Java, C, LLVM, Haskell, Prolog, TRS)
	common specification language	✓
	textbook available	✗
	comprehensive online manual	✗
Automation	automation	fully automated
Scalability	scalability	multiple academic examples

Table 3.7.: Description and maturity evaluation of tool: AproVE

3.4.2.5. Tool - Astrée

Astrée is a static code analyzer for C or C++ code that can prove the absence of runtime errors and invalid concurrent behavior. It targets primarily embedded applications but is not restricted to that. As an exception to a lot of other tools Astrée offers full support for floating-point computations. It is a standalone tool with a graphical user interface. Astrée is used by several well known companies like Airbus, BMW, and Bosch. Astrée is sound but may raise false alarms due to it using an over approximation in certain cases. It is commercially distributed by AbsInt with the offer of additional training and support when purchasing a license.

Info	description	Fully automatic static analysis tool for safety properties in C
	method	static analysis
	website	https://www.absint.com/astree/index.htm
	verifiable properties	safety
	current version	Release 22.10 (October 2022)
Community	community	large
	commercial support	✓
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✗
Expertise	no custom programming language	✓(C, C++)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.8.: Description and maturity evaluation of tool: Astrée

3.4.2.6. Tool - CBMC

CBMC (the C Bounded Model Checker) is a tool for verifying C and C++ programs. As a bounded model checker it is not able to provide full proofs in general but rather up to a certain bound (e.g. for loop iterations and recursion depth). CBMC is able to verify memory safety, absence of exceptions and checks for user-specified assertions. Binaries and documentation as well as examples are available on the website. CBMC is open source and available at GitHub (`https://github.com/diffblue/cbmc`). The main paper about CBMC [Clarke et al. 2004] is one of the most cited papers found during our literature research. A variant for Java is available under the name JBMC (see website).

Info	description	A bounded model checker for C
	method	deductive verification
	website	https://www.cprover.org/cbmc/
	verifiable properties	bug finding, lightweight properties
	current version	Version 5.67 (September 2022)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-in (Eclipse)
	open source	✓
Expertise	no custom programming language	✓(C, Java)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.9.: Description and maturity evaluation of tool: CBMC

3.4.2.7. Tool - Coq

Coq is a platform that provides a formal language for writing mathematical proofs, as well as the means to machine check those proofs. Writing proofs in Coq involves a considerable amount of manual work, but the expressiveness of the Coq language is very high, allowing proofs of a very wide range of properties. Coq has been used to verify the C compiler CompCert [Leroy et al. 2016]. Development of Coq is open on GitHub (`https://github.com/coq/coq`), and with over 200 contributors, Coq has one of the most established communities in the list of tools considered. Coq is available for all major platforms. Several books about Coq as well as documentation can be found on the website.

Info	description	theorem prover with the ability to be applied to programming languages
	method	proof assistant
	website	https://coq.inria.fr/
	verifiable properties	functional correctness, safety, security, refinement
	current version	Release 8.16.0 (September 2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI, Plug-in (VSCode, Emacs, Vim)
	open source	✓
Expertise	no custom programming language	✗(Coq)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	industrial use cases

Table 3.10.: Description and maturity evaluation of tool: Coq

3.4.2.8. Tool - CPAChecker

CPAchecker is a fully automatic configurable verification tool for C. It is openly available at the website. CPAchecker is running on all major operating systems. It achieved several medals over the last years in the SV-Competition series. Rudimentary documentation as well as the main papers about the tool are available at the website. In addition to the program itself and the specification, a configuration is required to run the tool, which describes the kind of analysis that is to run on the given program.

Info	description	configurable fully automatic checker for
	method	model checking
	website	https://cpachecker.sosy-lab.org/
	verifiable properties	safety, functional correctness
	current version	2.2 (Nov 2022)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✓(C)
	common specification language	✗
	textbook available	✗
	comprehensive online manual	✗
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.11.: Description and maturity evaluation of tool: CPAChecker

3.4.2.9. Tool - Dafny

Dafny is a language (and the corresponding verification tool) which was developed with formal verification in mind. It allows writing specifications in the code as a standard feature of the language and incorporates typical features for verification like quantifiers or lemmas. Any Dafny program can always be transpiled to several standard languages including C#, Java and JavaScript. Verification in Dafny is done fully automatically using the SMT solver Z3 with Boogie as an intermediate language. Dafny is distributed under the MIT license and openly available at GitHub (`https://github.com/dafny-lang/dafny`).

Info	description	Language and automatic verifier for custom language Dafny (can be refined to C)
	method	deductive verification
	website	https://dafny.org/
	verifiable properties	functional correctness, bug finding
	current version	3.9.1 (10.2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-in (VSCode, Emacs)
	open source	✓
Expertise	no custom programming language	✗(Dafny)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	autoactive
Scalability	scalability	small academic examples

Table 3.12.: Description and maturity evaluation of tool: Dafny

3.4.2.10. Tool - Event-B

Event-B is a formal method based on set theory and refinement to model and analyze at system-level. The basic idea is to provide a very coarse grained abstraction of the desired system and prove properties of it and then refining it down to a suitable level of abstraction (possibly even down to a full implementation). An Eclipse-based IDE for Event-B is available under the name Rodin. Event-B has been used by several major companies including Bosch, Siemens and SAP. Documentation as well as binaries are available via the website. The source code is openly available on SourceForge (`https://sourceforge.net/p/rodin-b-sharp/svn/HEAD/tree/`).

Info	description	Set theory based refinement tool
	method	deductive verification, refinement
	website	http://www.event-b.org/index.html
	verifiable properties	refinement
	current version	3.7 (25.5.2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI, Plug-in (Eclipse)
	open source	✓
Expertise	no custom programming language	✗(Event-B)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.13.: Description and maturity evaluation of tool: Event-B

3.4.2.11. Tool - F*

F* is a functional programming language designed with the intent to ease verification. Programs written in F* can be transpiled to other well established language including OCaml, F# and C. A main goal of the F* language is to build a fully verified HTTPS stack. F* is openly developed on GitHub (`https://github.com/FStarLang/FStar`). With over 100 contributors, F* is one of the tools with the largest active developing communities. Precompiled binaries are available for Windows and Linux on the website. Additionally, the website offers a list of tutorials as well as publications and documentation.

Info	description	a language designed for formal verification with corresponding tools which can be compiled to several standard languages
	method	deductive verification
	website	https://www.fstar-lang.org/
	verifiable properties	functional correctness
	current version	2022.01.15 (Jan 2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✗ (Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-in (Emacs, VS-Code)
	open source	✓
Expertise	no custom programming language	✗(F*)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	autoactive
Scalability	scalability	industrial use cases

Table 3.14.: Description and maturity evaluation of tool: F*

3.4.2.12. Tool - Frama C

Frama-C is rather a framework of analysis plugins than a single tool. All of those plugins are concerned with the formal correctness of C programs but use different techniques and approaches which range from deductive verification over runtime verification to test-case generation. Frama-C is backed by a substantial community which has been built over several years of active development and use. An extensive list of publications as well as manuals and tutorials can be found on the website. Frama-C is open source but no public repository is available. Binaries for Linux and Mac can be downloaded from the website however Windows is currently only supported via the Windows Subsystem for Linux (WSL). A commercial version of Frama-C, called TrustInSoft Analyzer, is available.

Info	description	Framework for the analysis of C programs
	method	deductive verification (batch)
	website	https://frama-c.com/
	verifiable properties	functional correctness, bug finding
	current version	26 (2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	OSX, Win (WSL only), Linux
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✓(C)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.15.: Description and maturity evaluation of tool: Frama C

3.4.2.13. Tool - HOL4

HOL4 is an interactive theorem prover for higher order logic that is mainly developed at Cambrige University (UK) and the Chalmers University (Sweden). It is openly available at Github (`https://github.com/HOL-Theorem-Prover/HOL`). Precompiled binaries are not available so the user has to build from sources and some configuration is needed. However good tutorial material together with an extensive documentation and examples is available at the tool website. Windows is only supported via CygWin or Linux subsystem for Windows. “HOL is particularly suitable as a platform for implementing combinations of deduction, execution and property checking” (HOL4 website).

Info	description	an interactive theorem prover for higher order logic
	method	proof assistant (interactive)
	website	https://hol-theorem-prover.org/
	verifiable properties	functional correctness
	current version	Kananaskis-14 (February 2021)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✗
	supported OS	✗ (Linux, OSX)
	up-to-date	✓
	standalone	✓
	user interface	CLI/Plugin(Emacs)
	open source	✓
Expertise	no custom programming language	✗(ML)
	common specification language	✗(ML)
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	multiple academic examples

Table 3.16.: Description and maturity evaluation of tool: Hol4

3.4.2.14. Tool - Isabelle

Isabelle is a proof assistant which is nowadays most famous in its version as Isabelle/HOL which is a theorem prover for higher-order logic. Although proving in Isabelle naively requires a lot of manual work (writing out the proof) several internal tools allow for a moderate automation of proofs. Additionally, Isabelle allows turning executable specifications into code in OCaml, Haskell, and Scala. Furthermore, Isabelle comes with a large archive of formal proofs which allows to directly apply already proven theorems to ones own proofs. Isabelle is available for all major operating systems as well as a Docker image. Documentation can be found at the website. Source code is openly available at `https://isabelle-dev.sketis.net/source/isabelle/`.

Info	description	Proof assistant for mathematical formulas
	method	proof assistant
	website	https://isabelle.in.tum.de/
	verifiable properties	functional correctness, safety, security, refinement
	current version	Isabelle2022 (10.2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✓
Expertise	no custom programming language	✗(HOL)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	industrial use cases

Table 3.17.: Description and maturity evaluation of tool: Isabelle

3.4.2.15. Tool - KeY System

KeY [Ahrendt et al. 2016] is an interactive deductive verification tool for Java using JML for specifications. It has a medium-sized community primarily backed by TU Darmstadt and the Karlsruhe Institute of Technology. The KeY tool has a long history and is available as open source along with full development process insights[3]. An extensive overview of publications can be found at the project website[4]. KeY is a platform-independent standalone tool with binaries available at the website[4]. It provides a GUI as well as Eclipse integration. KeY supports verification of Java (particularly Java Card) based on JML specifications. Some verification properties require a JML dialect. The KeY community provides a standard textbook [Ahrendt et al. 2016] and online documentation[4]. Applications of KeY comprise several scientific examples, e.g., sorting algorithms of the Java Standard Library [Beckert et al. 2017] or parts of an electronic voting software (`http://urn.nb.no/URN:NBN:no-82770`). There are no industry-oriented applications recorded. For Automation KeY supports some sophisticated functionalities. Limitations in terms of programming language features are limited floating-point arithmetic for Java Card and no support for, e.g., generics or lambdas. Table 3.18 shows a summary of the maturity evaluation of KeY.

Info	description	Interactive verification tool for Java with JML specifications
	method	deductive verification
	website	https://www.key-project.org/
	verifiable properties	functional correctness, bug finding (test case generation)
	current version	2.10 (12.2021)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✓
Expertise	no custom programming language	✓(Java)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	multiple academic examples

Table 3.18.: Description and maturity evaluation of tool: Key System

[3] KeY git repository: `https://git.key-project.org/key-public/key`

[4] KeY project website: `https://www.key-project.org/`

3.4.2.16. Tool - KIV

The KIV system is a tool for development and interactive verification of software developed at the University of Augsburg. KIV is mainly intended to use as a plugin for Eclipse and requires the installation of Eclipse. As a reasoning tool, KIV relies on Kodkod. KIV was used in several industrial pilot studies which can be found at the website. Additionally, KIV teams regularly participated in VerifyThis competitions. KIV is free for non-commercial use but not open source.

Info	description	interactive verification of abstract models with refinement capabilities
	method	deductive verification (interactive)
	website	`https://kiv.isse.de/`
	verifiable properties	functional correctness, bug finding, security, refinement
	current version	8.1 (10/14/2022)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✗
	user interface	Plug-in (Eclipse)
	open source	✗
Expertise	no custom programming language	✓(Abstract System Description)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	industrial use cases

Table 3.19.: Description and maturity evaluation of tool: KIV

3.4.2.17. Tool - Lean

LEAN is a functional programming language which is aimed at easily writing correct code. Additionally, LEAN can be used as an interactive theorem prover. It is developed by Microsoft Research and is openly available at GitHub (`https://github.com/leanprover/lean4/`). LEAN can be set up on all major operating systems and supports integration into several common editors. Extensive documentation as well as a list of papers about LEAN and papers using LEAN can be found on the website. LEAN comes with a library of mathematical proofs called mathlib.

Info	description	extensible interactive Theorem prover
	method	deductive verification
	website	https://leanprover.github.io/
	verifiable properties	functional correctness
	current version	4.0.0 (Aug 2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	-
	open source	✓
Expertise	no custom programming language	✗(LEAN)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	multiple academic examples

Table 3.20.: Description and maturity evaluation of tool: Lean

3.4.2.18. Tool - mCRL2

mCRL2 is a language with a corresponding tool for verification of concurrent systems and protocols. It is available for all major operating systems. Source code as well as online documentation and a list of publications can be found at the website. Several major academic use cases are also available online.

Info	description	language and toolset for verification of concurrent systems and protocols
	method	model checking
	website	`https://www.mcrl2.org/`
	verifiable properties	functional correctness
	current version	202206.1 (June 2022)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✓(labeled transition systems)
	common specification language	✗
	textbook available	✗
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.21.: Description and maturity evaluation of tool: mCRL2

3.4.2.19. Tool - Microsoft Windows Static Driver Verifier

The Microsoft Static Driver Verifier is a static analysis tool based on symbolic execution and a fixed set of properties which is able to verify that this set of properties is respected by a driver. Microsoft requires official drivers to be checked by this tool. As it is specialized on Windows drivers it is not available for other platforms. It is currently not open sourced, but documentation is available at the website. Additionally, an integration into Visual Studio is provided.

