FOI
Swedish Defence Research Agency
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se

Overview of formal methods in software engineering

IOANA RODHE, MARTIN KARRESAND

FOI-R--4156--SE
ISSN 1650-1942
December 2015

Ioana Rodhe, Martin Karresand

Overview of formal methods in software engineering

Bild/cover: Martin Karresand

Titel	Översikt över formella metoder i programvaruutveckling
Title	Overview of formal methods in software engineering
Report no	FOI-R--4156--SE
Month	December
Year	2015
Pages	50
ISSN	ISSN-1650-1942
Customer	The Swedish Armed Forces
Project no	E36077
Approved by	Christian Jönsson
Division	Information- and Aeronautical Systems
FOI Research area	
Armed Forces R&T area	Command and control

FOI Swedish Defence Research Agency

Abstract

The most common way to check for faults and bugs in IT systems is to use testing, simulations and human reviews. Nevertheless, these techniques are not exhaustive and cannot guarantee the absence of defects. A more promising method is to use formal methods during the development process of the IT system, for example by writing a formal specification of the system on which different properties can be proved and then mathematically proving that the implementation of the system abides by the specification.

In this report we aim to give a brief introduction to the field of formal methods together with a classification of the different existing formal methods. We also guide the reader through some of the existing literature in the field and we present some systems where formal methods are used to prove security properties. An overview of a functional correctness proof for a security-critical micro-kernel is also provided.

From the literature we have surveyed we conclude that using formal methods can be cost-effective in the long run and it can reduce the number of defects to close to zero. Nevertheless, the methods are not yet widely used and, therefore, there are a limited number of tools that can be used.

Keywords

formal methods, functional correctness, IT system, software development

Sammanfattning

Det vanligaste sättet att leta efter fel och buggar i IT-system är att använda testning, simulering och manuell kodgranskning. Dessa tekniker är dock inte uttömmande och kan inte garantera frånvaro av defekter. En mer lovande metod är att använda formella metoder under IT-systemets utvecklingsprocess, till exempel genom att skriva en formell specifikation för systemet genom vilken olika egenskaper kan bevisas och sedan matematiskt bevisa att implementationen följer specifikationen.

I denna rapport avser vi att ge en kortfattat introduktion till forskningen kring formella metoder tillsammans med en klassificering av de olika existerande formella metoderna. Vi guidar också läsaren genom ett urval av forskningslittarturen inom området och vi presenterar några system där formella metoder använts för att bevisa säkerhetsegenskaper. Även en överblick av ett funktionellt riktighetsbevis för en säkerhetskritisk mikrokärna ges.

Utifrån den litteraturstudie vi gjort kan vi konstatera att användning av formella metoder verkar vara kostnadseffektivt i det långa loppet och att det reducerar antalet defekter till nära noll. Oavsett det har metoderna ännu inte fått någon större spridning och det finns därför ett begränsat antal verktyg att tillgå.

Nyckelord

formella metoder, funktionella riktighetsbevis, IT-system, mjukvaruutveckling

Contents

1 Introduction

Ever since the first systems appeared, a question has been why systems are not fault free. The simple answer is that even the most rudimentary system is affected by parameters we cannot grasp and stimuli we cannot control. If we limit the scope of the question to IT systems, the development process itself is full of opportunities to introduce faults and bugs into the code.

In this chapter we introduce formal methods, what they are and how they can reduce the number of faults and bugs, and we present a background describing the development process, its pitfalls and how formal methods come into the picture. We further define the scope of the report and give an overview of the remaining chapters.

1.1 Formal methods in software engineering

In computer science, formal methods are "mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems" [43]. Mathematically rigorous means that the specification consists of well-formed statements using mathematical logic and that a formal verification consists of rigorous deductions in that logic. The strength of formal methods is that they allow for a complete verification of the entire state space of the system and that the properties that can be proved to hold in the system will hold for all possible inputs. When formal methods cannot be used through the entire development process (due to the complexity of the system, lack of tool or other reasons), they can still successfully be used on parts of the system, for example for the requirements and high-level design or only on the most safety or security critical components.

The diversity of available formal methods is a result of the different modelling methods and proof approaches needed by different application domains. Also, different development phases of a system might require different tools and techniques.

Although many developed formal methods are the result of research efforts in universities, more and more tools and techniques are available outside the academic community.

Several of the standards used for system development require formal methods at the highest levels of accreditation. Some examples are the Common Criteria standard[1] and the DO-178C standard *Software Considerations in Airborne Systems and Equipment Certification*[2] for software for airborne systems in commercial aircraft.

[1]`http://www.commoncriteriaportal.org/`
[2]`https://en.wikipedia.org/wiki/DO-178C`

1.2 Background

When a new system is to be implemented, the first step is to write a requirement specification (usually in natural language). The specification should correctly describe the system's desired behaviour and it should be complete and unambiguous, which can be hard to achieve.

The specification is then transformed into code by a programmer, who has to understand the specification correctly and handle any ambiguities. Also, the programmer's way of coding and solving technical challenges can introduce faults in the code. Then there is the sheer size of the system; nowadays systems are so big that it can be hard to keep track of all the parts to make sure that they correctly follow the specification. Furthermore, there is often a team of programmers working together, which also is a source of faults since they will all have their own interpretations of the specification and of the information shared during the development process.

During and after the coding of the system, the system's functionality is usually tested to make sure that the resulting program satisfies the requirements and that no errors or bugs are present. Testing big and complex systems can be very time consuming and, due to the size of the system and the amount of code, an exhaustive testing is not practically feasible.[3] Nevertheless when the system is safety and security critical, correct functionality has to be guaranteed, which requires either exhaustive testing or a way of proving that the code correctly implements the specification [24].

The concept of *formal method*s introduces tools to mathematically describe a system (or parts of a system) in a specification and to prove that the resulting program meets the requirements described in the specification. A formal specification is precise and there is no risk for misinterpretations. Also, if there is a proof that the implementation abides by the specification, then one can be sure that the programmers have implemented what is described in the specification. In practice, one cannot completely guarantee that the resulting implementation is fault free, since the formal method used can have defects, or there might be some error in the proof. Nevertheless, increased use of formal methods and tools will result in better and more reliable methods and tools. To summarise: by using formal methods in the system development, errors can be found earlier and some classes of errors can be nearly eliminated [30].

A limitation of formal methods is that they only can be used to prove a system's correctness with respect to a specification. Therefore, just because a program has been mathematically proved to abide to the specification, there is no guarantee that the specification in itself is correct and fault free. Nevertheless, properties can be proved on the specification to strengthen the belief that the specification correctly represents the desired functionality.

[3]A simple program that adds two six digit numbers give 10^{12} execution paths to test. Testing 10^5 paths per second we still need almost 2778 hours, more than 115 days, to do an exhaustive test.

1.3 Scope and method

In this report we aim to give a brief introduction to the field of formal methods together with a classification of the different existing formal methods. We also guide the reader through some of the existing literature in the field.

In order to gather as much relevant background information as possible in the field of formal methods, we contacted several researchers from different universities in Sweden and asked them some questions related to the field. The questions can be found in Section 10. We furthermore gathered information on which research groups there are in Sweden and on some of their members.

1.4 Outline

Chapter 1: The current section provides an introduction to the research problem and the scope of the report.

Chapter 2: Presents what we together with some Swedish researchers within the area regard as relevant literature regarding formal methods. The selection is not meant to be exhaustive, but a start for someone wanting an overview of the area.

Chapter 3: Contains a classification of the different formal methods currently available. The classification is based on [3].

