Chapter 1
Formal Methods

Markus Roggenbach, Bernd-Holger Schlingloff, and Gerardo Schneider

Abstract Formal Methods are one means in software engineering that can help to ensure that a computer system meets its requirements. Using examples from space industry and every programmer's daily life, we carefully develop an understanding of what constitutes a Formal Method. Formal Methods can play multiple roles in the software design process. Some software development standards actually require the use of Formal Methods for high integrity levels. Mostly, Formal Methods help to make system descriptions precise and to support system analysis. However, their application is feasible only when they are supported by tools. Consequently, tool qualification and certification play a significant role in standards. Formal Methods at work can be seen in the many (academic) surveys, but also in numerous published industrial success stories. Hints on how to study Formal Methods in academia and on how to apply Formal Methods in industry conclude the chapter.

1.1 What Is a Formal Method?

You have just bought a book on Formal Methods and are making holiday plans in the Caribbean to read it on the beach. In order to guarantee the reservation, your travel agent requires a deposit. You decide to pay electronically via credit card.

When performing such a transaction, obviously you have certain expectations on the electronic payment system. You don't want the agent to be

Markus Roggenbach
Swansea University, Wales, United Kingdom

Bernd-Holger Schlingloff
Humboldt University and Fraunhofer FOKUS, Berlin, Germany

Gerardo Schneider
University of Gothenburg, Sweden

able to withdraw more than required. The agent wants at least the amount which was asked for. Thus, both you and the agent expect that the payment system gets its numbers right. The payment should go through – as, clearly, your credit card is a valid one. Also, you don't want too much information to be disclosed, e.g., your PIN should stay secret. The transaction should solely concern the holiday reservation, no further contracts shall follow from this. Finally, you want to be able to use the system without the need to consult a user manual of hundreds of pages. All these points are typical requirements for an electronic payment system. Formal Methods are one way how software engineering can help to ensure that a computer system meets such requirements.

So, what is a Formal Method? Instead of trying to start with a comprehensive definition of the term, we give two motivating examples.

1.1.1 An Application in Space Technologies

Formal Methods are often used in safety-critical areas, where human life or health or a large sum of money depends on the correctness of software. We start with an example from the largest aerospace project mankind has endeavored so far.

Example 1: ISS Fault Tolerant Computer

The International Space Station (ISS) which was docked on November 2^{nd}, 2000 (ISS-Expedition 1), has provided a platform to conduct scientific research that cannot be performed in any other way.

At the heart of the ISS is a fault tolerant computer (FTC) "to be used in the ISS to control space station assembly, reboost operations for flight control and data management for experiments carried out in the space station" [BKPS97].

In outer space, the probability of hardware faults due to radiation is much higher than on earth. Thus, in the ISS-FTC there are four identical interconnected hardware boards, which perform essentially the same computation. A software fault management layer is responsible for detecting, isolating, rebooting and reintegrating malfunctioning boards.

One problem in the design of this layer is the recognition of a faulty board, since it not only can generate wrong messages, but also modify messages of the other (correct) boards. To overcome this problem, a so-called Byzantine agreement protocol is used, which abstractly models the problem of distributed consensus in the presence of faults.

Lamport et al. use the following story to exemplify the distributed consensus problem [LSP82]:

> We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that
>
> - **A.** All loyal generals decide upon the same plan of action.
>
> The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition **A** regardless of what the traitors do. The loyal generals should not only reach agreement, but should agree upon a reasonable plan. We therefore also want to ensure that
>
> - **B.** A small number of traitors cannot cause the loyal generals to adopt a bad plan.

In Lamport's paper, various pseudocode algorithms for this problem are given and proven to be correct. For these proofs, certain assumptions about the possible actions of the generals are made, e.g., that a traitorous general may send different, contradicting messages (attack and retreat) to different other divisions.

Even though Lamport et al. prove their algorithms to be correct, the questions on whether the communication by messengers can block (deadlock) or the exchange of a message can lead to infinite internal chatter (livelock) in the communication system are not in the scope of his consideration.