Info	description	Verification of Windows drivers
	method	symbolic execution
	website	https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier
	verifiable properties	safety, reachability
	current version	4.0.2204.1 (Aug 2022)
Community	community	large
	commercial support	✓
Availability	precompiled binaries	✓
	supported OS	✗ (Win)
	up-to-date	✓
	standalone	✓
	user interface	VS Integration, MS Build
	open source	✓
Expertise	no custom programming language	✓(C, C++)
	common specification language	✓
	textbook available	✗
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.22.: Description and maturity evaluation of tool: Static Driver Verifier

3.4.2.20. Tool - Nagini

Nagini is a fully automatic verification tool for statically typed Python based on the Viper framework. It is developed as an open source project on GitHub (`https://github.com/marcoeilers/nagini`) by the ETH Zurich. As a python tool it is compatible with all major operating systems. Rudimentary documentation as well as the main paper describing Nagini can be found on the website.

Info	description	functional correctness of typed Python based on Viper framework
	method	verification condition generation, symbolic execution
	website	https://github.com/marcoeilers/nagini
	verifiable properties	functional correctness
	current version	0.9 (May 2021)
Community	community	small
	commercial support	✗
Availability	precompiled binaries	✗
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-In (PyCharm)
	open source	✓
Expertise	no custom programming language	✓(Python)
	common specification language	✓
	textbook available	✗
	comprehensive online manual	✓
Automation	automation	autoactive
Scalability	scalability	small academic examples

Table 3.23.: Description and maturity evaluation of tool: Nagini

3.4.2.21. Tool - nuXmv

nuXmv is a model checker which is the successor of NuSMV. It relies on SAT/SMT-solvers as backends for reasoning about finite- as well as infinite-state systems. The main solver used in nuXmv is MathSAT5. It is free for non-commercial or academic purposes but is not open source. Binaries as well as documentation can be found at the website.

Info	description	symbolic model checker for the analysis of synchronous systems
	method	model checking
	website	https://nuxmv.fbk.eu/
	verifiable properties	safety
	current version	2.0 (14.10.2019)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✗
Expertise	no custom programming language	✗(NuSMV language)
	common specification language	✗
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.24.: Description and maturity evaluation of tool: NuXMV

3.4.2.22. Tool - OpenJML

OpenJML is a fully automatic verification tool for Java programs annotated with JML specifications. It is the successor of ESC/Java2. It was recently updated to support Java 17. Based on a deductive verification approach, JML uses SMT solvers as backend to discharge verification conditions. An eclipse integration is provided, but has not yet been adapted for the latest Java versions. A mode to automatically create runtime assertions from JML contracts is also available. The source code is openly available at GitHub. With four contributors listed on the GitHub repository, OpenJML is one of the tools with a rather small community. It is however actively developed and maintained. As a Java application, OpenJML runs on any major operating system.

Info	description	Fully automatic prover for Java specified by JML
	method	deductive verification, runtime verification
	website	https://www.openjml.org/
	verifiable properties	functional correctness, safety
	current version	0.17.alpha 6.6.2022
Community	community	small
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✗(OSX, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-in (Eclipse)
	open source	✓
Expertise	no custom programming language	✓(Java)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	autoactive
Scalability	scalability	small academic examples

Table 3.25.: Description and maturity evaluation of tool: OpenJML

3.4.2.23. Tool - PVS

PVS is a platform which contains a language for specification as well as an interactive theorem prover and a symbolic model checker for verification. It is available for Mac as well as Linux but offers no native support for Windows. Sources are openly available at GitHub (`https://github.com/SRI-CSL/PVS`). PVS is extensively used and supported by NASA which also added a collection of formal proofs and theorems to the platform. Documentation and a textbook explaining the language and all of the contained verification tools is available at the website.

Info	description	interactive verification tool based on custom language for high level description of complex systems
	method	proof assistant
	website	https://pvs.csl.sri.com/
	verifiable properties	functional correctness, safety, security, refinement
	current version	7.1 (2020)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✗ (OSX, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI, Plug-in (VSCode)
	open source	✓
Expertise	no custom programming language	✗(PVS)
	common specification language	✗
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	interactive
Scalability	scalability	industrial use cases

Table 3.26.: Description and maturity evaluation of tool: PVS

3.4.2.24. Tool - SeaHorn

SeaHorn is an open-source tool for the analysis of LLVM-based languages. The source code is available on GitHub (`https://github.com/seahorn/seahorn`). The analysis is based on the transformation of the underlying problem into horn clauses and applying a suitable solver to it. SeaHorn specializes in checking for termination as well as finding dead code. It is available as docker image and as such compatible with all major operating systems. Publications are listed online however a textbook and extensive documentation is missing.

Info	description	A fully automated analysis framework for LLVM-based languages
	method	model checking, abstract interpretation
	website	https://seahorn.github.io/
	verifiable properties	termination, reachability
	current version	10 (nightlies, no official release)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✗
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✓(LLVM, C)
	common specification language	✓
	textbook available	✗
	comprehensive online manual	✗
Automation	automation	fully automated
Scalability	scalability	small academic examples

Table 3.27.: Description and maturity evaluation of tool: SeaHorn

3.4.2.25. Tool - SMACK

SMACK is a model checker and verification toolchain for the LLVM intermediate language. It allows verification of all languages that can be compiled to LLVM, including C and C++. Internally, SMACK translates the given assertions into Boogie (see Section 3.2.2) and thus allows the execution of all reasoning tools that can handle Boogie. SMACK is natively supported for Unix-based operating systems only and does not provide precompiled binaries, but does provide a Docker image. A list of publications and demos can be found on the website, but extensive documentation and a full textbook are missing. SMACK is open source and openly developed on GitHub (`https://github.com/smackers/smack`).

Info	description	SMACK is both a modular software verification toolchain and a self-contained software verifier
	method	model checking
	website	http://smackers.github.io/
	verifiable properties	functional correctness
	current version	2.8.0 (Oct 2021)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✗
	supported OS	✗ (Linux, OSX)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✓(LLVM)
	common specification language	✓
	textbook available	✗
	comprehensive online manual	✗
Automation	automation	fully automated
Scalability	scalability	small academic examples

Table 3.28.: Description and maturity evaluation of tool: SMACK

3.4.2.26. Tool - Software Analysis Workbench SAW

SAW is a scripting language for proofs with a suitable prover. The reasoning backend used is Z3 and SAW can thus be used as a fully automatic prover. Due to the nature of being a scripting language it also allows for manual proof scripting for more complex properties. SAW is developed by the company Galois but is open source and available at GitHub (`https://github.com/GaloisInc/saw-script`). It has been used in several commercial applications, mainly for the verification of cryptographic protocols. Accepted input languages include Java and LLVM. A scientific publication introducing the tool is available at the website. Extensive documentation is available at the GitHub project website.

Info	description	a verification tool based on dependent type theory
	method	symbolic execution
	website	https://saw.galois.com
	verifiable properties	functional correctness
	current version	0.9 (Oct 2021)
Community	community	large
	commercial support	✓
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✓(C, Java, Cryptol)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.29.: Description and maturity evaluation of tool: SA Workbench

3.4.2.27. Tool - Spark ADA

SPARK is a commercially available platform which offers the Ada-based SPARK programming language and suitable verification tools for it. Language features can be disabled to increase the ease of verification and Spark is combinable with Ada and C code. Relying on the GNAT verification suite SPARK provides formal static verification to prove the absence of runtime exceptions as well as full functional correctness.

Info	description	platform with language, verification tool and design method to construct secure and safe software
	method	symbolic execution, static analysis
	website	https://www.adacore.com/about-spark
	verifiable properties	functional correctness
	current version	Community 2021 (Juni 2021)
Community	community	large
	commercial support	✓
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✗
Expertise	no custom programming language	✗(Ada)
	common specification language	✗
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	autoactive
Scalability	scalability	industrial use cases

Table 3.30.: Description and maturity evaluation of tool: Spark ADA

3.4.2.28. Tool - SPIN

Spin is a model checker specialized in the verification of multithreaded software as described in a specification language called Promela. It is open-source software and is included in Linux Debian distributions as well es Ubuntu since version 16.10. Several case studies including safety critical software for space missions were conducted with Spin. Documentation, publications, and tutorials are available at the website.

Info	description	model checker explicitly for multithreaded software
	method	model checking
	website	`https://spinroot.com/`
	verifiable properties	safety
	current version	6.5.1 (July 2020)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	CLI
	open source	✓
Expertise	no custom programming language	✗(Promela, extraction from C possible)
	common specification language	✗
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.31.: Description and maturity evaluation of tool: SPIN

3.4.2.29. Tool - TLA+

TLA+ is a high-level modeling language for systems and specialized in concurrent and distributed systems in particular. It also offers a suite of tools to verify the specified systems including a model checker. TLA+ has been used by several major tech companies including Amazon, Microsoft, and Intel. It is openly developed on GitHub (`https://github.com/tlaplus/tlaplus/`). Extensive documentation, tutorials, and publications as well as binaries can be found on the website.

Info	description	language for modeling concurrent and distributed systems with several tools
	method	model checking
	website	https://lamport.azurewebsites.net/tla/tla.html
	verifiable properties	functional correctness
	current version	1.7.2 (Feb 2022)
Community	community	large
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI, Plug-in (VSCode)
	open source	✓
Expertise	no custom programming language	✓(State machines)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.32.: Description and maturity evaluation of tool: TLA+

3.4.2.30. Tool - UPPAAL

UPPAAL is an integrated tool environment allowing for the modeling and verification of real-time systems based on timed automata. It is developed jointly at the university of Uppsala as well as the university of Aalborg. UPPAAL is free for non-commercial use but is not open source. The tool offers an elaborate GUI in which the user can model systems graphically and verify them. It is available for all major operating systems.

Info	description	verification of real-time systems based on timed automata
	method	model checking
	website	https://uppaal.org/
	verifiable properties	safety, liveness properties of temporal automata
	current version	4.015 (11.2019), dev version 4.1.26-2(5.10.2022)
Community	community	large
	commercial support	✓
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✗
Expertise	no custom programming language	✗(custom timed automata)
	common specification language	✗
	textbook available	✗
	comprehensive online manual	✓
Automation	automation	fully automated
Scalability	scalability	industrial use cases

Table 3.33.: Description and maturity evaluation of tool: UPPAAL

3.4.2.31. Tool - VeriFast

VeriFast is an open-source tool developed at the University of Leuven. It is available for all major operating systems. Based on separation logic, VeriFast is able to show correctness properties of singlethreaded and multithreaded C as well as Java programs. The fully automatic reasoning is done using an SMT solver. Rudimentary documentation as well as the main publications concerning the tool are available at the GitHub page.

Info	description	separation logic based tool for Java and C programs
	method	deductive verification (batch)
	website	https://github.com/verifast/verifast
	verifiable properties	functional correctness
	current version	Nightly – Jan 22 (offiziell 21.04)
Community	community	medium
	commercial support	✗
Availability	precompiled binaries	✓
	supported OS	✓ (OSX, Win, Linux)
	up-to-date	✓
	standalone	✓
	user interface	GUI
	open source	✓
Expertise	no custom programming language	✓(Java, C)
	common specification language	✓
	textbook available	✓
	comprehensive online manual	✗
Automation	automation	autoactive
Scalability	scalability	multiple academic examples

Table 3.34.: Description and maturity evaluation of tool: VeriFast

3.4.3. Language Support

There are two different ways of looking at programming languages. Is a language good for verification? Is a language widely used in industrial applications? Strongly typed languages with limited type conversions, such as Java or C#, are much easier to reason about and it is easier to prove properties of programs written in these languages. Functional languages have no side effects and show advantages when reasoning about concurrency. However, languages with hard-to-control type conversions, such as C/C++, are widely used in industry and are much harder to reason about.

There are also languages such as C# or Rust with support for marking safe and unsafe regions of code. With this separation, the hard-to-verify unsafe sections of code tend to be small. C# is typically used for application development, so unsafe code regions are rarely used. In contrast, Rust is used for both application and system development, so unsafe code regions are used more frequently.

3.4.3.1. Java

Java is a programming language that is still used in many financial and enterprise systems. Java is typically compiled into portable JVM bytecode. Along with the other JVM-based languages such as Kotlin or Scala, it is also used to develop applications for the Android ecosystem used in phones, TVs, and cars. Several formal verification tools are available for Java and JVM-based languages, including CRUX, Java Path Finder (JPF), JBMC, JDart, Jimple, Krakatoa, Why, KeY, Isabelle/HOL, Jinja, OpenJML, and Software Analysis Workbench (SAW).

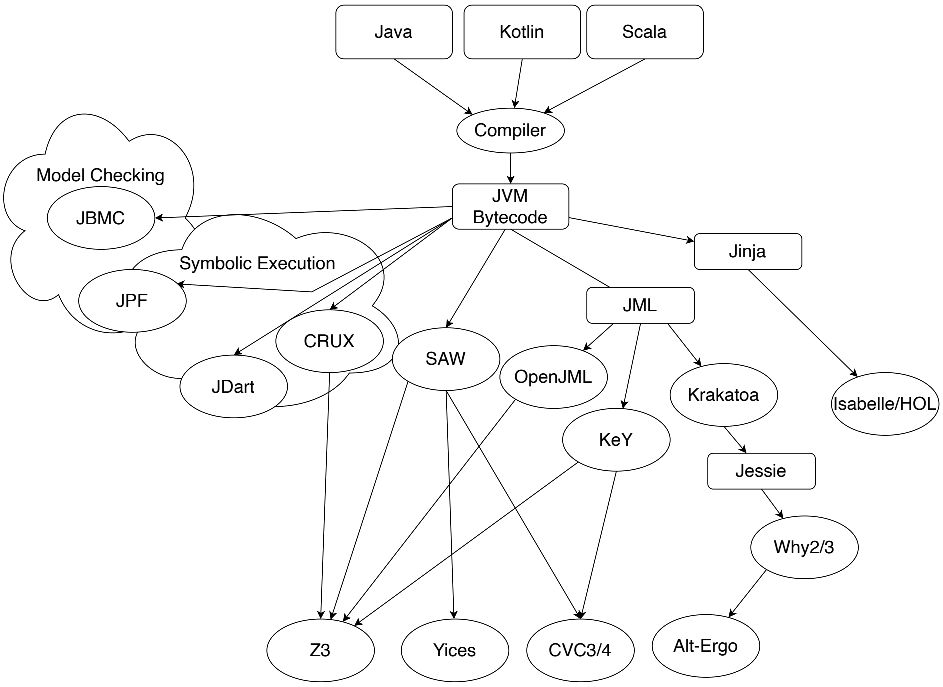

Figure 3.1.: Overview on common verification tools for Java

Figure 3.1 shows that most tools rely on the precompiled Java bytecode. Moreover, a strong usage of the JML specification language is shown (circles denote tools and squares represent used languages or specifications).