Chapter 4: Provides a review of how to choose the best formal method for a project. To aid in that a classification framework for systems is presented. The classification is based on [51].

Chapter 5: Contains a brief presentation of non-academic sources of information on formal methods.

Chapter 6: Discusses limitations of the use of formal methods.

Chapter 7: Presents a few examples of using formal methods. The focus lies on what has been proved and what assumptions that have been made.

Chapter 8: Gives an overview of how a functional correctness proof can be carried out.

Chapter 9: A listing of some of the research groups and researchers active in the field of formal methods in Sweden.

Chapter 10: Presents the questions used in the interviews with the different Swedish researchers and groups listed in Section 9.

Chapter 11: Contains the conclusions of the study.

2 Relevant literature

There are thousands of conference and journal papers on the subject of formal methods, most of them focusing on a particular method and very few overviews over the area. In order to find relevant literature, a search for literature giving an overview of the area of formal methods and their use was performed. The keywords used were "formal methods overview" and "formal method tutorial". Furthermore, we asked researchers in the area about relevant overviews.

In [10, 11, 12] the fundamentals of software engineering based on a formal methodology is presented by Dines Bjørner. He writes in the preface that, unlike other system engineering books where formal methods are tucked away in a separate chapter, this book has the formal methods embedded in the text. The reason for that is the fact that formal methods apply in all stages of software engineering.

Dines Bjørner, together with Klaus Havelund, wrote a conference article on the history of formal methods [13]. It is a short text, but contains a comprehensive account of the history, current status and future development of formal methods. They also list the core research groups, conferences, etcetera in a section called "A Syntactic Status Review". In the second half of the paper, they also present their personal views on eight obstacles to the success of formal methods. The authors believe that the academic and industry obstacles regarding formal methods can be overcome. The lack of scalable and practical support tools is, according to the authors, a main reason for the slow adoption in the industry. As for the future of formal methods, the authors believe that we are moving towards "a point of singularity where the specification and programming will be done within the same language and verification tooling framework".

A survey of the actual use of formal methods in industry is presented in [60]. Data was collected from 62 industrial projects by using a structured questionnaire. The projects are from different application areas. The results of the survey show that the effect on time (that is, time to do the work) was on average beneficial, three times as many reported a reduction in time rather than an increase. In about half of the projects, it was considered hard to judge the effect on time taken. The same was noticed for the cost, five times as many projects reported reduced costs rather that increased and in about half the cases it was hard to judge the effect. Furthermore, in most of the cases, 92%, it is believed that formal methods have helped increase the quality. From this survey and past surveys, the authors can observe that formal methods and verification has successfully been applied to problems of industrial scale and significance. Nevertheless, the authors identify remaining challenges for the broader adoption of formal methods in the industry. Some of these challenges are tool support, increased automation and convincing evidence that formal methods can be cost-effective. The survey is accompanied by an account of a series of industrial projects where formal methods were used with emphasis on developing verified systems cost-effectively. Some of the projects are briefly presented in Chapter 7.

A survey of automated techniques for formal software verification is presented in [22]. The three techniques presented are Abstract Static Analysis, Model Checking, and Bounded Model Checking. The authors provide a short tutorial of each method with discussions of the differences between them. Also, tools implementing these techniques are presented and their maturity, strengths and weaknesses are discussed.

In [49], the European project DEPLOY is presented. The project lasted for four years and aimed at introducing the Event-B formal method [2] to a variety of industrial organisations in different application domains. 15 partners participated and reported the lessons learned from the project. The results were used to improve the process and the formal method. The book contains an "honest and insightful" account of the participants' experiences. The DEPLOY project also has a useful website `http://www.fm4industry.org` with resources on success stories and FAQ on formal methods in industry.

In [36], the fundamentals of the use of formal methods in operating system verification is presented. The first half of the article is a general overview of formal methods and their use in software engineering. The second half of the article covers different formal methods used in operating system verification. The authors look at OS verification projects between 1973 to 2008, and therefore the paper does not include the recently verified micro-kernel seL4. We present seL4 as one of our examples in chapter 7 and further give an overview of its functional correctness in Chapter 8.

In [28], a comprehensive survey of the area is presented where both safe and secure systems are considered. The survey aims to give an overview of the state-of-the-art in the field of formal methods, including tools, languages and methodologies. The authors also aim to give an evaluation of strengths and limitations of formal methods and to set guidelines for the deployment and use of formal methods in the industry. The authors believe that formal methods have been advertised too early in the industry, before the languages and tools were mature enough to be used in an industry scenario. They also identify recent advances in the area that make formal methods more available for the industry, such as: the foundational principles are more widely taught and understood; more expressive and user-friendly languages are available; new tools are available; and the diversity of problems that formal methods can tackle increases continuously.

There are many books on formal methods, both about the theory and on practical applications. Regarding the application of formal methods in the industry, the following books have been recommended by the researchers we have talked to:

1. Stefania Gnesi and Tiziana Margaria (ed.), Formal Methods for Industrial

Critical Systems: A Survey of Applications [54]

2. Jean-Louis Boulanger (ed.), Industrial Use of Formal Methods [14]

We finish the literature list with a webpage where many formal verification reviews and surveys are listed: `http://www.cerc.utexas.edu/~jay/fv_surveys/`.

3 Classification of formal methods

Almeida et al. [3] present several classifications for formal methods which we use in this chapter. At a general level, formal methods are used in two aspects of software development:

- To enforce the desired behaviour in the specification of the system. A specification is a model of the system that describes its behaviour and formal methods are used for model validation.
- To verify that an implementation has the same behaviour as the specification, or to obtain an implementation that has the same behaviour as the specification. Here one talks about the formal relationship between implementation and specification.

Another general classification of formal methods can be made by considering how the modelled system is described:

- as a transition system with states, transitions and state transformations, or
- some program logic with pre- and post-conditions as well as axioms and inference rules.

A third way to classify formal methods, which we also elaborate on in the remainder of this chapter, is:

1. formal methods used to *specify and analyse* the specification,
2. formal methods used to *specify and prove* properties of the specification (formal verification),
3. formal methods used to *specify and derive* an implementation from the specification, and
4. formal methods used to *specify and transform* the specification, transformations which either hide details or enrich the specification with extra details.

3.1 Specify and analyse

A specification describes the data that the system manipulates and what operations transform the data. The system can be described in two ways. One way is to consider an internal state and to describe the operations that modify this state. These kinds of specifications of specifications are called *model-based* or *state-based specifications* and build a unique model of the system. The second way to describe systems is by focusing on the data that is being manipulated and how the data evolves. These kinds of specifications of specifications are called *algebraic specifications* and, in this case, there can be many models that provide the required functionality based on the manipulated data. To reduce the number of models, properties of the specification are provided and, commonly, only a few of the models satisfy the required properties.

Model-based specifications are described in languages where the internal state is the central point. Such languages are abstract state machines, set and category theories, automata-based modelling and modelling languages for real-time systems.

An *abstract state machine* describes the system in terms of states and state transformations. Some examples of such languages are the B Method together with the B specification language [2].

When the *set* or *category theory* is used, states are described in terms of mathematical structures such as sets, relations or functions. Transitions are expressed as invariants[1], together with pre- and post-conditions. Examples of set theory formal methods are Z [47] and VDM [34] and examples of category theory methods are Specware [35] and Charity [18].