Example 1.1: ISS Fault Tolerant Computer – Findings

For the implementation of the fault management layer in the FTC, one of the algorithms presented by Lamport et al. [LSP82] was coded in the programming language Occam.

As mentioned above, the algorithm is proven to be correct, and great care was taken to assure that the actual code matches the pseudocode as closely as possible. However, this still did not guarantee that the software worked as expected: in a series of publications [BKPS97, BPS98, PB99], Buth et al. report that using code abstraction into the process algebra Csp,

- "seven deadlock situations were uncovered", and
- "about five livelocks were detected"

in the software of the FTC fault management layer.

The language Occam uses *synchronous communication* between tasks: the sender of a message is blocked until the receiver is willing to pick up this message. With such a communication paradigm, a deadlock can occur if two actors send each other messages at the same time. Thus, even though the algorithm on which the code is based was proven to be correct on the conceptual

layer, there still was the possibility of errors in the underlying communication layer.

In Ch. 3, which is devoted to the process algebra Csp, we will give precise definitions of deadlock and livelock. Furthermore, we will provide proof techniques to show their absence.

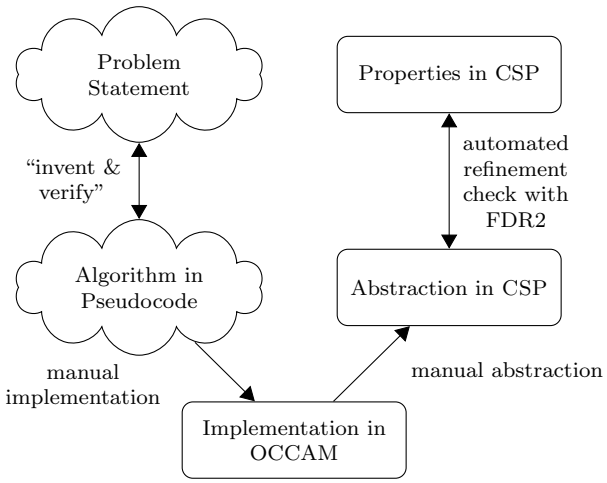

Fig. 1.1 The overall verification approach [BKPS97, BPS98, PB99].

Example 1.2: ISS Fault Tolerant Computer – Reflections

The programming language Occam has been designed as an implementation language for the process algebra Csp. Thus, it is rather easy to abstract an Occam program, e.g., the fault management layer of the FTC, into Csp. Compared to the original Occam program, the Csp abstraction has a significantly reduced state space, since all computations that have nothing to do with the Byzantine agreement protocol can be omitted. However, it still preserves the deadlocks and livelocks of the original program.

Csp has a formal semantics and proof methods to verify properties. In this example, the FDR tool was used to automatically analyse the Csp code and thus – indirectly – the Occam program. If a deadlock or livelock is found in the Csp abstraction, FDR generates a sequence of events which exhibits the problem. Fig. 1.1 shows the overall verification approach. The sequence generated by FDR can be used to trace the problem in the original code, which then can be analysed and corrected.

Interestingly, most of the observed issues concerned the message exchange, i.e., the layer which was not in the scope of the formal correctness proof of the algorithm. This situation is rather typical in such a context. In Sect. 3.3, we will discuss in more detail how to avoid such problems.

In Example 1, Formal Methods were used for quality assurance rather than in the design and implementation phase. For the process algebra Csp, we will discuss a different approach in Sect. 3.3. There, the idea is to automatically translate a Csp model into a C++ program. Before translation, the model can first be analysed using a model checker such as FDR. The C++ program can then be enriched with additional functionality without compromising the properties established earlier for the Csp model.

1.1.2 An Everyday Application

Our second example is from the area of formal languages and text processing. It shows the importance of having a precise formal semantics even in common-day tools such as text processing.

Example 2: Text Processing

Assume that we want to replace all occurrences of certain patterns in a text, e.g., remove all comments from an HTML document. In HTML, comments are marked beginning with '`<!--`' and ending with '`-->`'. Most editors offer a facility for replacement based on *regular expressions*, that is, you may specify the symbol '`*`' as a wildcard in the search. With this, replacing '`<!--*-->`' by an empty string yields the desired result.