3.4.3.2. Functional Languages

Functional languages such as Erlang, Haskell, OCaml, or ReasonML usually have good support for formal verification, since pure functions have no side effects and are usually supported by a strong type system. The Coq prover is a common choice for verifying programs written in a functional language, and it is possible to generate Haskell and OCaml code from Coq as part of the verification process. However, industry adoption of functional languages is low. Currently, ReasonML, which is tightly coupled to Facebook's React framework, seems to be attracting developers (see Section 4.1.3 on JavaScript as an application example for details).

3.4.3.3. C/C++

While C and its object-oriented counterpart C++ are both programming languages that are often considered to not be modern languages, they are still very important[5]. Especially in the area of embedded and operating system software, both languages are used. Several standards, such as MISRA-C and MISRA-C++, have been established and are used to develop safety-critical systems ranging from defense and aerospace to telecommunications and medical devices. Therefore, it is obvious that formal verification is a topic of high relevance and numerous tools have been developed, including Astrée, PolySpace, CPAchecker, CBMC, Frama-C, SMACK, Corral, Software Analysis Workbench (SAW), SeaHorn, KLEE/Kleaver, Isabelle/HOL, and C-SIMPL.

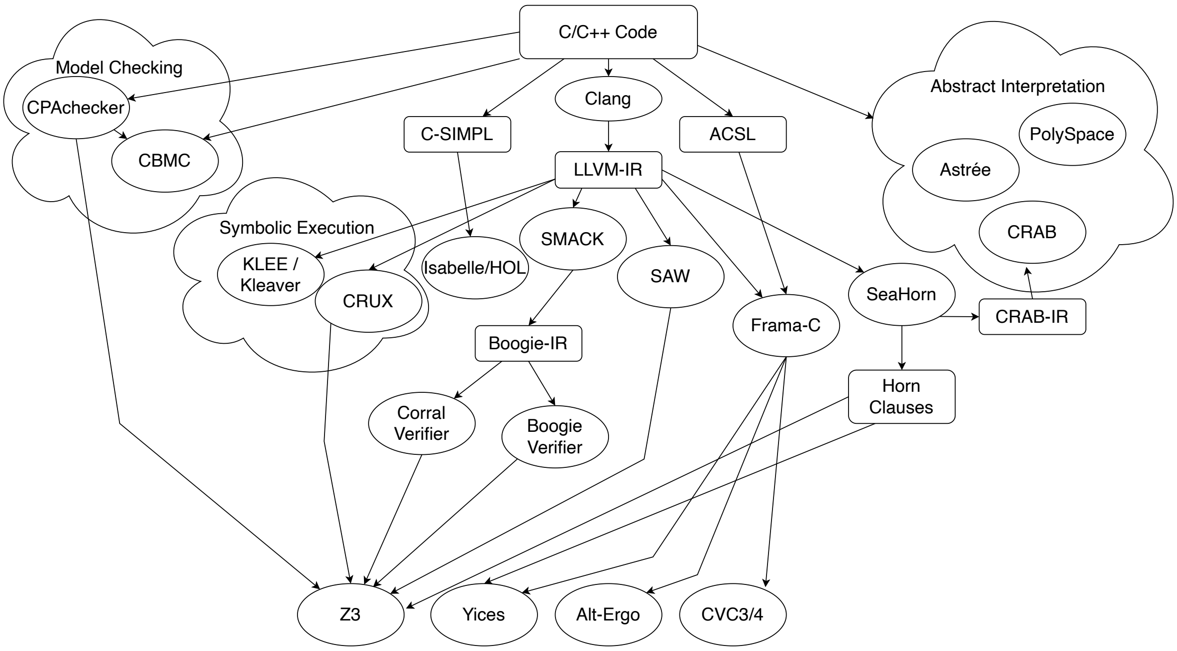

Figure 3.2.: Overview on common verification tools for C/C++

Figure 3.2 shows the dependencies for some C/C++ verification toolchains (circles represent tools and squares represent languages or intermediate representations). Many tools are based on the LLVM framework, which facilitates the development of language frontends for C/C++ and other LLVM-supported languages. The planned introduction of contracts for C++ in the upcoming C++23 stan-

[5] Also reflected by their rankings in the TIOBE Index 2022: `https://www.tiobe.com/tiobe-index/`

dard may further increase the adoption of formal verification among C++ developers, as it standardizes the specification of pre- and post-conditions.

3.4.3.4. Rust

Rust is a relative new language that has gained large attention by the formal verification community due to its ownership-based type system and useful guarantees about Rust programs that simplify reasoning [Astrauskas et al. 2019]. Moreover, Rust has been recently allowed to be included in the Linux kernel as alternative to the traditional C. Rust uses a Mid-level Intermediate Representation (MIR) that is then converted to LLVM-IR for compilation. Thus, many LLVM-based verification tools could be easily adapted to support Rust as well, as shown in Figure 3.3 (circles denote tools and squares represent languages or intermediate representations). Based on Rust, currently an ISO 26262 compliant toolchain is being developed called Ferrocene[6]. Verification tool support for Rust includes: KLEE, SMACK, SeaHorn, Crux-MIR, Prusti / Viper, and MIRAI.

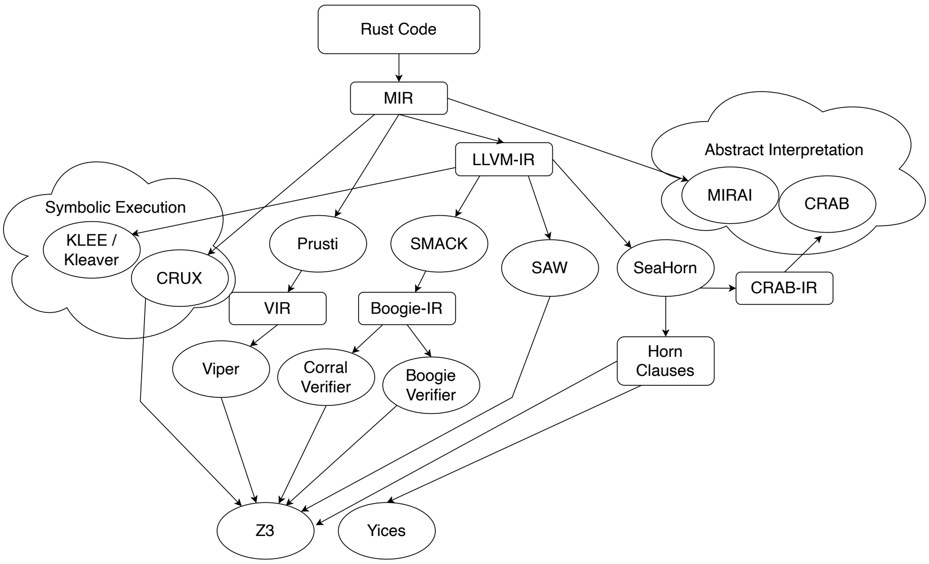

Figure 3.3.: Overview on common Rust verification tools

3.5. General Limitations

Although a variety of mature software verification tools exist, there is no silver bullet for general software verification. All tools have drawbacks and limitations. In this section, we discuss the most relevant classes of limitations that apply to software verification tools, distinguishing between three different categories of limitations: those related to the target language, those related to the verifiable properties, and those related to the underlying verification method of the tool.

[6] Ferrocene: https://ferrous-systems.com/ferrocene/

3.5.1. Programming Languages Features

Most tools do not support verification of arbitrary languages, but rather focus on verifying programs written in a particular language (or set of languages). Furthermore, for real-world programming languages such as Java, C, or C++, most tools do not support all features of the language, but rather a subset. For Java, a commonly used subset is to support the JavaCard standard, as the KeY system does [Ahrendt et al. 2016]. This problem becomes even more apparent when considering de-facto-standard libraries, e.g., for handling strings. Such libraries are theoretically not an essential part of the language, but reasoning about real-world programs without support for such libraries is almost useless. For that reason, many tools provide built-in support for such libraries. Taken together, the language itself and the language features supported (or rather not supported) by a tool describe a class of limitations typical of software verification.

Concurrency A challenging topic for OS verification in particular is concurrency – parallel execution with shared memory. Few tools can handle the concurrency features of programming languages. Mature tools such as the static analyzer Astrée or the model checker TLA+ support the verification of concurrent software; however, reasoning about concurrent software does not scale well (see discussion of scalability dimensions in Section 4.2).

The no longer maintained deductive verification tool VCC [Cohen et al. 2009] developed by Microsoft has been used to verify (parts of) the Hyper-V hypervisor for Windows. So far, only the CertiKOS project[7] within the DARPA HACMS program [Fisher et al. 2017] demonstrated how to build a verified concurrent OS kernel (named mC2) written in C and assembly code [Gu et al. 2019]. CertiKOS uses system layers which are independently verified using Coq and enables proving information flow properties for software consisting of C and assembly code [Costanzo et al. 2016]. Lorch et al. [Lorch et al. 2020] demonstrated with their Armada language and tool how to generate verified high performance concurrent C code using refinement and the Dafny verification tool/language (see also Section 3.1.3).

The VerCors framework uses a JML-style specification language for Java, C, OpenCL, and OpenMP and translates to the Viper intermediate language before proving. The focus of VerCors is on application and not OS level, see [Monti et al. 2022] for an industrial use case. Verifying distributed systems – parallel execution without shared memory – is discussed in Section 4.1.2.

Embedding and Mixing Languages When different programming languages are embedded or mixed, as is often done with assembler within C code, there are several ways to handle this: (1) both languages are translated into a common abstraction [Costanzo et al. 2016], (2) one of the languages is translated into the other [Recoules et al. 2019], and (3) two different tools are used to verify the languages separately, assembling the results.

3.5.2. Verifiable Properties

To be verifiable, a property of a program must be expressible in some kind of formal language. This language is typically called the specification language. The specification language(s) supported by a tool thus directly limits the properties that can be verified by that tool. If the specification language supported by a tool is based on propositional logic, then first-order logical properties are not

[7] CertiKOS project: `https://flint.cs.yale.edu/certikos/index.html`

expressible in that language, and thus cannot be reasoned about and verified. This is the second class of limitations of verification tools: The properties that can be reasoned about by the tool.

3.5.3. Limitations regarding the Underlying Method

The last class of limitations are those regarding the underlying methodology of a tool. Consider as an example a bounded model checker like CBMC. A bounded model checker inherently is unable to prove statements about programs with unbounded loops. So the choice of the underlying method is a direct limitation to what a tool based on that method is able to verify.

3.5.4. Tradeoff Between Limitations

It is important to consider that tool limitations are often willingly accepted in order to gain an advantage elsewhere. In many cases, accepting limitations, regardless of their class, enables further automation capabilities. This is the reason why tools that support verification of arbitrary properties for arbitrary languages are very rare, as they usually have to pay for this generality with a severe loss of automation. Conversely, it is very common to sacrifice the generality of an approach in order to provide a fully automated tool. See Figure 3.4 for an illustration of this relationship.

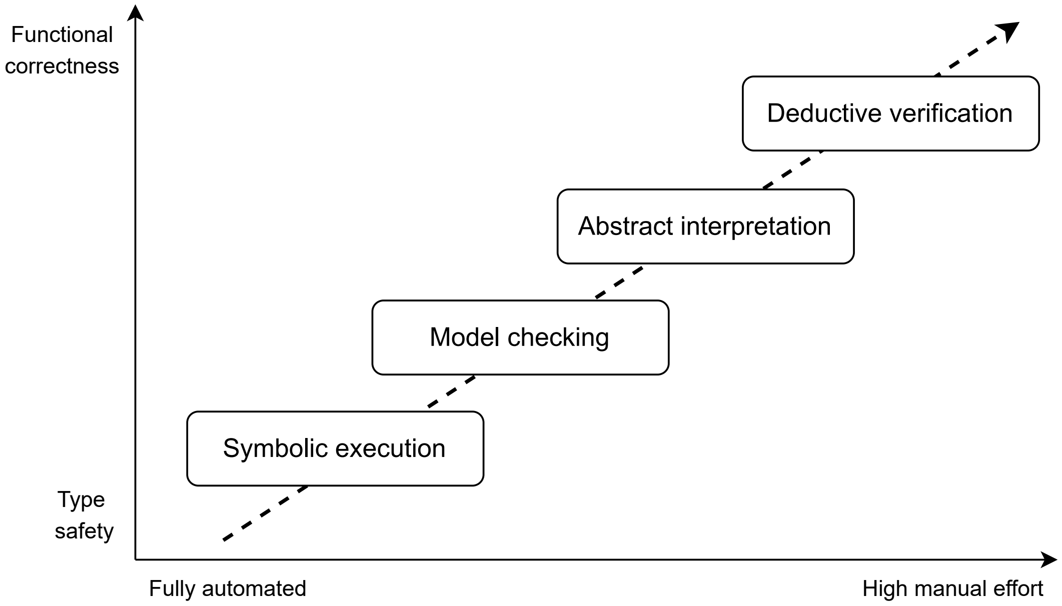

Figure 3.4.: Relation between the degree of automation and the complexity of verified properties (adapted from [Fisher et al. 2017]).

3.6. Conclusion

Our research on the state of the art of software verification tools reveals **numerous mature methods, tools, and communities** – many of them driven by academic stakeholders. Comparing this situation with closely related areas of software engineering tooling, such as testing or automated bug finding, we can conclude that **there has not yet been widespread adoption by industrial software developers**. For both testing and bug finding, we have seen inflection points where demand from industrial developers led to a rapid increase in the attractiveness of a few tools (JUnit for testing, or FindBugs and PMD for automated bug finding). This focus on a few tools enabled multiplier effects through a huge increase in media coverage and availability of training materials and services. And, thus, a rapidly accelerating adoption of testing and automated bug finding by industrial users. After the phase of focusing on a few tools, we have seen a more differentiated tool landscape with the entry of commercial vendors as well as new academic projects bringing new innovations to the field.

The same is true for modeling languages, where many formalisms have been introduced until the Unified Modeling Language (UML) has been adopted as the standard by industrial developers.