Automata are a different class of transition systems, which are particularly adequate to describe concurrent, reactive and communicating systems. Automata can be extended and, as such, used to specify real-time systems. The most common such automata are timed automata [4], but extensions for modelling of other characteristics of systems such as temperature, inclination or altitude can be found. SCADE [1] is a complete modelling environment, based on Lustre [15], that can be used to model synchronous concurrent real-time systems.

Algebraic specifications concentrate on the specification of data and the input-output behaviour of functions that manipulate data. They are particularly well suited for describing interface specification (i.e., when a large system is divided into several subsystems with well-defined interfaces between them). Nevertheless, when object operation depends on object state, that is when the result of applying an operation is dependent of the results of previous operations, then algebraic specifications become cumbersome [52]. Sommerville further on claims that a combination of model-based and algebraic specifications defines the overall behaviour of the system [52].

Furthermore, declarative languages, such as logic-based languages, functional languages and rewriting languages, can be used when writing the specification. Many functional languages are based on the *λ-calculus* [6] and the notion of function is a central point of the specification. There also are available proof assistants like Isabelle [46] and HOL [29] and languages like Haskell [56] based on variants or extensions of the λ-calculus.

[1]An invariant is a property P, with respect to a program, that holds initially and that is preserved by all the atomic properties of the program.

3.2 Specify and prove

Another aspect of formal methods is to prove properties about the specification (or about the implementation), which is also called formal verification. In order to do formal verification, some logical system has to be used, such as first-order logic or temporal logic.

The formal frameworks for verification are divided by Rushby [50, p. 26] in three different levels:

First level Demonstrations are carried out by hand with no computer support. Natural languages can be used.

Second level Demonstrations are carried out by hand but a rigorous formulation of demonstrations is required.

Third level Computer-based tools with support for carrying out demonstrations and proofs.

Different approaches to formal verification exist, such as *proof tools*, *model checking* and *program annotations*. When using *proof tools* one can choose between logical expressiveness (that gives the possibility to study complex properties) and the simplicity of the logical formalism (that gives the possibility to automatically generate the proof). On one hand, automated theorem provers can be used, which focus on simplicity rather than on expressiveness. The construction of proofs is automatic since they rely on the decidability of the underlying theory. Example of automatic theorem provers are Satisfiability Modulo Theory solvers such as Yices [23], CVC3 [8] and Z3 [21].

On the other hand, if one needs higher expressiveness, *proof assistants* can be used where undecidable logic such as higher-order logic is used.

Model checking [5] is a technique for verification of finite-state systems broadly used in the industry. The main idea is to express the system with help of temporal logic and then traverse the model entirely and validate the desired properties. If a property is not valid then a counterexample is shown. The drawback with model checking is state space explosion where the transition graph grows exponentially with the size of the system. There are different techniques available that try to solve the state space explosion problem, for example one can use a higher level of abstraction.

Bounded model checking (BMC) [9] is a form of model checking that performs a depth-bounded exploration of the state space, i.e., only states reachable within a bounded number of steps are explored. Due to the bounded exploration, faults that require a longer path are missed. BMC is mostly used for semiconductor verification although it can also be applied to software.

Program annotations are used to check behavioural properties of programs (source code in particular). An annotation is a logical formula of the specification, placed together with the program whose behaviour is to be verified. The formula includes the pre and post conditions for that piece of program, for example a method. One specific form of annotations, that is widely spread, is *contracts* and the *runtime verification* of contracts. Almost all widespread programming languages have such a contract layer. For example the JML annotation language[2] for Java and the ACSL annotation language for C.

3.3 Specify and derive

When a specification with desired properties is available, the next step is to obtain an implementation with a behaviour that matches the specification. There are two solution categories for this step. One is to have a specification that can directly be executed, for example by using logic-based languages like Prolog or functional languages like Scheme and OCalm. Another one is to derive an implementation from the specification. In the first case one already has an implementation that satisfies the desired properties. In the second case one has to prove that the derivation is correct. There are different approaches for how to prove correctness of a derivation. One approach is to focus on the derivation process and prove that it is a correct derivation process. Another approach is to make the derivation process generate a set of proof obligations and prove that they are correct. Therefore either the derivation mechanism or the individual derivation will be subject to formal verification.

A very popular technique, that is used in the industry, is *refinement* where a program is synthesized step by step from the specification, such that each step increases the degree of precision with respect to the implementation. Steps are implementation choices, such as, which algorithm to use or which data type to choose for a specific variable. Each step is then proved correct. Z, VDM and B use the refinement technique.

3.4 Specify and transform

It is often desirable to construct transformations of a specification in order to either abstract details (by approximating them) or to enrich the specification with extra details. The theory of *abstract interpretation*[3] [20] is the main foundation for such transformations.

As in engineering, where large problems often are divided into smaller ones, transformations allow for modularity in formal verification. By decomposing the behaviours of the global model into different views the work is simplified. In some situations, studying individual aspects can even give equivalent results to studying the global model.

A problem with such transformations is that the abstractions in their general form are easily undecidable. Moreover, transformations are deeply tied to the modelling language used and the properties to be proved. As a consequence, there are a limited number of tools that can be used for such transformations.

[2]`http://www.jmlspecs.org`

[3]A framework for relating abstract analyses to program execution.

Approximations are often done in an ad-hoc fashion and it is hard to make them sound and appropriate.

However, there exists one example of an early ad-hoc model transformation tool set called *JaKarTa*, which is used for reasoning about JavaCard specifications. JaKarTa provides a rule-based language and transformation specifications based on the effect on the analysed model's data structures. An example of such transformations is when the focus is on the typing policy of the operational semantics of a virtual machine. By focusing on the typing policy, the actual values are of no interest. The transformation to be performed on the data manipulated by the virtual machine can automatically be passed by JaKarTa on to the operations of the machine and then the soundness of the transformation can be ensured by producing proof obligations in the Coq proof assistant [4].

[4]https://coq.inria.fr/

4 How to choose a formal method

The Formal Methods Europe association has on its website a brief document [25] about what steps to take when beginning with formal methods.

Furthermore, the researchers we have spoken with suggested to find qualified people with a wide background in formal methods, so that they can choose the right formal methods for a specific project.

Which formal method that is most suitable for a specific project is dependent on the kind of system and the kind of properties to be proved. In this chapter we provide classifications over systems and system properties that can help when choosing a formal method.

4.1 Systems classification

According to Klaus Schneider [51], systems can be classified as follows, based on their architecture:

- Asynchronous or synchronous hardware
- Analogue or digital hardware
- Mono- or multi-processor systems
- Imperative/functional/logic-based/object-oriented software
- Multi-threaded or sequential software
- Conventional or real-time operating systems
- Embedded systems or local systems or distributed systems

The author discusses some of the distinguishing characteristics of these systems and what formal methods are most appropriate for them. For example, multi-threaded software might require the implementation of complex control tasks and therefore requires a high-level description language. In sequential systems one could use logic-based languages, such as Prolog, and functional languages, such as ML or Haskell.

Further, also based on [51], systems can be classified based on the type of interaction:

Transformational systems
: Systems that read some input data and produce output. As the output is produced at termination, these systems should always terminate. A compiler is an example of such a system.

Interactive systems
: Systems that continuously run and interact with the environment. When the environment gives an action, the system replies with a reaction. The environment has to wait until the system is ready for new actions.

Reactive systems
: Similar to interactive systems, only that the environment can freely decide when to start new actions. Therefore the system has to react to a given action before the next action comes. This kind of system falls under the real-time systems category.

Considering this last classification, it is reactive systems that are the most challenging to implement.

4.2 System properties

Many formalisms can be classified by which system properties they can express. A taxonomy of properties is given next, based on [51]:

Safety properties "something bad will never happen".