The reasons preventing formal verification from making a similar breakthrough are not entirely clear. However, there are a number of issues that make its use in industrial practice difficult. First and foremost, **writing concise specifications is very time-consuming**. Further, **formal methods are harder to integrate seamlessly into existing build chains** than testing. In addition, **writing tests is closer to the mental model of most software developers than writing specifications**. A test case makes an assumption about a single or a few traces of program logic, while predicates used for specifications must apply to all possible traces. This makes it easier for developers to write tests. Finally, unlike tests, formal verification may not provide concise counterexamples for failed proof attempts (along with debugging features). However, this is changing rapidly.

As discussed in Section 3.4.3, many verification tools support Java, functional programming languages, or some custom language, while industry demand is for C/C++. There is a mismatch between supply and demand. The growing adoption of Rust shows that **developers prefer to move to a new language that is type and memory safe, rather than go through the hassle of verifying C/C++ programs**.

There is very little information on the web about how to integrate verification tools into common build chains. The reason for this is that, with the exception of simple static analysis such as type checking, verification tools cannot be fully automated and, in particular, the **proofs performed cannot be transferred to new code versions**. In contrast to verification, tests can be easily automated and repeated for new code versions. Usually, only few tests need to be adapted, if at all.

These problems could be mitigated by a good choice of tools according to one's own needs. However, **the investment required to fully master a verification tool is likely to be so high that changing tools becomes uneconomical and a lock-in effect occurs**. This is also true for programming languages. However, there are often solutions that allow more interoperability. Often there are interfaces that allow the integration of other programming languages or wrappers that allow components written in different languages to be used in the same software system.

4. Scalability

Before discussing the three main dimensions of scalability, we start this chapter with examples of successful applications of software verification. Afterward, we propose a scalability model to enhance the state of the art.

4.1. Applications of Software Verification

After presenting tools and their corresponding developer communities in the previous chapter, we now show examples of applications of software verification. Some user communities are closely related to specific tools. Other user communities are independent and rely on different tools to verify different properties. The selection of the following example applications is motivated by the idea of a fully verified software stack of a basic IT systems – a system that consists entirely of verified components, from the operating system to applications such as a web browser.

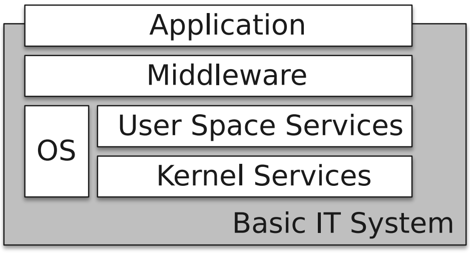

Figure 4.1.: Software stack of a basic IT system.

We chose operating systems, communication and cryptographic libraries, and JavaScript engines as examples. As a rather small number of operating systems is used to build most of today's systems, successful attacks against them have a huge impact. In turn, providing a fully verified operating system can thus eliminate a whole range of possible attacks at once and is a necessary component for almost all systems. The same is true for widely used communication and cryptographic libraries. A well known example of a compromised library with huge impact is the heartbleed bug in the OpenSSL library. As a last example, we chose JavaScript engines as they are one of the most crucial components for web browsers, since they execute remote code from generally untrusted sources.

4.1.1. Operating Systems

For more than 20 years, Microsoft has been providing the Static Driver Verifier (SDV) tool, which uses static analysis of C code to ensure that device drivers correctly use the Windows kernel interfaces and to verify their memory safety [Ball et al. 2011]. SDV uses Corral and Boogie for verification, which leverage the Z3 SMT solver [Pavlinovic et al. 2016] as reasoning backend. There even exists a benchmark for Boogie based on collected SDV data[1]. This is an interesting example as the very

[1] Microsoft SDV Benchmarks: `https://github.com/boogie-org/sdvbench`

narrow application case allows for a general set of specified properties, which alleviates the burden of specifying each driver manually.

Recently, the Linux developers added a lightweight runtime verification approach based on online monitoring of states to the Linux kernel v6.0 for usage with safety critical systems[2]. This approach allows the system to monitor the execution trace of system events and compare it to a specification using finite automata models [Oliveira et al. 2019]. The system can then react on detected deviations and either log a warning or the kernel enters a panic state. Using this method, developers can potentially detect and identify unexpected interactions and potential security and safety flaws, e.g., in drivers.

With regard to operating systems, the community around the seL4 microkernel is the most active. Important properties were successfully verified for the seL4 microkernel for the first time in 2009 [Klein et al. 2009]. This was later extended to include other properties and verification of the correctness of the binary code. Isabelle/HOL is the main verification tool used. Since 2020, the seL4 Foundation organizes the further development of seL4[3]. An interesting contribution of the original verification of the seL4 kernel was the demonstration of the effort required to verify such a complex system. They reported that the pure verification effort without the development of the tool or the gaining of expertise was about 11 person years [Klein et al. 2009].

In October 2022, Google announced the development of CantripOS – an OS written in Rust and based on the seL4 microkernel[4]. While it is currently unclear, which properties shall be verified for the CantripOS components, its availability under Apache license and the use of Rust could attract a huge community. Moreover, Google works on a low-power embedded platform called Sparrow as the reference platform for CantripOS.

There are two seL4-based OSs with commercial licenses and support available – TRENTOS[5] by HENSOLDT Cyber and Kry10 OS[6] by Kry10. While Kry10 OS is rather new, TRENTOS has been available for several years and HENSOLDT provides an SDK usable with the Raspberry Pi 3 board.

Regarding OS data structures for inter-process communication (IPC) Meta reports on verification using Iris/Coq [Carbonneaux et al. 2022], while Amazon reports on verifying FreeRTOS IPC using VeriFast [Chong and Jacobs 2021]. The activities of Google, Meta, and Amazon on software verification are important for the further development of the topic. They usually tie their investments in software security to the establishment of open source communities in order to benefit from scaling effects of these communities. Google is a member of the seL4 Foundation.

4.1.2. Communication and Cryptographic Libraries

For communication and cryptographic libraries there is the project Everest[7] aiming at a provably secure communication stack [Bhargavan et al. 2017]. Project Everest covers a TLS 1.3 record layer implementation, the HACL cryptographic libraries, and the QUIC protocol record layer. Parts of these implementations are used in the Linux kernel, as well as the Mozilla web browser. Most parts use F* and the corresponding verification tool chain. Some parts use Dafny. The libraries are delivered as C or high-performance assembler code.

Furthermore, specialized verification tools like ProVerif, Tamarin, EasyCrypt and CryptoVerif exist,

[2] Linux Kernel Runtime Verification: `https://docs.kernel.org/trace/rv/runtime-verification.html`

[3] The seL4 foundation: `https://sel4.systems/`

[4] CantripOS: `https://github.com/AmbiML/sparrow-cantrip-full`

[5] TRENTOS (seL4-based OS): `https://www.trentos.de/`

[6] Kry 10 (seL4-based OS): `https://www.kry10.com/`

[7] Project Everest: `https://project-everest.github.io/`

that are commonly used to reason about cryptographic properties and security of protocols. Hassan et al. give a comprehensive overview of these tools [Hassan et al. 2021].

In the domain of distributed systems, the P framework[8] together with the P language is currently getting momentum. It provides a state-machine based language for the formal specification of complex distributed systems and is apparently used within the Amazon AWS team to analyze complex protocols. The P framework toolchain uses model checking and symbolic execution (using SMT and SAT solvers like ABC, Z3, CVC or YICES) to reason about properties like concurrency. The framework can generate C, C#, and Java based code (see also Section 3.1.3).

The model checker mCRL2 supports verifying distributed systems and protocols, too.

4.1.3. JavaScript Engines

There is currently little work on verifying JavaScript engines, which are crucial components of web browsers. While some research addressed non-optimizing just-in-time compilers (JIT) for JavaScript, only [Brown et al. 2020] reports work on verifying state-of-the-art optimizing JITs. Using a tool named VeRA, they verified the range analysis of JITs.

There is a lot more research on verifying JavaScript code itself. From a security view point, this is not a replacement for a verified script engine of a browser. However, it supports web applications from trusted sources. Plain JavaScript without the use of a strict type system is hard to reason about. Therefore, current work on securing JavaScript focuses on type safe replacement languages, which can be translated to JavaScript to be integrated into JavaScript applications and tool chains. Facebook has introduced the OCaml inspired Reason language[9], closely integrated with their React framework. Reason can be translated to JavaScript as well as OCaml and thus use any verification tooling of the OCaml community. The VerifiedReact project[10] directly targets reasoning about applications built using Reason/React. Google has introduced the Dart language[11] which can be translated to JavaScript, too. Dart continues to add built-in security properties, like null-pointer safety. However, Dart lacks a verification community comparable to Reason.

4.1.4. Conclusion

As shown in this section, there is quite a lot formal verification done in industrial applications. But at the same time, the examples indicate that software verification is mostly advanced and promoted by academic stakeholders and large IT companies like Microsoft, Google (Alphabet), Amazon, and Facebook (Meta) – in particular their research divisions. This does not reflect the software market, with numerous small and mid-sized companies. However, one has to keep in mind that applications in aviation and some other critical areas are often referred to but are very hard to report about due to the lack of transparency in these fields. This means that no matter what is done in these areas, their push towards better and more applicable tools is very limited. They rarely formulate requirements to tool communities. In exception, formal verification done by NASA is more transparent. DARPA funded research has to be transparent, at least partially.

Currently, many activities are focused on type checking and memory safety, which is partially achieved by switching to languages like Rust or ReasonML. It is unclear, if we will see more model checking too.

[8] P Framework: `https://p-org.github.io/P/`

[9] Reason: `https://github.com/reasonml/reason`

[10] VerifiedReact: `https://github.com/imandra-ai/verified-react`

[11] Dart programming language: `https://dart.dev/`

There is a notable shift in the interest for software verification. Microsoft static driver verifier has been introduced to keep system available and prevent Windows system crashes caused by faulty drivers developed by device vendors. In contrast, securing cryptographic libraries or JavaScript code is clearly dedicated to prevent attacks and information leakage. This appears to be a general trend towards security rather than safety, as the fear of targeted cyberattacks is growing in industry as well as governments.

4.2. Three Dimensions of Scalability

Scalability in the context of software verification can be considered in three different dimensions: the software, the properties, and the method used. We will briefly discuss each of these dimensions.

The software to verify One may think of the size and complexity of the software being verified as as the most important dimension when considering scalability. This is perhaps the most natural aspect of scalability, i.e., a particular method or tool scales well if it can verify larger pieces of code (in the same amount of time). In this dimension, however, we will consider not only the size of the code to be verified, but also its complexity. We will intentionally not define exactly what we consider complex software, as what is considered complex may vary from tool to tool. However, typical examples of code properties that are considered complex are concurrency and floating-point arithmetic. Depending on the language (or type of language) used, other features may also add complexity, such as generic data types, lambda expressions, or even complex data types.

Following this definition, we consider a tool or method to scale better if it can handle larger or more complex code (in the same amount of time).

The properties to verify As a second dimension, we consider the complexity of the properties to be verified. Similar to the complexity of software, the complexity of properties is not uniquely defined and thus cannot be objectively determined. As with software complexity, metrics would only be useful for the narrow purpose for which they are defined. Again, the approach used to verify a particular property has a direct impact on how complex that property is considered to be, since some tools are either specifically designed for, or at least focus on, certain types of properties and are thus naturally suited for them. However, if a particular class of properties is a subset of another class of properties, the latter may objectively be considered more complex. In this sense, functional correctness properties are probably the most complex properties to verify, since functional correctness subsumes most other properties, including lightweight and generic properties, safety, and termination. In this sense, we consider a tool to scale better if it is able to verify a more complex property of the same program (in the same time frame).

The verification method The final dimension of scalability is the verification method used. Here, we consider limitations that are inherent to a particular approach or method. Examples of such limitations might be the inability of a bounded model checker to provide guarantees above a specified threshold, the need for deductive verification tools such as KeY to require a huge amount of auxiliary specifications, or the degree to which a tool is able to reuse proofs from a previously verified version of the same software. This dimension is probably the most difficult to evaluate objectively, since these limitations are very different in nature, and the impact that such a limitation may have in a given situation varies significantly with the scenario under consideration. However, we still consider this dimension to be very relevant, as it is the fundamental limitations associated with certain approaches that may prove to be the most difficult to overcome, and thus should be carefully considered when choosing a particular approach or tool for a verification task.

Correlation of the three dimensions We have already indirectly mentioned that the different dimensions of scalability are correlated. We often observe that tradeoffs must be made between the size of the system that can be studied and the complexity of the properties that are being examined. Similarly, approaches that accept major limitations may be able to verify more complex code or larger portions of code than approaches that try to impose as few limitations as possible on the user. In a perfect world, we would have a tool that has no constraints and verifies arbitrarily complex properties for large systems. However, such a system is very unlikely to ever exist for the reasons mentioned above, so careful considerations must be made. We think that the dimensions presented here can provide a good basis for comparing different approaches and tools.

4.3. Scalability Model

In the previous section, we introduced three dimensions of scalability for software verification. Improving any approach or tool in any of these three dimensions can be considered an improvement in scalability. With this in mind, we propose a scalability model aimed at improving the scalability of formal methods in a general sense. Specifically, we will not talk about detailed minor adjustments to individual tools to push the limits of what they can do, but rather in a more general sense about how formal methods for software verification can be scaled to larger systems.

We envision four major ways to achieve such an improvement in scalability:

- reduce the amount of **specification** needed
- enable **composability**
- enhance **parallelization**
- reuse **proofs**

Reducing specification effort For a long time, the bottleneck for verifying functional correctness has been the development of appropriate specifications [Baumann et al. 2012]. So it makes sense to investigate ways to reduce the effort required. We need to carefully distinguish between requirement specifications, which are the specifications needed to describe the property one wants to verify, and auxiliary specifications (e.g., loop invariants), which are only needed for some tools to be able to find proofs. While one can argue that in an ideal world auxiliary specifications can be avoided completely, the requirement specification will always be necessary. Thus, one way to reduce the specification effort is to reduce the amount of auxiliary specifications needed. Depending on the approach, this varies considerably, e.g., bounded model checkers often need no auxiliary specifications at all, while deductive verification tools tend to need a lot of them. There are already several approaches that aim at automatically inferring auxiliary specifications (mainly loop invariants). One idea to increase scalability is to extend these approaches and try to use them to provide even more auxiliary specifications. For requirement specifications, this is (as noted above) not so easy, but one could imagine tools and languages that reduce the effort of writing such requirement specifications.