Liveness properties "something good will eventually happen".

Fairness properties "some property will infinitely often hold".

Persistence properties "stabilisation of certain properties". After some point in time, a given property will always hold.

4.3 Application domains

The authors of [28] consider that formal methods can be categorised based on the application domain and based on the environmental assumptions. These categories can be used when choosing an appropriate formal method. The following application domains are considered:

Protocol design and engineering Formal description, and verification, have been used when designing new protocols. Some protocol specific languages are ESTELLE and SDL, but they have been replaced by more general languages.

Software design and engineering In this area theoretical foundations and analysis tools are provided by formal methods.

Hardware design and engineering Formal verification tools are widely used which are aimed at detecting design mistakes.

Furthermore, different environments are considered, based on how well understood and predictable they are, with appropriate formal methods suggested for each environment as follows:

Nominal environment which is well-understood and predictable. In such an environment, formal methods focus on correctness and performance issues.

Faulty environment which is mostly understood and predictable, but abnormal events can affect the system. In such environments, the formal methods focus on dependability and performance issues.

Hostile environment which is neither totally understood nor predictable and its behaviour cannot be trusted. In such environments, the formal methods focus on security issues. Both general formal methods and dedicated formal methods have been successfully applied to security issues. For example model checking has been used to find unknown attacks in security protocols [40, 41]. Examples of dedicated formal methods are security-oriented formal notations and software tools for automated analysis of security protocols. Furthermore, in order to ensure security one has to ensure both correctness and dependability. At the same time, formal methods cannot address security issues such as computer hacking, tampering or social engineering.

4.4 Where to use formal methods

In this section we give a brief overview of how formal methods have evolved with time and in what kind of systems they can be used, based on [28].

To begin with, formal methods were developed for sequential programs focusing on formally proving program correctness. Then concurrent systems were the focus and they were formalised with help of *Petri nets*[1]. Here the focus was on message-passing and shared memory. Petri nets were further used for communication systems and distributed and mobile systems. Later formal methods were used in time-critical systems, both for software and for hardware. Both discreet time (*reactive systems*) and continuous time (*hard real-time systems*[2]) can be modelled. Then quality of service and performance evaluation were covered by formal methods. As a result, *hard real-time systems*, *stochastic systems*[3] and *probabilistic systems*[4] are also covered by formal methods. Formal methods have also expanded to new application domains such as computer security and bioinformatics. Considering the broad area of use, it is not at all surprising the diversity of methods that exists and the importance of choosing the right method.

Although there are many different formal methods that have been developed (mostly as research projects in academia), there is no broad adoption in the

[1] a mathematical modelling language for describing distributed systems

[2] a system where missing a deadline is a total system failure

[3] a system where the result of the system is undeterministic, but can be predicted by probability distributions

[4] a system where state transitions are done according to probabilities

industry with the exception of two areas: mission-critical systems (where mistakes are very expensive and sometimes impossible to correct after the system has been started) and life-critical systems (where there are legal requirements to adhere to specific standards) [28].

High-security information systems also require adherence to relevant standards, such as Common Criteria For Information Technology Security Evaluation[5]. However, Common Criteria only requires formal verification in the highest assurance levels, and there are only a few systems that are evaluated to such levels.

The authors of [28] also discovered that the use of formal methods in industry is not broad, it is mainly used to solve particular issues, and there is consensus neither on which formal methods to use nor for which part of the development process.

The EU project DEPLOY has on its website several FAQs related to where to use formal methods, for example: What important system concepts can be handled "elegantly" with a selected formal method? [6].

[5]http://www.commoncriteriaportal.org

[6]http://www.fm4industry.org/index.php/G-EA-1

5 Associations and agencies

Formal Methods Europe (FME)[1] is a world wide association in the area of formal methods for software development, with the aim of encouraging research and application of formal methods. Through their symposia and sponsored events they contribute with dissemination of research findings and industrial experience. The latest organised symposium (at the time of writing this document) is the 20th International Symposium on Formal Methods [2]. The symposium is organized once every 18 months and usually gathers 200–300 world-leading researchers from academia and industry.

The Defense Advanced Research Projects Agency (DARPA) has several projects that focus on or include formal methods. One such project is Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) that focuses on "the design of new computer systems that are highly resistant to cyber-attack, can adapt after a successful attack to continue rendering useful services, learn from previous attacks how to guard against and cope with future attacks, and can repair themselves after attacks have succeeded" [16]. Results from the project are gathered on DARPA's Open Catalog [3]. Among the results from this project there is Smten[4], a unified language for general-purpose functional programming and SMT query orchestration and ACL2[5], which is both a programming language for modelling computer systems and a tool for helping to prove properties of those models.

Another interesting DARPA project is the High-Assurance Cyber Military Systems (HACMS) with the goal "to create technology for the construction of high-assurance cyber-physical systems, where high assurance is defined to mean functionally correct and satisfying appropriate safety and security properties" [31]. Results from the project are also gathered on DARPA's Open Catalog [6].

[1] `http://www.fmeurope.org`
[2] `http://fm2015.ifi.uio.no/`
[3] `http://opencatalog.darpa.mil/CRASH.html`
[4] `http://www.cl.cam.ac.uk/research/security/ctsrd/smten.html`
[5] `http://www.cs.utexas.edu/users/moore/acl2/v6-3/`
[6] `http://opencatalog.darpa.mil/HACMS.html`

6 Limitations of formal methods

In this chapter we discuss the limitations of formal methods. We do not discuss the limitations of any specific formal method, but consider limitations of the area as a whole.

One important question about the use of formal methods is if it results in products with zero defects. The answer is that it results in products with very few defects, close to zero, but that there is still no guarantee that the product will be bug free. Moy and Wallenburg [42] list some reasons for why formally verified products may still contain defects:

- not all parts of the product are formally verified,
- formal verification can only guarantee that the specification has been followed, and therefore only the specified properties can be verified,
- not all properties can be formally verified, for example covert channels can be hard to detect as well as dead code, and
- the formal method and the tools used can have defects, for example the theorem prover.

Proofs can also be wrong, especially if they are done manually. Therefore researchers nowadays recommend the use of automated tools [45].

Another limitation is that there is no way to easily choose a suitable formal method. There are many formal methods available, especially ones developed be research groups in universities, and their maturity varies considerably.

Another aspect to consider is the needs of an organization, in terms of people, to start using formal methods. For most formal method tools it should be enough with one specialist in the development team. Nevertheless, for some formal methods, e.g., proof assistants, specialized mathematical skills are required [3].

7 Examples of practical use of formal methods

We present, in short, some projects where formal methods have been used outside academia and where security is a requirement. The focus is on what has been proved and, where possible, what assumptions have been made.

7.1 Mondex smart card

The Mondex application [55] consists of smart cards with electronic purses for electronic commerce. The smart cards are used for low-value cash-like transactions and require no third-party involvement. The security requirements are considered critical for this application, being vital that it is free of implementation or design bugs that would allow for subversion once in the field. Due to the critical security requirements, Mondex was certified to the highest standard available at the time: ITSEC Level E6 [33], which is equivalent to Common Criteria Evaluation Assurance Level 7 [19]. Mondex was the first product to achieve ITSEC Level E6.