An interesting new area of research is how AI can provide new ways of specifying programs by simply informally telling a system the underlying idea of some piece of code (e.g., “This is a sorting algorithm”) and having the system automatically propose an appropriate formal specification – which the user still has to inspect for its appropriateness and possibly adapt but not write from scratch. One system that is already able to do this for simple specifications is ChatGPT[12]. Considering the amount of specification that is often required for complex systems, this could be a huge factor in scaling.

Enabling composability When describing the dimensions of scalability, we mentioned that different tools and approaches have different weaknesses and strengths across all dimensions. Thus, a natural consideration to improve scalability is to try to combine different tools and approaches in a beneficial way. We are convinced that this is a very promising line of thinking, and one that has already been adopted to some extent. In Section 3.1, where we described the different formal approaches, we mentioned several ideas on how to combine different concepts to achieve better results. A famous example is the use of symbolic execution in model checkers to prevent state explosion.

We envision two high-level ways to combine different tools advantageously. The first idea is to use different tools for different types of properties. For example, it might be promising to prove functional correctness with a different set of tools than those used for lightweight properties. Using this approach, each tool could specialize in a set of properties and be used specifically for that set. Other tools may rely on already proven properties as additional assumptions, which could make the proof easier. Extending this idea, the second way to combine tools is to use tools together to prove complex properties. Often, especially fully automatic provers are not able to establish a proof for very complex properties, but succeed quickly once some additional lemma is given. One obvious way to do this is to use different tools (e.g., fully automatic and interactive) to first (manually) prove a complex lemma and then use that proof to establish the desired property. Both ways, however, require that the tools used are somehow able to interoperate. The easiest way to achieve this is for both tools to be able to understand the same specification language.

Enhancing parallelization It may sound like a trivial idea, but parallelization has probably a lot of potential when scaling formal methods for software. Automatic reasoning like solving SAT or SMT problems proved to be very hard to parallelize, however verification tools which have to prove complex properties can parallelize on a higher level. Especially in combination with the previous aspect of combining different tools, this could lead to interesting performance enhancements. This is an alternative to waiting for more powerful hardware.

Particularly, quantum computing is often considered as a silver bullet for all computational hard problems. However, currently hard to solve SAT or SMT instances are rather large, i.e., encoding them on a quantum computer needs many qubits. This prevents SAT/SMT solving from qualifying as a short- or midterm application of quantum computing. Since SAT and SMT solving are core algorithms for many applications, much research on quantum algorithms is done, e.g., see the state of the art report on quantum SAT algorithms published by the European NEASQC project[13].

Proof reuse In Section 4.1, we provided several examples of relevant systems that have already been verified. A natural idea to increase scalability is to make use of existing proofs. This can be done

[12] Conversational language model - ChatGPT: `https://openai.com/blog/chatgpt/`

[13] NEASQC report on quantum SAT: `https://www.neasqc.eu/several-neasqc-deliverables-published/`

in two ways. Either build a system on top of components that have already been verified and thus inherit the guarantees, or have a system that is able to extend on top of proofs. While the first version of reusing proofs is easier to do, it also requires that the exact guarantees provided are sufficient for one's own use. The second idea would be to be able to extend proofs, for example by adding additional properties. This, however, requires a sophisticated approach to reusing and extending proofs that have already been done. In any case, we think that both ways should be considered when building new systems.

Intentionally not considered For the sake of completeness, we mention that a well-established way to provide scalable formal methods is to give up on soundness. This basically means that one is willing to accept that no formal guarantees can be established in order to achieve better scaling. Many prominent companies such as Meta, Amazon, or SAP have systems that are of this type – for example, Facebook's static analysis tool infer[14]. These systems provide alerts that developers can then review and fix potential bugs, but not getting an error does not guarantee that the system is correct. Since we have explicitly stated that we do not consider such approaches because they do not provide formal guarantees, we exclude this approach as a way to achieve scalability here. It is worth mentioning, however, as it is often applicable to industrial-scale code bases.

Combination of the improvements We have already hinted at the possible combinations of the individual scalability improvements. We envision combining all of these approaches into a single development model to facilitate the implementation and verification of a fully verified software system. We believe that in order to achieve the best possible scalability, improvements in each of the dimensions are necessary. Specific recommendations on how to achieve this are given in Chapter 5. Many of these recommendations build on each other, and thus form a kind of natural combination.

[14] Infer – static analyzer for Java, C, C++: `https://github.com/facebook/infer`

5. Recommended Actions

With the vision of verifying the full software stack of complex systems, we see three important goals for research and community action:

1. improving the verification of small critical components with respect to individual properties,
2. improving the robustness of verification with respect to changes in software and requirements, and
3. improving the composability of verified components and different properties to fully verified complex systems

Figure 5.1 visualizes the three goals and the fact that robustness to change is needed for both small components and for complex systems.

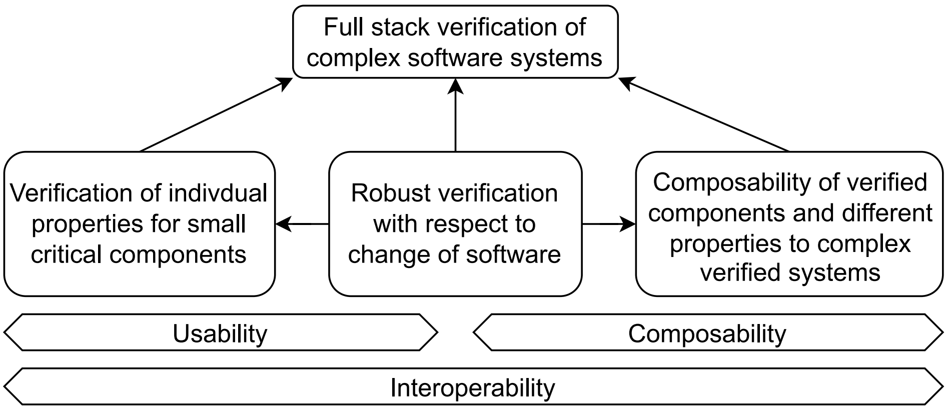
Full stack verification of complex software systems
Verification of indivdual properties for small critical components
Robust verification with respect to change of software
Composability of verified components and different properties to complex verified systems
Usability
Composability
Interoperability

Figure 5.1.: Goals towards the vision of verifying the full software stack of complex systems

We make the distinction between verifying a **single property for small components** and verifying **multiple properties or complex systems** because **we believe the former is possible with today's technology and tools**. However, depending on the chosen property, the programming language to be verified, and the availability of specifications, the necessary investment to verify a small component can be very high. Therefore, research should focus on improving usability and interoperability to reduce the effort and investment required.

In contrast, the verification of properties for complex systems is still an open question. From a theoretical point of view, composability is given, since all formal verification methods – and the proofs they generate – are based on some kind of logic. In practice, however, **combining proofs from different tools remains a challenge**. As a result, we believe that the major research effort should be directed toward making the use of formal methods more scalable. For this reason, most of our recommended actions are direct consequences of the observations made in Section 4.3.

Achieving the three goals above will require many research and community actions. In the next section, we describe appropriate actions and what a roadmap to a fully verified software stack might look like. We distinguish between community actions and research actions. Community actions are intended to stimulate the interest of industrial developers in software verification. Research actions will address fundamental open problems in software verification. Table 5.1 gives an overview of the two proposed community actions and seven research actions.

	Action	**Goal**	**Target**	**Risk**	**Time**	**TRL**
C1	Reference applications and platforms	Focus research efforts / easily enter research area	s(c)	l	<2	a
C2	Guidance for verification	Guide users which tools and properties to address to maximize benefit	s	l	<2	a
R1	Interoperability	Enable users to switch tooling if required	s	m	2-5	a
R2	Robustness to change	Reduce cost to verify changed code	s	h	2-5	f
R3	Demonstrate concurrency with Rust	Improve state-of-the-art of verifying concurrent systems	s	m	2-5	f
R4	AI-based generation of specifications	AI-based assistance for generating (auxiliary) specifications	s	h	2-5	f
R5	Composability of properties	Demonstrate tooling to prove a set of properties	s	m	2-5	a
R6	Composability for complex systems	Manage coverage of proofs for complex systems	c	h	>5	f
R7	Robustness to change of complex systems	Manage impact of changes to complex systems regarding verification	c	h	>5	f

Table 5.1.: Overview of recommended community and research actions (Target: small components, complex systems, Risk: low/medium/high, Time in years, TRL = technology readiness level: applied research, fundamental research)

5.1. Community Actions

5.1.1. C1: Reference Applications and Platforms

As a seed for a high-impact software verification community, providing reference applications and platforms could be a good starting point. They provide an incentive to participate in a community, to integrate own methods and tools, and to compete with others. They also ensure more comparable research results. Reference applications should be tightly coupled to a platform to include low-level OS software and to enable links to hardware verification activities.

As a suitable reference application and platform, we envision **a minimalistic board** (e.g., Risc-V ESP32) **with a minimal software stack** (OS, a few drivers, and a demo application with minimal functionality). The OS could be very limited, with a fixed schedule and no support for user-level concurrency. Essential for the selection of this system would be to minimize the number of external input channels (especially excluding Internet connections), to minimize the number of devices and thus drivers needed to run the application, and to minimize the code complexity of the application itself. The main goal of such a first reference platform would be to gather information and experience in verifying whole systems rather than individual components, as is most often done in research. For the hardware platform, Google's Sparrow board or HENSOLDT's Raspberry Pi 3 based TRENTOS SDK could be used as a reference (see Section 4.1.1 on applications of software verification).

An advantage of starting with such a setup is that it can be easily extended incrementally to include more features that are directly applicable to the ultimate goal of a fully verified system. As a long-term roadmap, we envision three such practice-oriented reference systems, each building on the experience of the previous one. We have outlined the basics for the first one above.

The second would be **a mid-range application such as an IoT system for machine control**, including more features such as concurrency, networking, and cryptographic support for key exchange.

The third reference application could be **a complete desktop system including a web browser to run web applications**. In fact, the complexity of a web browser represents a wide variety of systems and includes complex communication. This third reference application would already meet the goal of verifying complex systems. Critical to the success of each phase is that a well-established process guides the documentation and ensures that the lessons learned are used in the next phase.

Each reference application should include descriptions of the application and its use case, implementations, and functional specifications for all software components. To assemble these artifacts, each reference application could be conducted as a case study. The disadvantage of providing implementations and specifications is that the set of programming and specification languages is fixed. However, having well-defined components and interfaces allows others to use their own techniques, such as generating code from specification languages and proving the transformations (see Refinement Method in Section 3.1.3). Providing a pure natural language specification and asking the community for formal specifications was very influential in the 90s (see the famous steam boiler control [Abrial et al. 1996]) and shaped the design of many specification languages as well as the understanding of their capabilities and limitations. Today, the focus should be on performing proofs rather than writing formal specifications.

The **goal** for all of these reference applications needs not to be the development of new tools or approaches, but rather the achievement of a verified artifact and the collection of new requirements from lessons learned regarding usability, interoperability, and composability of tools.

In **estimating the effort** required for this community action, it is important to keep in mind the flexibility of the scope and the possibility of incrementally increasing the complexity of these case studies. Therefore, the time and resources needed to complete them can be defined very flexibly. This depends on which features are included in each case study.

- The minimalist board case study could be a small project lasting 1-2 years with only a few people involved (estimated 7 person-years).
- In contrast, the full desktop with web browser case study would most likely require several years with many people from multiple organizations to accomplish a verification task of that magnitude. For comparison, consider the reported 11 person-years of effort to initially verify the seL4 kernel [Klein et al. 2009].
- At least for the minimalist board, the risk of such a project failing is very low, as we argue that performing such a case study with the tools available today is more a question of time and resources than whether it is possible or not.

5.1.2. C2: Guidance for Verification

To lower the barriers to formal verification, and thus to get more developers using formal methods, they need guidance on goals and tools. For example, the Open Web Application Security Project (OWASP) successfully guides developers in securing web applications by regularly compiling a top 10 list of the most critical security risks for web applications[1]. A similar list could be compiled in the context of software verification for basic IT systems – for example, buffer overflows are still one of the most critical risks, so proving memory safety should be a top priority. **The recommendation of properties to prove should include appropriate tools**. Perhaps even convenient programming languages, if developers are able to choose them. This guidance should be done in close cooperation with the community action C1 on Reference Applications and Platforms (see Section 5.1.1).

Thinking beyond this low-threshold recommendation, one could look for **standardized sets of properties that correspond to different levels of security**. This requires a common understanding and definition of these properties. Technically, these sets of properties should be formulated in a specification language that is used by many stakeholders. For practical use, there could be ascending levels of verification in terms of cost-benefit ratio, e.g., type checking, memory safety, functional correctness.

In the long term, standardization could be extended to define a fixed set of languages and tools for the development of safety-critical systems. This would help companies plan long-term investments by knowing what tools and languages they can build on. Furthermore, a standard would push towards libraries of already verified components. This would make it easier to reuse components and reduce the effort required to develop new applications. As a final benefit, such a standard would also allow for targeted investment and research in those specific tools and languages, thereby addressing potential remaining vulnerabilities.

On the other hand, defining a standard in terms of a closed set of tools and languages restricts the choices of developers and researchers and confines innovation to a narrow space. It is therefore necessary to **find a sweet spot between unnecessarily restricting the choice of tools and languages and allowing for the benefits of standardization**. We believe that finding such a sweet spot is possible, necessary, and most likely achievable by defining clear parameters while leaving the technical details to the developers. For example, one could require software to be memory safe, but leave it to the developer to choose whether this is achieved by using a memory safe language such as Java, or by manually proving it for a language such as C. This allows further progress for powerful (e.g., in terms of memory safety) languages such as Rust, which have rapidly growing industrial adoption but currently limited support for verification. Our recommended research action R1 on Interoperability (see Section 5.2.1) between tools is intended to improve flexibility in tool selection and, in particular, to enable tool switching.

Since we have described several sequential steps in this action, we estimate the **effort** for each of these steps.

- Recommending properties to be proven based on real-world bugs is a fairly small project, lasting 1-2 years and only a few people involved (estimated 4 person-years).
- Defining sets of properties corresponding to security levels could be part of the project mentioned above, slightly increasing the effort (estimated 3 person-year).
- Standardizing a set of languages and tools for developing security-critical systems is an idea we do not recommend for the next few years, as explained before.