The development of the Mondex application is described in [55]. Two models are developed: an abstract model and a low-level concrete model. The abstract model describes the world of purses and the exchange of value through atomic transactions and expresses the security properties that the card design needs to preserve. The concrete model includes the design of the purses and the message protocol for value exchange between purses. The models are described in the Z notation and proofs are carried out that the concrete model is a refinement of the abstract, which means that the concrete model abides by the abstract model's security requirements. The abstract model is simple and easy to understand, as are the security properties. The proofs were done manually, since the authors considered that there were no proof tools suited for such a large task at that time. In order to ensure proof correctness, the statements and many of the proof steps were type-checked using the *f*UZZ [53] and Formalizer [27] tools. Proofs were also independently checked by external evaluators.

The security properties were that:

- no value may be created in the system,
- all value must be accounted for in the system,
- all transfers are between authentic purses, and
- one needs sufficient funds to be able to transfer.

The proofs revealed a flaw in one of the minor protocols and the design was changed to rectify it [60].

One of the reasons for choosing manual proofs was the belief that automatising them would be very expensive. Nevertheless, Mondex was revised in 2006 as part of the Grand Challenge in Verified Software [59] with the goal of understanding the state of the art of mechanical verification. Eight research groups took up the challenge by using different formal methods and, as a result, the cost of automizing the Z proofs was 10% of the original development cost. An important result is that almost all formal methods used achieved the same level of automation.

7.2 Tokeneer secure entry system

Tokeneer [48] is a system developed by the NSA for securely accessing an enclave that has a controlled physical entry. The access control is done with help of biometrics and security tokens. When a user owning a security token passes the biometric tests and is allowed entry into the enclave, the token is loaded with authorisation information describing what the user is allowed to do during this visit in the enclave.

In order to show that it is possible to develop highly secure systems conforming with the Common Criteria Requirements at Evaluation Assurance Level 5 in a cost effective way, NSA decided to use formal methods for developing a well defined component of the Tokeneer system. The experiment was also time boxed and the skills needed to perform the development and the effort were monitored.

The component to be formally developed included a formal specification using the Z Notation, refinement of the specification to a formal design, software developed in SPARK (which is a subset of Ada with an accompanying tool-set) with proof of system properties and absence of run-time errors. Additional software was written (using traditional methods) in Ada to interface with peripherals.

The project was further on donated to the Verified Software Repository in 2008 and, as a result, was the subject of wide public scrutiny. J. Barnes notes that up to 2011 only five defects were found in the core software (the software that has been formally analysed) [7]. The defects were found by using a newer version of the tool-set and by critical review. More information about the defects can be found in [58].

Finding errors in this project is a good reminder that even when using formal methods there is no guarantee that all the defects will be found. Moy and Wallenburg [42] analysed the whole system including the software that has not been formally analysed, and found a total of 20 errors of which only two were in the core software. Considering that the whole system was written by the same developers, one can compare the number of errors found in the core software with the number found outside it and conclude that it is possible to build systems with considerable confidence by using formal methods.

7.3 The seL4 micro-kernel

The seL4 micro-kernel[1] is a third-generation micro-kernel that features abstractions for virtual address spaces, threads, inter-process communication and explicit in-kernel memory management model and capabilities for authorisation. The seL4 micro-kernel has been formally verified from its abstract specification down to its C implementation using machine checking. Correctness of a very detailed, low-level design is shown and the C implementation is formally verified. The formal proofs assume correctness of boot code, management of caches, the hardware and the hand-written assembly code[2].

A comprehensive description of the system and the formal methods used is given in [38]. According to the authors, seL4 is the only general-purpose OS kernel that is fully formally verified for functional correctness with machine checked end-to-end theorems. An abstract specification of the system was created as an operational model of the system. Then the Haskell programming language was used to implement a prototype of the kernel. This prototype could be automatically translated into the theorem prover Isabelle/HOL [44] to form an executable, design-level specification of the kernel. Then the model was manually implemented in C to form a high-performance C implementation of seL4. Refinement proofs link the specifications and the C code.

Functional correctness was proved for the kernel. This means that the behaviour of the binary implementation is fully captured by the abstract specification. As in the case of the Mondex application, this means that if properties can be proved to hold on the abstract model, then they will hold for the implementation as well. The functional correctness also means that the system will never fail and that it always has a defined behaviour. Some properties that are proved are that the kernel cannot be subverted by buggy encodings, spurious calls, maliciously constructed arguments to system calls or buffer overflow attacks.

Functional correctness means that the system is implemented correctly. Nevertheless, it does not mean that the right system is implemented (in this case "right" system means that the kernel has the high-level properties that are needed). To increase the confidence that the right system is implemented, formal verification has been used to verify that the system has desired security properties, such as authority confinement, integrity and confidentiality. The authors have great confidence that the proofs are logically correct since they are carried our in the Isabelle/HOA proof assistant and all derivations are from first principles. The security proofs are built on top of the functional correctness proofs and therefore inherit their assumptions. Furthermore, more assumptions are needed for the security proofs:

- for the integrity and confidentiality proofs: the system is correctly configured in accordance with a given access-control policy, and

[1]`https://sel4.systems/`

[2]parts of the kernel cannot be implemented in C, and are therefore written i assembly

- for the confidentiality proof, further system configuration assumptions are made: all non-timer interrupts are disabled and no partition has the authority to destroy any partition-crossing resource.

There are other logical assumptions for the security proofs that are well documented in [38]. The authors consider that all the assumptions made are reasonable for a high-assurance system.

Another aspect that is discussed is that the integrity theory is strong, which means that, if the contents of some memory cannot be changed by some thread, then that memory will remain unchanged under the proof assumptions. However, the confidentiality theory is not as strong since covert channels have to be considered that cannot be covered by the abstract model and this may result in information being inferred over such channels. One such example is timing channels, which are not modelled in the formal models. The authors believe that their security results enable such channels to be more easily identified and then complementary techniques can be applied.

The idea with seL4 is to be able to build secure applications on top of it and, for this, the authors present a method for provably bringing a system into a known configuration and protection state. This way one knows that the security properties will hold. Also, in such a system, both trusted and untrusted applications can run together.

The SMACCM project[3], funded by DARPA's HACMS program presented in Chapter 5, is developing an Unmanned Little Bird (ULB), which is an optionally-piloted autonomous helicopter with seL4 on-board.

[3]`http://ssrg.nicta.com.au/projects/TS/SMACCM/`

8 Functional correctness proof of the seL4 micro-kernel

We describe how functional correctness is proved for seL4, based on [38]. In this chapter, we choose to list code that the authors of seL4 have listed in several of their papers, and we also list some code that we take from their online repository available at `https://github.com/seL4`.

8.1 Kernel design process

Since OS developers usually take a bottom-up design approach and formal methods practitioners usually take a top-down design approach, the design approach used for seL4 is a compromise that combines both needs and uses the Haskell functional programming language. A Haskell prototype was written that can automatically be translated into an executable specification. When the Haskell prototype began to be stable, an abstract specification was also constructed in Isabelle/HOL.

Although the Haskell prototype is an executable model and implementation of the system, it is not used as the final production kernel. The model is reimplemented, manually, in C. One reason for not using the Haskell prototype is that it is a large amount of code and that it would be hard to verify its correctness. C allows for more optimisations of the low-level implementation. C code could have been automatically generated from the Haskell prototype, which could have simplified verification. Nevertheless, the authors consider that most opportunities to micro-optimise would then have been lost.

Klein, Derrin and Elphinstone report on their experience on implementing seL4 in [39], where they discuss in more detail the impact of the different choices made throughout the project.

8.2 Abstract specification

The abstract specification is written in Isabelle/HOL and describes what the system does without describing how it is done. Isabelle is a generic proof assistant where mathematical formulas can be expressed in a formal language and then be proved using logical calculus. Both the correctness of the software and properties of the protocols described can be proved.