[1] OWASP top 10 list: `https://owasp.org/www-project-top-ten/`

5.2. Research Actions

5.2.1. R1: Interoperability

To achieve widespread adoption of formal verification, it is important to keep in mind that developers are likely to start by using highly automated tools to prove relatively simple properties. If their expectations are met, they will move on to more complex properties. And they may need to move to more powerful tools. Currently, for most tool combinations (especially when moving to more powerful methods), this means redoing specifications and not being able to reuse parts of proofs. Therefore, we recommend improving interoperability between tools to reduce the loss of investment when switching to another tool.

To improve interoperability, there should be commonly used **specification languages with the ability to express** (1) properties related to the **semantics of the target language**, and (2) **proofs or lemmas**. In addition, tool communities should be encouraged to provide transformations to and from these languages. Initial work on proof transformation is being done by the Isabelle community[2] as part of the European COST action EuroProofNet [EuroProofNet 2021]. Their focus is on exchanging proofs within the Isabelle/Coq ecosystem, while we recommend extending the scope towards exchanging properties and proofs between completely different tools based on different verification methods.

Besides the work on specification languages, this action could include the organization of challenges to demonstrate interoperability by transferring properties and proofs. Interoperability is a good way to increase competition between tool communities, especially with respect to usability.

While we use interoperability as the term for this action, Hähnle and Huisman [Hähnle and Huisman 2019] use integration. They consider integration between verification tools and methods, as well as integration with other software quality assurance approaches and tools.

Looking at the **effort** required for this action, we note the following.

- Work on (a common) specification language that is expressive enough to enable interoperability between tools based on different verification methods should be done in one (or a few) large project(s) involving many people and multiple stakeholders, lasting 3-5 years (estimated 52 to 60 person-years, depending on number of stakeholders and number of projects).

- In addition, there could be smaller projects to demonstrate transformations between specific tools. These are rather small projects, lasting 1-2 years and involving only a few people (estimated 9 person-years per project).

5.2.2. R2: Robustness to Change

The era of agile development is defined by the regular delivery of new features that users demand. This means that focusing on infrequent releases is no longer possible if you want to keep up with the competition. For formal methods to be industry ready, they must ideally be fully assimilable into standard build chains. This means that formal verification must find solutions to reduce the cost of re-verifying changed code. Verifying a subsequent code version should require an order of magnitude less effort than verifying the original code version. We argue that there are **three main approaches** to achieving this: (1) the verification tool must be able to effectively **reuse existing proofs** for the updated versions of the code, (2) one must find a way to perform only those proofs

[2] Exporting Isabelle proofs to Dedukti: `https://github.com/Deducteam/isabelle_dedukti`

that are directly **affected by the changes** made, and (3) if only the code but not the specification is changed, one can **use relational verification**.

The first idea is already implemented in several tools to varying degrees. Any proof scripting language is a step in this direction, as small changes in the code could at best result in the same proof as before, but for industrial use this would need to be developed further and involve much more sophisticated methods. For the second idea, one way to achieve this goal is to perform an impact analysis on specifications, proofs, and code to determine the specific parts of code/proofs that need to be re-verified. For the last idea, relational verification is a well-established area of research that can provide significant advantages over traditional functional verification in some scenarios. In particular, equivalence checking of programs is exactly the problem that needs to be solved here. Again, to our knowledge, there is no approach large enough to satisfy the requirements we have outlined, but it is a promising area to consider in this context.

Since this is a rather fundamental problem, we have the following notes on the **effort** required.

- Achieving significant results in this area will require at least 3-5 years of concentrated research.
- We recommend several projects, each addressing at least one of the three approaches mentioned above. Alternatively, there could be one or two large projects addressing all three approaches.
- We anticipate that there could be small projects focused on methodological improvements carried out by a few people, as well as large projects involving demonstration proof-of-concept tools carried out by many stakeholders.
- Based on the expected project mix, we estimate an effort of 48 to 58 person-years.

5.2.3. R3: Demonstrate Concurrency with Rust

Concurrency in programs is a challenge for verification methods and tools. The challenge is to consider all possible thread interactions and prove the correctness of all of them. Tools typically approach this either by ignoring concurrency altogether and focusing on sequential programs, or by considering a rather high-level view of the system with a coarse abstraction and thus dealing with the complexity. The availability of tools that can do precise reasoning for concurrent programs is very limited. Since concurrency is ubiquitous in modern processors, it is essential that verification tools can deal with it. Further research in this area is needed to be able to verify complex systems as they are used today (see the state of the art in verification of concurrent software in Section 3.5.1).

Since C has no built-in support for concurrency, concurrent C software always depends on additional libraries. This makes verification difficult. While C++ has begun to integrate support for concurrency with the C++11 standard, we **advocate Rust with built-in support for concurrency and additional memory safety**. We expect to see increasing adoption of Rust over the next years, driven by events such as the acceptance of Rust code in the Linux kernel and the NSA recommendation for memory safe languages [NSA 2022].

By demonstration, we mean having a concurrent reference application written in Rust – which is more realistic than having an OS component like the kernel or drivers written in Rust. Given this reference application, a verification toolchain should be demonstrated. As described in Section 3.4.3.4 on state-of-the-art Rust tools, there are currently many new tools or extensions to tools for Rust. There is also an ongoing project called vellvm [Zakowski et al. 2021] to provide a verified compilation toolchain for LLVM, which is also used for Rust. It should be noted, however, that there is currently no established formal specification of the semantics of Rust. This could involve extending a tool or creating a new one.

Estimating the **effort** required for this action must take into account the rapid adoption and evolution of Rust.

- We recommend a small study to determine the state of the art (estimated 0,5 person-years).
- Depending on the state of the art, the number of building blocks required to have a complete verification chain for concurrent code in Rust will vary.
- Building on existing well-established tools together with appropriate extensions, we expect that a demonstration of a verified concurrent component in Rust could be done in 3-5 years by a mid-sized group of people (estimated 21 to 24 person-years).
- If the basic building blocks for verifying Rust are still missing, the time and effort required will increase.
- Since concurrency is explicitly difficult to verify and verifying Rust is a fairly new topic, we estimate the risk of such a case study to be higher than for actions R1 and R2.

5.2.4. R4: AI-based Generation of Specifications

As described in the Scalability Model (see Section 4.3), one idea for reducing the amount of specifications needed for verification is to use recent advances in AI. When considering the use of AI, there are two possible ways to apply AI to verification. The first is to **use AI to perform proofs**, mainly by selecting the next rule/tactic to apply in (partially) manual proof systems. At this point, we do not recommend addressing this topic, as it is already being investigated in Working Group 5 of the ongoing COST action EuroProofNet[3] (see [EuroProofNet 2021] for their working plan). The second is to **use AI to generate (auxiliary) specifications** and thus reduce a large part of the manual work required for complex proofs. We have already discussed how this can be done in detail in the context of scalability, and will therefore not go into it here (see Scalability Model in Section 4.3).

While the benefits of automatic specification generation could be huge, we are skeptical that sufficient quality of AI-generated specifications can be achieved. Therefore, we consider the risk of such a project failing to be relatively high.

- We recommend starting with a medium sized project of about 2-5 years, during which this approach needs to be further evaluated.
- Several stakeholders should be involved in order to take into account the required AI competencies.
- We estimate an effort of 20 to 24 person-years.

5.2.5. R5: Composability of Properties

While the previous actions focus on proving individual properties, there is also a need to prove a set of properties for an application. Such a set could be defined either based on importance with respect to real-world bugs and attacks, or to meet a defined level of security (see community action C2 on

[3] European COST action - EuroProofNet: `https://www.cost.eu/actions/CA20111/`

Guidance for Verification in Section 5.1.2). Most of the currently available tools specialize in proving specific properties or suffer from scalability issues. Therefore, to prove complex sets of properties, a collection of multiple tools is required. For example, Amazon started with a single tool (TLA+) to verify AWS services and then gradually expanded to using more than 10 different tools to cover more properties [Peisert 2022].

The goal of this effort is to demonstrate the **alignment of tools to prove a set of properties**. This includes finding tools that can be easily aligned, i.e., using the same programming and specification language. Or, alternatively, using transformations between the tools (see research action R1 on Interoperability in Section 5.2.1). If one of the selected tools is not able to handle the whole component in terms of scalability, a **decomposition suitable for all tools** is needed. The decomposition could address the component itself as well as the properties to be verified. The difficulty is to maintain a high level of reuse between the selected tools.

Regarding the **effort** required for this action, we denote the following.

- We recommend a single moderate-sized project involving multiple stakeholders (estimated 26 to 32 person-years).
- This project could be a direct follow-up to the project we recommend for community action R2 Guidance for Verification.

5.2.6. R6: Composability for Complex Systems

While the previous research actions clearly focus on the verification of small components written in one target programming language, this action focuses on complex systems composed of components written in different languages, e.g., C and Rust at the OS level and Java at the application level.

As described in the Scalability Model (see Section 4.3), composing individually verified components into a fully verified system is a much more promising way than improving the scalability of tools to the size of complex systems. For such a composed complex system, one needs to thoroughly manage the verified properties of all components. Such **a proof management system keeps track of which properties have been verified for which component**, along with their interactions. In particular, **the proof management system should be able to derive the overall coverage of the properties** from the coverage of individual proofs. When verifying a complex system, the first step is to establish a set of basic properties. This set of properties can then be incrementally expanded over time to include more and more complex properties. This is particularly interesting with respect to attacker models, where some components or properties are addressed more directly than others (see community action C2 on Guidance for Verification in Section 5.1.2). Such a proof management system would greatly benefit from improved interoperability between tools as described in the research action R1 on Interoperability (see Section 5.2.1).

To manage verified properties of components of a complex system, one must describe the underlying software architecture of the system, in particular the layers of the system. In Figure 4.1, we presented a possible architecture of a basic IT system. This architecture is still very coarse-grained and can easily be refined to a more detailed view. Identifying the relevant layers in the software architecture allows the rigorous specification of all interfaces between components. Such a layered specification has several advantages, the most important of which are that (1) each layer is an independent component and thus can be replaced at any point without affecting the correctness of the rest of the system as long as the specification is still respected, and (2) due to the modularization, each layer can be verified independently and thus parallelization is possible. Thus, such a decomposition provides the building blocks from which a system-wide verification effort can be started.

Looking at the effort required for this action, we note the following.

- We recommend a single medium sized project, building upon the interoperability results of action R1 and last 2-5 years.
- The project should involve multiple stakeholders to account for the many tools needed to verify components written in different programming languages.
- We estimate a effort of 22 to 26 person-years.

5.2.7. R7: Robustness to Change of Complex Systems

Research action R2 on Robustness to Change (see Section 5.2.2) focuses on small components. There, we described the need to analyze the impact of code changes on existing proofs. Obviously, tracking changes and their impact becomes more difficult as the system grows. Therefore, the solutions developed in R2 to handle changes for small components need to be scaled to complex systems. In particular, complex systems are made up of components written in different programming languages. This makes it difficult to use established methods for change impact analysis, as most of them were designed for complex systems written in a single programming language.

The goal of this action is to **find scalable solutions for analyzing the impact of changes to verified components and systems**. This task can either (1) be decomposed by first identifying changes to components and then using the solutions from research action R2 to analyze the internal impact within those components, or (2) improve change impact analysis methods to handle the whole system, even if its parts are written in different languages. Either way, the proof management techniques from research action R6 on Composability for Complex Systems (see Section 5.2.6) are an essential building block of this action.

This action could be a direct continuation of research action R2 (and R6). In terms of estimating **effort**, we note the following.

- We recommend several projects to give the opportunity to explore both directions we mentioned.
- The projects should be medium to large, last 2-5 years, and involve several stakeholders.
- For all projects we estimate an total effort of 38 to 42 person-years.

5.3. Dependencies between Actions

Figure 5.2 shows the dependencies of all actions along with their duration and risk estimation. As stated in the descriptions of the different actions, there are recommendations to align some actions with other (represented as dashed lines) and strong dependencies between actions (bold lines).

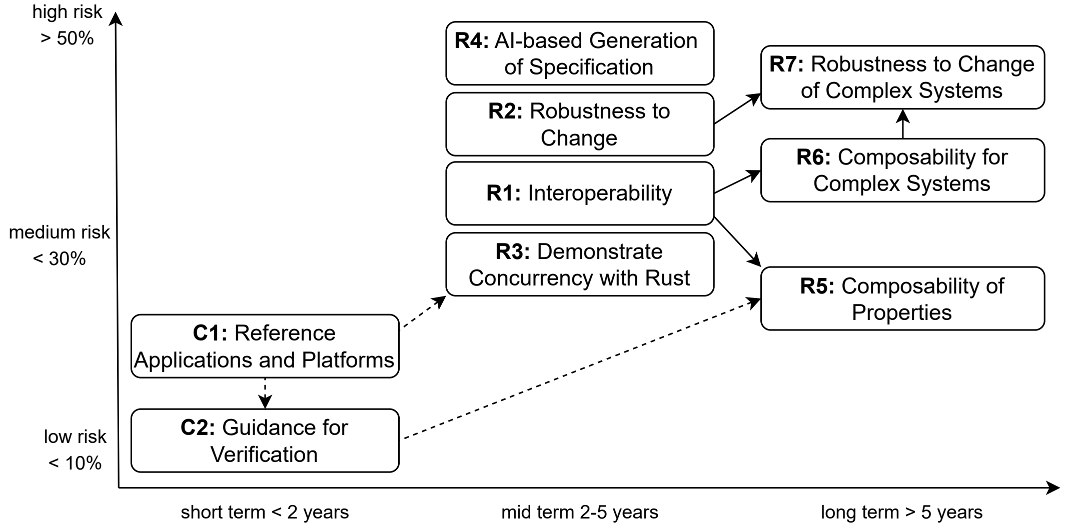
high risk
> 50%
medium risk
< 30%
low risk
< 10%
short term < 2 years
mid term 2-5 years
long term > 5 years
R4: AI-based Generation of Specification
R2: Robustness to Change
R1: Interoperability
R3: Demonstrate Concurrency with Rust
R7: Robustness to Change of Complex Systems
R6: Composability for Complex Systems
R5: Composability of Properties
C1: Reference Applications and Platforms
C2: Guidance for Verification

Figure 5.2.: Dependencies between actions (direction of arrow = *is important for*, dashed arrow = *recommendation to align*, bold arrow = *strong dependency*) and estimates on timeline and risk of each action

6. Summary

In this study, we have shown that there are many mature tools for formal verification. However, adoption by industrial users is still low. The tool communities are either driven by academic research or by research departments of large IT companies. The sharp increase in security attacks against all kinds of software components has encouraged large IT companies such as Google, Amazon, and Facebook to increase their investments in formal verification. Compared to other software assurance tools, there has not yet been a developer-driven focus on a few powerful and widely used verification tools. This is consistent with the findings of Berkeley Labs' current state analysis [Peisert 2022]. Preisert recommends increasing the usability of tools and improving the features of tools and languages to increase developer adoption. In addition, he recommends exploring the use of secure languages such as Rust and growing the still very small Rust verification community. We support these recommendations, and our recommended actions in Chapter 5 are quite similar. However, we recommend two additional important areas of research.