At this level argument formats, encodings and error reporting are described. Memory and typed pointers are modelled explicitly. Non-determinism is used so implementation choices can be left for the lower levels. This means that if there exist multiple correct results for an operation, all of them are returned and it is clear that there is a choice. In the implementation, any of the correct results can be picked. As an example, we consider the Isabelle/HOL code for the scheduler presented in [38]:

```
schedule ≡ do
       threads ← allActiveTCBs;
       thread ← select threads;
       switch_to_thread thread
od OR switch_to_idle_thread
```

At the abstract level there is no scheduling policy, the scheduler is modelled as a function of picking any thread that runs and is active or the idle thread. The `select` statement picks any element of the set and the `OR` makes a non-deterministic choice.

The function allActiveTCBs is listed below:

```
text {* Gets all schedulable threads in the system. *}
definition
       allActiveTCBs :: "(obj_ref set,l'z::state_ext) s_monad" where
       "allActiveTCBs \<equiv> do
              state \<leftarrow> get;
              return {x. getActiveTCB x state \<noteq> None}
       od"
```

returns a list of all runnable threads. The function is implemented as an abstract, logical predicate over the system.

Further, global invariants[1] of seL4's abstract specification are proved using a monadic Hoare logic[2]. More information about the Hoare logic can be found in [17].

8.3 Executable specification

Haskell was used to write a prototype of the system. The Haskell program was then translated into Isabelle/HOL, resulting in an executable specification where more details are specified compared to the abstract specification and where it is specified how the kernel works. Initially, the translation was done manually, but was later automatized. The Hoare logic used for the translation is presented with examples in [17].

The executable specification is deterministic and all data structures are explicit data types, records and lists. If we take the scheduler example again, when selecting a thread, there is an explicit search backed by data structures for priority queues. The specification also expresses when the idle thread will be scheduled. Part of the Haskell code for the scheduler, found in [38], is:

```
schedule = do
       action <- getSchedulerAction
```

[1]Invariant – a property P, with respect to a program, that holds initially and that is preserved by all the atomic properties of the program.

[2]The Hoare logic is a way to connect code to its logic specification.

```
       case action of
              ChooseNewThread -> do
                     chooseThread
                     setSchedulerAction ResumeCurrentThread
...

chooseThread = do
       r <- findM chooseThread' (reverse [minBound .. maxBound])
       when (r == Nothing) $ switchToIdleThread

chooseThread' prio = do
       q <- getQueue prio
       liftM isJust $ findM chooseThread'' q

chooseThread'' thread = do
       runnable <- isRunnable thread
       if not runnable then do
              tcbSchedDequeue thread
              return False
       else do
              switchToThread thread
              return True
```

8.4 High-performance C implementation

The C code was also translated into Isabelle/HOL, and for that the C semantics were mapped into the theorem prover. The translation is, according to [37], correctness-critical and the semantics of the C subset were modelled precisely and foundationally. Precisely means that the C semantics, types and memory model are treated as prescribed by the C99 standard [32]. Foundationally means that the behaviour of C is derived from first principles[3] as far as possible.

The translation of the C code into the Isabelle/HOL representation is presented in detail with examples in [57].

8.5 Formal verification

The main property proved on the system is functional correctness, which means that the C implementation adheres to its specifications. Klein et al. [37] motivate why this kind of functional correctness proof is one of the strongest properties to prove about a system: once functional correctness is proved with respect to a model, the model can be used to establish further properties instead of having to reason about the low-level code.

[3]"A first principle is a basic, foundational proposition or assumption that cannot be deduced from any other proposition or assumption." [26]

The proof of functional correctness is done in Isabelle/HOL by using the abstract specification, the executable specification and the translation of C code into Isabelle/HOL. Funtional correctness is shown by formal refinement, which is first proved between the abstract and executable specification and second between the executable specification and the C code. The first proof in described in [17] and the second proof in [57]. The implementation refines the abstract specification if the behaviours of the implementation are a subset of the behaviours of the specification.

The two refinement layers result into one abstract-to-C refinement proof in Isabelle/HOL.

Furthermore, the authors have proved refinement between the C implementation and the kernel binary. This refinement is proved by using the HOL4 theorem prover[4] and two SMT solvers, Z3[5] and Sonolar[6]. More details about this refinement proof can be found in [38].

[4]`http://hol-theorem-prover.org/`

[5]`https://github.com/Z3Prover/z3/`

[6]`http://www.informatik.uni-bremen.de/agbs/florian/sonolar/`

9 Research groups and researchers in Sweden

We list the research groups that focus on formal methods in Sweden. The list in not intended to be exhaustive – it includes researchers we have spoken to and researchers that have been referred to in our interviews. We have also examined the main Swedish universities and, for each group, we list the senior members.

9.1 Chalmers/University of Gothenburg

At the *Software Technologies Division* at the *Department of Computer Science and Engineering*, there is a group on formal methods[1] that focuses on development and use of formal software verification approaches based on model checking, testing, automated reasoning and contract analysis. Some members of the group are:

Prof. Gerardo Schneider
Research interests: formal specification and analysis of contracts, formalisation of privacy policies, model checking, verification of real-time and polygonal hybrid systems, verification of embedded systems (in particular smart Java cards), semantics, logic for computer science, security.
`http://www.cse.chalmers.se/~gersch`

Assoc. Prof. Laura Kovacs
Research interests: formal software analysis and verification, more specifically designing new methods for computer-aided verification by combining automated theorem proving, automated assertion generation and symbolic computation.
`http://www.cse.chalmers.se/~laurako`

Assoc. Prof. Wolfgang Ahrendt
Research interests: model generation and disproving, automated deduction and formal methods in software engineering.
`http://www.cse.chalmers.se/~ahrendt`

There is also a language-based security group that focuses on development of security models and software construction methods for secure systems based on programming language technology. Some members of this group are:

Prof. Dave Sands
Research interests: programming languages, computer security, and

[1]`www.chalmers.se/en/departments/cse/organisation/st`

their combination.
`http://www.cse.chalmers.se/~dave`

Prof. Andrei Sabelfeld
Research interests: web security, data and application security, language-based security, and location privacy.
`http://www.cse.chalmers.se/~andrei`

Assis. Prof. Alejandro Russo
Research interests: rigorous theoretical programming language techniques, e.g. type-systems, instrumented-semantics, and label transition systems.
`http://alejandrorusso1.wix.com/personal-web-page`

9.2 Uppsala University

Some researchers in formal methods at the *Division of Computing Science* [2] at the *Department of Information Technology* are:

Lars-Henrik Eriksson, Senior Lecturer
Research interests: theory and implementation of logic programming, logical frameworks, and formal methods (specification, verification and synthesis).
`http://www.it.uu.se/katalog/lhe`

Prof. Bengt Jonsson
Research interests: formal methods, especially in connection with real-time and distributed systems, semantics of concurrent systems, and verification of concurrent systems.
`http://user.it.uu.se/~bengt/`

Prof. Wang Yi
Research interests: model-checking of real-time systems, multiprocessor scheduling and analysis, real-time applications on multicore.
`http://user.it.uu.se/~yi/`

9.3 Royal Institute of Technology/Stockholm University

The *Theory group*[3] at the *School of Computer Science and Communication (CSC)*, Royal Institute of Technology (KTH) and *Department of Numerical Analysis and Computer Science (Nada)*[4] at Stockholm University is conducting research on formal methods. Researchers interested in formal methods are:

[2]`http://www.it.uu.se/research/computing_science`
[3]`http://www.csc.kth.se/tcs/`
[4]Nada, a joint department of Stockholm University and KTH, is a part of CSC at KTH

Prof. Mads Dam
: **Research interests:** logic, semantics, programming languages, security, and communication networks.
`http://www.csc.kth.se/~mfd/`

Assoc. Prof. Dilian Gurov
: **Research interests:** program correctness, with focus on compositional reasoning.
`https://www.nada.kth.se/~dilian`

9.4 Mälardalen University

Formal Modelling and Analysis of Embedded Systems group at *Division of Embedded Systems*[5] is focusing on formal modelling, analysis, and verification techniques for real-time embedded systems. In particular, formal syntax and semantics of component-based and service-oriented models with extra-functional properties such as time or resources. The group leaders are:

Prof. Paul Pettersson
: **Research interests:** component-based design and model-based verification techniques, in particular model-checking and model-based testing, for real-time and embedded systems.
`http://www.es.mdh.se/staff/166-Paul_Pettersson`

Assoc. Prof. Cristina Seceleanu
: **Research interests:** developing formal models and verification techniques for designing predictable real-time embedded systems.
`http://www.es.mdh.se/staff/173-Cristina_Seceleanu`

9.5 Linköping University

Real-Time Systems Lab at the *Software and Systems Division*[6] within the *Department of Computer and Information Science* is conducting research in the areas of dependability and resource allocation in distributed systems. The research is applicable in many areas, one example is fault-tolerant and mission-critical systems in disaster area networks.

Prof. Simin Nadjm-Tehrani
: **Research interests:** formal analysis of safety and fault tolerance.
`http://www.ida.liu.se/~simna73`

[5] `http://www.es.mdh.se/divisions/11-Division_of_Embedded_Systems`
[6] `https://www.ida.liu.se/divisions/sas`

10 Questions

We contacted several researchers in the area of formal methods from different universities in Sweden and asked them about the area of formal methods in software engineering. The goal was to obtain, among other things, relevant literature and examples of projects where formal methods were implemented in the industry.

In our discussions with the researchers we have used the following questions:

1. What should I read to get an overview and a basic understanding of the area of formal methods?
2. What formal methods are mature enough to use in practice?
3. What formal methods related to software security requirements are there? Are they mature and can be used in practice?
4. Examples of industrial projects (both successful and unsuccessful)? Some IT security related projects?
5. What do I need to know in order to start using formal methods? How do I choose a suitable formal method?
6. How can formal methods be used to ensure the separation logic (one important aspect is the separation of information between different levels of classified information)? Do you know of any projects where they try this?

11 Discussion and conclusions

Results from actual use of formal methods in software engineering show that there are usable formal methods that work. Studies also show that their use can be cost and time effective. Formal methods projects spend more time in the specification phase, which results in fewer bugs in the later phases of the development. It is nevertheless a considerable change to introduce formal methods and therefore the gain in efficiency comes from using formal methods in several projects. Formal methods can be introduced stepwise in the development process and they will give effect from the beginning. For example one could write a formal specification and then continue as usual.

We present several examples of practical use of formal methods to illustrate that formal methods have successfully been applied in real-world applications. Choosing an appropriate formal method seems to be a challenge in itself, where careful consideration needs to be taken as to which system and which application domain are targeted and which properties that need to be proved. We present classifications for the system, system properties and application domain, which can guide the development team in choosing an appropriate method. In most cases one formal methods specialist is enough in the development team to introduce formal methods into the design process. Furthermore, different formal methods can be used during the different stages in the development process. We therefore also present a classification of formal methods based on their functionality.

Several of the standards used for system development require formal methods at the highest levels of accreditation. Some examples are the Common Criteria standard and the DO-178C standard *Software Considerations in Airborne Systems and Equipment Certification.* The fact that formal methods is a requirement in different standards indicates that it is considered a valuable asset in the verification and validation process.

We consider formal methods to be an interesting field for the Swedish Armed Forces since it can reduce the number of defects in the system and, since this reduction is achieved early in the development process, formal methods have the potential to reduce the cost of the whole system. Nevertheless, a more thorough investigation would be necessary to better understand how the Swedish Armed Forces could begin using formal methods and which projects or which parts of the projects would be best suited to use them.

To conclude, the literature in the area indicates that using formal methods can be cost-effective and can reduce the number of possible defects to close to zero. Nevertheless, it is an evolving field and, to take full advantage of its potential, it is advisable to follow the development of new methods and tools and keep looking for new ones.

Bibliography

[1] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. ”Designing Safe, Reliable Systems Using Scade”. In: *Proceedings of the First International Conference on Leveraging Applications of Formal Methods*. ISoLA04. Paphos, Cyprus: Springer-Verlag, 2006, pp. 115–129. DOI: 10.1007/11925040_8.

[2] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.

[3] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. Melo de Sousa. ”Rigorous Software Development”. In: Springer Verlag London, 2011. Chap. An Overview of Formal Methods Tools and Techniques, pp. 15–44.

[4] R. Alur and D. L. Dill. ”A Theory of Timed Automata”. In: *Theoretical Computer Science* 126.2 (Apr. 1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[5] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[6] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Revised. Vol. 103. North Holland, 1984. URL: http://www.cs.ru.nl/~henk/Personal%20Webpage.

[7] J. E. Barnes. ”Experiences in the Industrial use of Formal Methods.” In: *ECEASST* 46 (2011). URL: http://dblp.uni-trier.de/db/journals/eceasst/eceasst46.html#Barnes11.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. ”CVC4”. In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV11. Snowbird, UT: Springer-Verlag, 2011, pp. 171–177. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. ”Symbolic Model Checking Without BDDs”. In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS 99. London, UK: Springer-Verlag, 1999, pp. 193–207. URL: http://dl.acm.org/citation.cfm?id=646483.691738.

[10] D. Bjørner. *Software Engineering 1*. Springer-Verlag Berlin Heidelberg, 2006.

[11] D. Bjørner. *Software Engineering 2*. Springer-Verlag Berlin Heidelberg, 2006.

[12] D. Bjørner. *Software Engineering 3*. Springer-Verlag Berlin Heidelberg, 2006.

[13] D. Bjørner and K. Havelund. ”FM 2014: Formal Methods”. In: vol. 8442. LNCS. Springer International Publishing Switzerland, 2014. Chap. 40 Years of Formal Methods — Some Obstacles and Some Possibilities?, pp. 42–61.

[14] J.-L. Boulanger. *Industrial Use of Formal Methods.* Wiley, 2012.

[15] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. ”LUSTRE: A Declarative Language for Real-time Programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL 87. Munich, West Germany: ACM, 1987, pp. 178–188. DOI: 10.1145/41625.41641.

[16] *Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) project.* URL: http://www.darpa.mil/program/clean-slate-design-of-resilient-adaptive-secure-hosts.

[17] D. Cock, G. Klein, and T. Sewell. ”Secure Microkernels, State Monads and Scalable Refinement”. In: *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics.* Montreal, Canada: Springer, Aug. 2008, pp. 167–182.

[18] R. Cockett and T. Fukushima. *About Charity.* Yellow Series Report No. 92/480/18. Department of Computer Science, The University of Calgary, June 1992.

[19] *Common Criteria for Information Technology Security Evaluation. Part 1: Introduction and general model.* Tech. rep. CCMB-2012-09-001, Version 3.1, Revision 4. Common Criteria Recognition Agreement, Sept. 2012.