First, we argue that tool interoperability needs to be increased to allow developers to switch tools as needed and to support the composability of verified components into fully verified complex systems. Second, we argue that the issue of robustness to change needs to be addressed. Verifying a particular release provides strong guarantees for that code. However, in the real world, we see that new bugs are introduced into the code all the time. Therefore, the ability to re-verify each new version of the code is an important goal. Hähnle and Huisman [Hähnle and Huisman 2019] subsume interoperability and robustness to change under the term integration – the former as tool and method integration, and the latter as integration into the software production environment.

Formal verification tools are mature enough to prove many different properties of small components. However, industrial developers need guidance on how best to do this and to encourage investment in formal verification. Verification of complex systems is still a vision with an unclear roadmap. We have recommended some actions that we believe will improve the composability of smaller verified components into verified complex systems. For us, composability includes interoperability between tools as well as transparency of what has been verified in a complex system.

Glossary

Abstract Interpretation	A formal method which abstracts concrete values by abstract domains and is thus able to reason render infinite domains finite.	14
ACL2	A theorem prover for high-level descriptions of systems	23
Agda	a programming language which also allows writing proofs	24
AI	In the context of this work we consider approaches based on machine learning under the term artificial intelligence (AI).	65, 73
Alloy Analyzer	Fully automatic tool based on the alloy language	25
AproVE	termination proofs for several languages	26
Astrée	Fully automatic static analysis tool for safety properties in C	27
C	C is one of the oldest and most used high level languages. It is still predominantly used in the development of performance critical and low level system. C has to be compiled dependant on the used operating system.	20
C++	C++ was develop as an extension to C to add several features including the support for classes and thus object oriented programming. C++ has to be compiled and is operating system dependant.	20
CBMC	A bounded model checker for C	28, 58
CLI	Command Line Interface describes the way of interacting with a program/tool via a console by writing out a command which is setting options and parameters for the program/-tool.	26, 28, 30, 31, 33, 34, 40, 42–48, 50
Coq	theorem prover with the ability to be applied to programming languages	29
CPAChecker	configurable fully automatic checker for	30
Dafny	Language and automatic verifier for custom language Dafny (can be refined to C)	14, 31
Deductive Verification	A formal approach in which a software system is translated into some underlying logic and formal reasoning techniques (deduction) are then applied to show the adherence of the system to a given specification.	13
Event-B	Set theory based refinement tool	32
F*	a language designed for formal verification with corresponding tools which can be compiled to several standard languages	14, 33
Frama C	Framework for the analysis of C programs	34
GUI	Graphical User Interface describes the way of interacting with a program via a graphical interface which is the normal way for user friendly programs.	23, 25, 27, 29, 32, 36, 37, 49, 51–53

intermediate language	As the name suggests we consider an intermediate language an intermediate representation between the original programming or specification language before being then finally again translated into machine code or logic. Thus this concept is working and used for both programming and specification languages.	15
Isabelle	Proof assistant for mathematical formulas	36
Java	Java is a famously object oriented language which is still one of the most used in the world. Java code is compiled to byte-code which runs in the Java virtual machine and is thus cross platform compatible.	20
Key System	Interactive verification tool for Java with JML specifications	37
KIV	interactive verification of abstract models with refinement capabilities	38
Lean	extensible interactive Theorem prover	39
mCRL2	language and toolset for verification of concurrent systems and protocols	40
Model Checking	A formal approach in which a system is model as a (normally finite) state automaton and shown that this automaton adheres to a given specification.	14
Nagini	functional correctness of typed Python based on Viper framework	42
NuXMV	symbolic model checker for the analysis of synchronous systems	43
OpenJML	Fully automatic prover for Java specified by JML	44
PVS	interactive verification tool based on custom language for high level description of complex systems	45
Refinement	A formal approach in which a system is described a very coarse high level first and then iteratively refined down to actual code each time proving the refinement preserves the original specification.	14

Rust Rust is a rather new programming language that is similar to syntactically close to C however provides memory-safety and native support for concurrency. 20, 70, 72

SA Workbench Software Analysis Workbench is a verification tool based on dependent type theory 48

SAT Boolean satisfiability problem the NP-hard problem of determining whether a formula of Boolean variables is satisfiable. A tool able to solve answer such questions is called SAT-solver. 15

SeaHorn A fully automated analysis framework for LLVM-based languages 46

SMACK SMACK is both a modular software verification toolchain and a self-contained software verifier 47

SMT satisfiability modulo theory the NP-hard problem of determining whether a given formula is true (given a certain formal theory). A tool able to solve answer such questions is called SMT-solver. 15

Spark ADA platform with language verification tool and design method to construct secure and safe software 49

SPIN model checker explicitly for multithreaded software 50

Static Driver Verifier The Windows Static Driver Verifier is a tool for the verification of Windows drivers 41

TLA+ language for modeling concurrent and distributed systems with several tools 51

UPPAAL verification of real-time systems based on timed automata 52

VeriFast separation logic based tool for Java and C programs 53

Bibliography

Abbassi, Imed and Z Joua (2014). "An event-b driven approach for ensuring reliable and flexible service composition". In: *International Journal of Services Computing* 2.1, pp. 45–57. DOI: 10.29268/stsc.2014.2.1.4.

Abrial, Jean-Raymond, Egon Börger, and Hans Langmaack, eds. (1996). *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*. Vol. 1165. LNCS. Springer. DOI: 10.1007/BFb0027227.

Ahrendt, Wolfgang et al., eds. (2016). *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. LNCS. Springer. DOI: 10.1007/978-3-319-49812-6.

Alpern, Bowen and Fred B. Schneider (1985). "Defining Liveness". In: *Information Processing Letters* 21.4, pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0.

Appel, Andrew W. et al. (2017). "Position paper: the science of deep specification". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104, p. 20160331. DOI: 10.1098/rsta.2016.0331.

Astrauskas, Vytautas et al. (2019). "Leveraging Rust Types for Modular Specification and Verification". In: *Proc. ACM Program. Lang.* 3.OOPSLA. DOI: 10.1145/3360573.

Ball, Thomas, Vladimir Levin, and Sriram Rajamani (2011). "A decade of software model checking with SLAM". In: *Commun. ACM* 54, pp. 68–76. DOI: 10.1145/1965724.1965743.

Barbosa, Haniel et al. (2022). "cvc5: A Versatile and Industrial-Strength SMT Solver". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Springer, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

Barnett, Mike et al. (2006). "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Lecture Notes in Computer Science. Springer, pp. 364–387. DOI: 10.1007/11804192_17.

Barrett, Clark, Pascal Fontaine, and Cesare Tinelli (2016). *The Satisfiability Modulo Theories Library (SMT-LIB)*. https://www.SMT-LIB.org.

Baudin, Patrick et al. (2008). *ACSL: Ansi c specification language*. Tech. rep. CEA-LIST, France. URL: https://www.frama-c.com/download/acsl_1.2.pdf.

Baumann, Christoph et al. (2011). "Proving memory separation in a microkernel by code level verification". In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, pp. 25–32. DOI: 10.1109/ISORCW.2011.14.

Baumann, Christoph et al. (2012). "Lessons Learned From Microkernel Verification – Specification Is the New Bottleneck". In: *Proceedings Seventh Conference on Systems Software Verification, SSV 2012*. Ed. by Franck Cassez et al. Vol. 102. EPTCS, pp. 18–32. DOI: 10.4204/EPTCS.102.4.

Beckert, Bernhard and Reiner Hähnle (2014). "Reasoning and verification: State of the art and current trends". In: *IEEE Intelligent Systems* 29.1, pp. 20–29. DOI: 10.1109/MIS.2014.3.

Beckert, Bernhard and Michal Moskal (2010). “Deductive Verification of System Software in the Verisoft XT Project”. In: *Künstliche Intell.* 24.1, pp. 57–61. DOI: 10.1007/s13218-010-0005-7.

Beckert, Bernhard et al. (2017). “Proving JDK’s Dual Pivot Quicksort Correct”. In: *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers*. Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. LNCS. Springer, pp. 35–48. DOI: 10.1007/978-3-319-72308-2_3.

Beyer, Dirk (2021). “Software Verification: 10th Comparative Evaluation (SV-COMP 2021)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Springer, pp. 401–422. DOI: 10.1007/978-3-030-72013-1_24.

Bhargavan, Karthikeyan et al. (2017). “Everest: Towards a Verified, Drop-in Replacement of HTTPS”. In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. Leibniz International Proceedings in Informatics (LIPIcs), 1:1–1:12. DOI: 10.4230/LIPIcs.SNAPL.2017.1.

Brown, Fraser et al. (2020). “Towards a Verified Range Analysis for JavaScript JITs”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. ACM, pp. 135–150. DOI: 10.1145/3385412.3385968.

Butterfield, Andrew, David Sanan, and Mike Hinchey (2014). “Formalisation of a Separation Micro-Kernel for Common Criteria Certification”. In: *DASIA 2014-DAta Systems In Aerospace* 725, p. 36.

Cadar, Cristian, Daniel Dunbar, and Dawson Engler (2008). “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. USENIX Association, pp. 209–224.

Carbonneaux, Quentin et al. (2022). “Applying Formal Verification to Microkernel IPC at Meta”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2022. ACM, pp. 116–129. DOI: 10.1145/3497775.3503681.

Chaki, Sagar and Anupam Datta (2009). “ASPIER: An automated framework for verifying security protocol implementations”. In: *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, pp. 172–185. DOI: 10.1109/CSF.2009.20.

Chalin, Patrice et al. (2005). “Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2”. In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4111. LNCS. Springer, pp. 342–363. DOI: 10.1007/11804192_16.

Chong, Nathan and Bart Jacobs (2021). “Formally verifying FreeRTOS’ interprocess communication mechanism”. In: *Embedded World Exhibition & Conference 2021*. URL: https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism.

Cimatti, Alessandro et al. (2013). “The MathSAT5 SMT Solver”. In: *Proceedings of TACAS*. Ed. by Nir Piterman and Scott Smolka. Vol. 7795. LNCS. Springer. DOI: 10.1007/978-3-642-36742-7_7.

Clarke, Edmund M., Daniel Kroening, and Flavio Lerda (2004). “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. LNCS. Springer, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

Cohen, Ernie et al. (2009). “VCC: A Practical System for Verifying Concurrent C”. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Ed. by Stefan Berghofer et al. Vol. 5674. LNCS. Springer, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2.

Conchon, Sylvain et al. (2018). “Alt-Ergo 2.2”. In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. URL: https://hal.inria.fr/hal-01960203.

Costanzo, David, Zhong Shao, and Ronghui Gu (2016). “End-to-End Verification of Information-Flow Security for C and Assembly Programs”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. ACM, pp. 648–664. DOI: 10.1145/2908080.2908100.

Dam, Mads et al. (2013). “Formal verification of information flow security for a simple ARM-based separation kernel”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 223–234. DOI: 10.1145/2508859.2516702.

Dross, Claire et al. (2021). “VerifyThis 2019: A Program Verification Competition”. In: *International Journal on Software Tools for Technology Transfer* 23.6, pp. 883–893. DOI: 10.1007/s10009-021-00619-x.

Dutertre, Bruno (2014). “Yices 2.2”. In: *Computer Aided Verification*. Ed. by David Hutchison et al. Vol. 8559. LNCS. Springer, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

EuroProofNet (2021). *Memorandum of Understanding (MoU) for COST Action CA20111 – European Research Network on Formal Proofs (EuroProofNet)*. online. URL: https://e-services.cost.eu/files/domain_files/CA/Action_CA20111/mou/CA20111-e.pdf.

Filliâtre, Jean-Christophe and Andrei Paskevich (2013). “Why3 — Where Programs Meet Provers”. en. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Lecture Notes in Computer Science. Springer, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

Fisher, Kathleen, John Launchbury, and Raymond Richards (2017). “The HACMS program: using formal methods to eliminate exploitable bugs”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104. DOI: 10.1098/rsta.2015.0401.

Gu, Ronghui et al. (2019). “Building Certified Concurrent OS Kernels”. In: *Commun. ACM* 62.10, pp. 89–99. DOI: 10.1145/3356903.

Hähnle, Reiner and Marieke Huisman (2019). “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools”. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Springer, pp. 345–373. DOI: 10.1007/978-3-319-91908-9_18.

Hassan, Adel, Isam Ishaq, and Jorge Minilla (2021). “Automated verification tools for cryptographic protocols”. In: *2021 International Conference on Promising Electronic Technologies (ICPET)*, pp. 58–65. DOI: 10.1109/ICPET53277.2021.00017.

Hawblitzel, Chris et al. (2017). “IronFleet: proving safety and liveness of practical distributed systems”. In: *Communications of the ACM* 60.7, pp. 83–92. DOI: 10.1145/3068608.

Heitmeyer, Constance L et al. (2006). “Formal specification and verification of data separation in a separation kernel for an embedded system”. In: *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 346–355. DOI: 10.1145/1180405.1180448.

Hoare, C. A. R. (1969). “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259.

Huisman, Marieke et al. (2020). “The VerifyThis Collaborative Long Term Challenge”. In: *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*. Ed. by Wolfgang Ahrendt et al. Lecture Notes in Computer Science. Springer, pp. 246–260. DOI: 10.1007/978-3-030-64354-6_10.

Klein, Gerwin et al. (2009). “seL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP ’09. ACM, pp. 207–220. DOI: 10.1145/1629575.1629596.