[20] P. Cousot and R. Cousot. ”Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL 77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[21] L. De Moura and N. Bjørner. ”Z3: An Efficient SMT Solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* TACAS08/ETAPS08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[22] V. D'Silva, D. Kroening, and G. Weissenbacher. ”A Survey of Automated Techniques for Formal Software Verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27.7 (July 2008), pp. 1165–1178.

[23] B. Dutertre and L. de Moura. *The Yices SMT solver.* Tech. rep. SRI International, 2006. URL: http://yices.csl.sri.com/tool-paper.pdf.

[24] L.-H. Eriksson. *Matematik kan ge färre fel i datorprogram.* URL: `http://user.it.uu.se/~lhe/fmpop.pdf`.

[25] F. M. Europe. *Choosing a Formal Method.* URL: `http://www.fmeurope.or/?page_id=264`.

[26] *First principle.* URL: `https://en.wikipedia.org/wiki/First_principle`.

[27] M. Flynn, T. Hoverd, and D. Brazier. "Formaliser — An Interactive Support Tool for Z". In: *Z User Workshop.* Ed. by J. Nicholls. Workshops in Computing. Springer London, 1990, pp. 128–141. DOI: `10.1007/978-1-4471-3877-8_8`.

[28] D. H. Garavel and D. S. Graf. *Formal Methods for Safe and Secure Computers Systems.* Tech. rep. Federal Office for Information Security, 2013.

[29] M. J. C. Gordon and T. F. Melham, eds. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* New York, NY, USA: Cambridge University Press, 1993.

[30] A. Hall. "Seven Myths of Formal Methods". In: *IEEE Software* 7.5 (Sept. 1990), pp. 11–19. DOI: `10.1109/52.57887`.

[31] *High-Assurance Cyber Military Systems (HACMS).* URL: `http://www.darpa.mil/program/high-assurance-cyber-military-systems`.

[32] ISO/IEC JTC1 SC22 WG14. *ISO/IEC 9899:TC2 Programming Languages - C.* Tech. rep. May 2005. URL: `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`.

[33] ITSEC. *Information technology security evaluation criteria (ITSEC): Preliminary harmonised criteria.* Tech. rep. Document COM(90), Version 1.2. Commission of the European Communities, July 1991.

[34] C. B. Jones. *Software development : a rigorous approach.* Englewood Cliffs, N.J. Prentice/Hall International, 1980. URL: `http://opac.inria.fr/record=b1085664`.

[35] R. Juellig, Y. Srinivas, and J. Liu. "SPECWARE: An advanced environment for the formal development of complex software systems". In: *Algebraic Methodology and Software Technology.* Ed. by M. Wirsing and M. Nivat. Vol. 1101. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 551–554. DOI: `10.1007/BFb0014339`.

[36] G. Klein. "Operating system verification — An overview". In: *Sadhana* 34.1 (Feb. 2009), pp. 27–69.

[37] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. "seL4: Formal Verification of an Operating-System Kernel". In: *Communications of the ACM* 53.6 (June 2010), pp. 107–115. DOI: `10.1145/1743546.1743574`.

[38] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (Feb. 2014), 2:1–2:70. DOI: `10.1145/2560537`.

[39] G. Klein, P. Derrin, and K. Elphinstone. "Experience Report: SeL4: Formally Verifying a High-performance Microkernel". In: *SIGPLAN Not.* 44.9 (Aug. 2009), pp. 91–96. DOI: `10.1145/1631687.1596566`.

[40] G. Leduc and F. Germeau. "Verification of security protocols using LOTOS-method and application". In: *Computer Communications* 23.12 (Sept. 2000), pp. 1089–1103. DOI: `10.1016/S0140-3664(99)00239-X`.

[41] G. Lowe. "An Attack on the Needham-Schroeder Public-key Authentication Protocol". In: *Information Processing Letters* 56.3 (Nov. 1995), pp. 131–133. DOI: `10.1016/0020-0190(95)00144-2`.

[42] Y. Moy and A. Wallenburg. "Tokeneer: Beyond Formal Program Verification". In: *Proc. 5th Int. Congress on Embedded Real Time Software and Systems (ERTS'10).* Toulouse, France, May 2010.

[43] *NASA Langley Formal Methods.* URL: `http://shemesh.larc.nasa.gov/fm/fm-what.html`.

[44] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Berlin, Heidelberg: Springer-Verlag, 2002.

[45] J. Parrow. *The Greatest Challenge.* URL: `http://opct2014.cs.vu.nl/?post_type=document&p=286`.

[46] L. C. Paulson. *Isabelle: a generic theorem prover.* Lecture notes in computer science. Berlin, New York: Springer-Verlag, 1994. URL: `http://opac.inria.fr/record=b1085788`.

[47] B. Potter, D. Till, and J. Sinclair. *An Introduction to Formal Specification and Z.* 2nd. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1996.

[48] L. Reinert and S. Luther. *Tokeneer User Authentication Techniques Using Public Key Certificates Part 3: An Example Implementation.* Tech. rep. NSA Central Security Service, Feb. 1998.

[49] A. Romanovsky and M. Thomas, eds. *Industrial Deployment of System Engineering Methods.* Springer-Verlag Berlin Heidelberg, 2013.

[50] J. Rushby. *Formal Methods and their Role in the Certification of Critical Systems.* Tech. rep. CSL-95-1. accessed 2015-04-30. Computer Science Laboratory, SRI International, 1995. URL: `http://sdg.csail.mit.edu/6.894/dnjPapers/rushby-tr.pdf`.

[51] K. Schneider. "Verification of Reactive Systems: Formal Methods and Algorithms". In: SpringerVerlag, 2004. Chap. Introduction, pp. 1–43.

[52] I. Sommerville. "SOFTWARE ENGINEERING 9". In: Web chapter, 2010. Chap. 27 Formal Specification. URL: `http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/WebChapters/PDF/Ch_27_Formal_spec.pdf`.

[53] J. Spivey. *The fuzz Manual*. Tech. rep. 2nd edition. 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK: Computing Science Consultancy, 1992.

[54] T. M. Stefania Gnesi. *Formal Methods for Industrial Critical Systems: A Survey of Applications*. Wiley-IEEE Computer Society Pr, 2012.

[55] S. Stepney, D. Cooper, and J. Woodcock. *An Electronic Purse: Specification, Refinement, and Proof*. Technical monograph PRG-126. Oxford University Computing Laboratory, July 2000.

[56] S. Thompson. *Haskell: The Craft of Functional Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[57] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. "Mind the Gap: A Verification Framework for Low-Level C". In: *Theorem Proving in Higher Order Logics*. Ed. by S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel. Vol. 5674. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 500–515. DOI: `10.1007/978-3-642-03359-9_34`.

[58] J. Woodcock, E. G. Aydal, and R. Chapman. "The Tokeneer Experiments". In: *Reflections on the Work of C.A.R. Hoare*. Ed. by A. Roscoe, C. B. Jones, and K. R. Wood. Springer London, 2010, pp. 405–430. DOI: `10.1007/978-1-84882-912-1_17`.

[59] J. Woodcock and R. Banach. "The Verification Grand Challenge". In: *j-jucs* 13.5 (May 2007), pp. 661–668. URL: `http://www.jucs.org/jucs_13_5/the_verification_grand_challenge`.

[60] J. Woodcock, P. Gorm Larsen, J. Bicarregui, and J. Fitzgerald. "Formal Methods: Practice and Experience". In: *ACM Computing Surveys* 41 (Oct. 2009), 19:1–19:36. DOI: `10.1145/1592434.1592436`.