Lattner, Chris and Vikram Adve (2004). “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

Leroy, Xavier et al. (2016). “CompCert - A Formally Verified Optimizing Compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE. URL: https://hal.inria.fr/hal-01238879.

Lorch, Jacob R. et al. (2020). “Armada: Low-Effort Verification of High-Performance Concurrent Programs”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. ACM, pp. 197–210. DOI: 10.1145/3385412.3385971.

Monti, Raúl E., Robert Rubbens, and Marieke Huisman (2022). “On Deductive Verification of an Industrial Concurrent Software Component with VerCors”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13701. LNCS. Springer, pp. 517–534. DOI: 10.1007/978-3-031-19849-6_29.

Moura, Leonardo de and Nikolaj Bjørner (2008). “Z3: An Efficient SMT Solver”. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Springer, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

Müller, Peter, Malte Schwerhoff, and Alexander J. Summers (2016). “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Lecture Notes in Computer Science. Springer, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

NSA (2022). *Software Memory Safety*. Cybersecurity Information Sheet. URL: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.

Oliveira, Daniel Bristot de, Tommaso Cucinotta, and Rômulo Silva de Oliveira (2019). “Efficient Formal Verification for the Linux Kernel”. In: *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Vol. 11724. LNCS. Springer, pp. 315–332. DOI: 10.1007/978-3-030-30446-1_17.

Pavlinovic, Zvonimir, Akash Lal, and Rahul Sharma (2016). “Inferring Annotations for Device Drivers from Verification Histories”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. ACM, pp. 450–460. DOI: 10.1145/2970276.2970305.

Peisert, Sean (2022). *The Current State of Software Assurance Tools and Techniques*. Tech. rep. UC Davi. URL: https://escholarship.org/uc/item/3435g520.

Penix, John et al. (2005). “Verifying time partitioning in the DEOS scheduling kernel”. In: *Formal Methods in System Design* 26.2, pp. 103–135. DOI: 10.1007/s10703-005-1490-4.

Przigoda, Nils et al. (2018). “Frame conditions in the automatic validation and verification of UML/OCL models: A symbolic formulation of modifies only statements”. In: *Computer Languages, Systems & Structures* 54, pp. 512–527. DOI: 10.1016/j.cl.2017.11.002.

Recoules, Frédéric et al. (2019). “Get Rid of Inline Assembly through Verification-Oriented Lifting”. In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, pp. 577–589. DOI: 10.1109/ASE.2019.00060.

Roșu, Grigore and Traian Florin Serbănută (2010). “An Overview of the K Semantic Framework”. In: *The Journal of Logic and Algebraic Programming*. Membrane Computing and Programming 79.6, pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

Sommerville, Ian (2011). *Software Engineering*. 9th ed. Addison-Wesley. ISBN: 978-0-13-703515-1.

Vanfleet, W Mark et al. (2005). “MILS: Architecture for high-assurance embedded computing”. In: *CrossTalk* 18.8, pp. 12–16.

Zakowski, Yannick et al. (2021). “Modular, compositional, and executable formal semantics for LLVM IR”. In: *Proc. ACM Program. Lang.* 5.ICFP, pp. 1–30. DOI: 10.1145/3473572.

Zhao, Yongwang, Zhibin Yang, and Dianfu Ma (2017). “A survey on formal specification and verification of separation kernels”. In: *Frontiers of Computer Science* 11.4, pp. 585–607. DOI: 10.1007/s11704-016-4226-2.

A. Publications from Snowballing

This appendix lists the 11 publications retrieved by snowballing (see Section 3.3).

References

Beckert, Bernhard and Reiner Hähnle (2014). "Reasoning and verification: State of the art and current trends". In: *IEEE Intelligent Systems* 29.1, pp. 20–29. DOI: 10.1109/MIS.2014.3.

Beyer, Dirk and Matthias Dangl (2020). "Software Verification with PDR: An Implementation of the State of the Art". In: *Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2020*. Ed. by Armin Biere and David Parker. Vol. 12078. LNCS. Springer, pp. 3–21. DOI: 10.1007/978-3-030-45190-5_1.

Beyer, Dirk, Matthias Dangl, and Philipp Wendler (2018). "A Unifying View on SMT-Based Software Verification". In: *Journal of Automated Reasoning* 60.3, pp. 299–335. DOI: 10.1007/s10817-017-9432-6.

Beyer, Dirk, Sudeep Kanav, and Cedric Richter (2022). "Construction of Verifier Combinations Based on Off-the-Shelf Verifiers". In: *Fundamental Approaches to Software Engineering, FASE 2022*. Ed. by Einar Broch Johnsen and Manuel Wimmer. Vol. 13241. LNCS. Springer, pp. 49–70. DOI: 10.1007/978-3-030-99429-7_3.

Beyer, Dirk, Martin Spiessl, and Sven Umbricht (2022). "Cooperation Between Automatic and Interactive Software Verifiers". In: *Software Engineering and Formal Methods, SEFM 2022*. Ed. by Bernd-Holger Schlingloff and Ming Chai. Vol. 13550. LNCS. Springer, pp. 111–128. DOI: 10.1007/978-3-031-17108-6_7.

Damiani, Ferruccio, Reiner Hähnle, and Michael Lienhardt (2017). "Abstraction Refinement for the Analysis of Software Product Lines". In: *Tests and Proofs, TAP@STAF 2017*. Ed. by Sebastian Gabmeyer and Einar Broch Johnsen. Vol. 10375. LNCS. Springer, pp. 3–20. DOI: 10.1007/978-3-319-61467-0_1.

de Gouw, Stijn et al. (2019). "Verifying OpenJDK's Sort Method for Generic Collections". In: *Journal of Automated Reasoning* 62.1, pp. 93–126. DOI: 10.1007/s10817-017-9426-4.

Do, Quoc Huy, Richard Bubel, and Reiner Hähnle (2017). "Automatic Detection and Demonstrator Generation for Information Flow Leaks in Object-Oriented Programs". In: *Computers & Security* 67, pp. 335–349. DOI: 10.1016/j.cose.2016.12.002.

Gouw, Stijn de et al. (2015). "OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case". In: *Computer Aided Verification, CAV 2015*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. LNCS. Springer, pp. 273–289. DOI: 10.1007/978-3-319-21690-4_16.

Lathouwers, Sophie and Marieke Huisman (2022). "Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers". In: *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*. FormaliSE '22. ACM, pp. 69–79. DOI: 10.1145/3524482.3527652.

Scheurer, Dominic, Reiner Hähnle, and Richard Bubel (2016). "A General Lattice Model for Merging Symbolic Execution Branches". In: *Formal Methods and Software Engineering, ICFEM 2016*. Ed. by Kazuhiro Ogata, Mark Lawford, and Shaoying Liu. Vol. 10009. LNCS, pp. 57–73. DOI: `10.1007/978-3-319-47846-3\_5`.

B. Publications from relevant Conferences

This appendix lists the 11 publications retrieved by screening the relevant conference: FASE, TACAS, FM, CAV, IJCAR and NFM (see Section 3.3).

References

Abate, Alessandro et al. (2018). "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. LNCS. Springer, pp. 270–288. DOI: 10.1007/978-3-319-96145-3_15.

Abdulla, Parosh Aziz et al. (2015). "Stateless Model Checking for TSO and PSO". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. LNCS. Springer, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28.

Alur, Rajeev, Arjun Radhakrishna, and Abhishek Udupa (2017). "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Axel Legay and Tiziana Margaria. LNCS. Springer, pp. 319–336. DOI: 10.1007/978-3-662-54577-5_18.

Barbosa, Haniel et al. (2022). "Cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. LNCS. Springer, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

Beyer, Dirk (2015). "Software Verification and Verifiable Witnesses". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. LNCS. Springer, pp. 401–416. DOI: 10.1007/978-3-662-46681-0_31.

– (2016). "Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Marsha Chechik and Jean-François Raskin. LNCS. Springer, pp. 887–904. DOI: 10.1007/978-3-662-49674-9_55.

– (2017). "Software Verification with Validation of Results". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Axel Legay and Tiziana Margaria. LNCS. Springer, pp. 331–349. DOI: 10.1007/978-3-662-54580-5_20.

– (2019). "Automatic Verification of C and Java Programs: SV-COMP 2019". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer et al. LNCS. Springer, pp. 133–155. DOI: 10.1007/978-3-030-17502-3_9.

– (2020). "Advances in Automatic Software Verification: SV-COMP 2020". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Armin Biere and David Parker. LNCS. Springer, pp. 347–367. DOI: 10.1007/978-3-030-45237-7_21.

– (2021). "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. LNCS. Springer, pp. 401–422. DOI: 10.1007/978-3-030-72013-1_24.

Bjørner, Nikolaj, Anh-Dung Phan, and Lars Fleckenstein (2015). “νZ - An Optimizing SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. LNCS. Springer, pp. 194–199. DOI: 10.1007/978-3-662-46681-0_14.

Bunte, Olav et al. (2019). “The mCRL2 Toolset for Analysing Concurrent Systems”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. LNCS. Springer, pp. 21–39. DOI: 10.1007/978-3-030-17465-1_2.

Calcagno, Cristiano et al. (2015). “Moving Fast with Software Verification”. In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. LNCS. Springer, pp. 3–11. DOI: 10.1007/978-3-319-17524-9_1.

Cavada, Roberto et al. (2014). “The nuXmv Symbolic Model Checker”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 334–342. DOI: 10.1007/978-3-319-08867-9_22.

Chudnov, Andrey et al. (2018). “Continuous Formal Verification of Amazon S2n”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. LNCS. Springer, pp. 430–446. DOI: 10.1007/978-3-319-96142-2_26.

Cook, Byron (2018). “Formal Reasoning About the Security of Amazon Web Services”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. LNCS. Springer, pp. 38–47. DOI: 10.1007/978-3-319-96145-3_3.

Cordeiro, Lucas et al. (2018). “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. LNCS. Springer, pp. 183–190. DOI: 10.1007/978-3-319-96145-3_10.

Dutertre, Bruno (2014). “Yices 2.2”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

Garg, Pranav et al. (2014). “ICE: A Robust Framework for Learning Invariants”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 69–87. DOI: 10.1007/978-3-319-08867-9_5.

Gibson-Robinson, Thomas et al. (2014). “FDR3 - A modern refinement checker for CSP”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. LNCS. Springer, pp. 187–201. DOI: 10.1007/978-3-642-54862-8_13.

Giesl, Jürgen et al. (2014). “Proving Termination of Programs Automatically with AProVE”. In: *Automated Reasoning*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. LNCS. Springer, pp. 184–191. DOI: 10.1007/978-3-319-08587-6_13.

Gurfinkel, Arie et al. (2015). “The SeaHorn Verification Framework”. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. LNCS. Springer, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

Komuravelli, Anvesh, Arie Gurfinkel, and Sagar Chaki (2014). “SMT-Based Model Checking for Recursive Programs”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 17–34. DOI: 10.1007/978-3-319-08867-9_2.

Kroening, Daniel and Michael Tautschnig (2014). “CBMC – C Bounded Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. LNCS. Springer, pp. 389–391. DOI: 10.1007/978-3-642-54862-8_26.

Kumar, Rajesh et al. (2018). “Effective Analysis of Attack Trees: A Model-Driven Approach”. In: *Fundamental Approaches to Software Engineering*. Ed. by Alessandra Russo and Andy Schürr. LNCS. Springer, pp. 56–73. DOI: 10.1007/978-3-319-89363-1_4.

Liang, Tianyi et al. (2014). “A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 646–662. DOI: 10.1007/978-3-319-08867-9_43.

Menezes, Rafael et al. (2018). “Map2Check Using LLVM and KLEE”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. LNCS. Springer, pp. 437–441. DOI: 10.1007/978-3-319-89963-3_28.

Niemetz, Aina et al. (2018). “Btor2 , BtorMC and Boolector 3.0”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. LNCS. Springer, pp. 587–595. DOI: 10.1007/978-3-319-96145-3_32.

Rakamarić, Zvonimir and Michael Emmi (2014). “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7.

Reger, Giles, Helena Cuenca Cruz, and David Rydeheard (2015). “MarQ: Monitoring at Runtime with QEA”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. LNCS. Springer, pp. 596–610. DOI: 10.1007/978-3-662-46681-0_55.

Reinbacher, Thomas, Kristin Yvonne Rozier, and Johann Schumann (2014). “Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. LNCS. Springer, pp. 357–372. DOI: 10.1007/978-3-642-54862-8_24.

Reynolds, Andrew et al. (2015). “Counterexample-Guided Quantifier Instantiation for Synthesis in SMT”. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. LNCS. Springer, pp. 198–216. DOI: 10.1007/978-3-319-21668-3_12.

Schubert, Philipp Dominik, Ben Hermann, and Eric Bodden (2019). “PhASAR: An Inter-procedural Static Analysis Framework for C/C++”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. LNCS. Springer, pp. 393–410. DOI: 10.1007/978-3-030-17465-1_22.

Sharma, Rahul and Alex Aiken (2014). “From Invariant Checking to Invariant Inference Using Randomized Search”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 88–105. DOI: 10.1007/978-3-319-08867-9_6.

Sinn, Moritz, Florian Zuleger, and Helmut Veith (2014). “A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 745–761. DOI: 10.1007/978-3-319-08867-9_50.

Solovyev, Alexey et al. (2015). “Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions”. In: *FM 2015: Formal Methods*. Ed. by Nikolaj Bjørner and Frank de Boer. LNCS. Springer, pp. 532–550. DOI: 10.1007/978-3-319-19249-9_33.

Stump, Aaron, Geoff Sutcliffe, and Cesare Tinelli (2014). “StarExec: A Cross-Community Infrastructure for Logic Solving”. In: *Automated Reasoning*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Vol. 8562. LNCS. Springer, pp. 367–373. DOI: 10.1007/978-3-319-08587-6_28.

Voronkov, Andrei (2014). “AVATAR: The Architecture for First-Order Theorem Provers”. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. LNCS. Springer, pp. 696–710. DOI: 10.1007/978-3-319-08867-9_46.

Wang, Xinyu et al. (2018). “Learning Abstractions for Program Synthesis”. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. LNCS. Springer, pp. 407–426. DOI: 10.1007/978-3-319-96145-3_22.

cyberagentur