Regular replacements are a convenient tool for text processing. However, the semantics (meaning) is not always easy to understand.

The wildcard sign is explained in the documentation of Word 2007 as

> matching any string of characters. Word does not limit the number of characters that the asterisk can match, and it does not require that characters or spaces reside between the literal characters that you use with the asterisk.

For GNU Emacs, it is defined by the following explanation:

> The matcher processes a '*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '*'-modified construct in case that makes it possible to match the rest of the pattern.

These descriptions might or might not be intelligible to the ordinary reader. However, if the text processing component is used as part of a safety-critical tool chain, it is important that it has a clear semantics. Imagine that the regular replacement is used for macro expansion as part of a compiler. In

this case, it is essential that the result of any replacement command is unique and predictable.

Example 2.1: Tool Experiments for Text Processing

What happens if we replace the wildcard sign '*' by the single character 'x'? As original text we take the string 'abc'.

- In Word, the result of replacing '*' by 'x' in 'abc' is 'xxxx';
- in Word, the same result is returned when taking the wildcard symbol '?@';
- in Emacs, replacing '*' by 'x' in 'abc' gives 'abc';
- in Emacs, replacing '.*' by 'x' in 'abc' gives 'x'.

This might come as a surprise.

The problem is that both for Word and Emacs, there is no formal semantics of "replacement of regular expressions". Whereas the syntax of admissible regular expressions is (more or less) fixed in the documentation, the semantics is only informally explained.

This example allows us to show the key ingredients of a Formal Method: syntax, semantics and method.

Syntax

Syntactically, each Formal Method deals with objects from a formal language. A *formal language* is a well-defined set of words from a given alphabet. Usually it is defined by a *grammar*, which is a set of rules determining the membership of the language.

There are also Formal Methods dealing with graphical objects (e.g., Petri nets). In most of these cases, there is a textual description of these objects as well (e.g., in XML). For such models, the syntax is fixed by a metamodel, which determines membership of the class of graphical objects under consideration.

Example 2.2: Syntax of Regular Expressions

Given an alphabet $\mathcal{A}$, the language of regular expressions is given by the following grammar:

- every letter from the alphabet is a regular expression.
- $\emptyset$ is a regular expression.
- if φ and ψ are regular expressions, then $(\varphi\ \psi)$ and $(\varphi + \psi)$ are regular expressions.
- if φ is a regular expression, then φ^* is a regular expression.

The same definition can be written in so-called Backus-Naur-Form [Bac59]:

$$Regexp_{\mathcal{A}} \;::=\; \mathcal{A} \;\mid\; \emptyset \;\mid\; (Regexp_{\mathcal{A}}\; Regexp_{\mathcal{A}}) \;\mid\; (Regexp_{\mathcal{A}} + Regexp_{\mathcal{A}}) \;\mid\; Regexp_{\mathcal{A}}^{*}$$

According to this definition, each regular expression is a string which contains only letters of the alphabet and the symbols '$\emptyset$', '(', ')', '+' and '*' (the so-called *Kleene-star*).[a] Of course, for such a definition to make sense, these symbols themselves must not be letters.[b] Backus-Naur-Form (BNF) notation will be used also for several other formal languages in this book.

[a] If the alphabet contains letters composed of several characters, there might be several ways to parse a given string into a regular expression. For example, if $\mathcal{A} = \{a, aa\}$, then (aaa) could be read as $(a\ aa)$ and $(aa\ a)$. A solution to this problem is to require appropriate white spaces in strings.

[b] To allow the use of special symbols in the alphabet, some tools use a 'quoting' mechanism: '\+' refers to the letter of the alphabet, whereas '+' denotes the symbol.

The benefit of having a 'minimal' syntax is that the definition of semantics and proofs are simplified. For practical applications, often the core of a formal language is extended by suitable definitions.

Example 2.3: Extended Syntax of Regular Expressions

Assume that the alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ is finite.

Most text processing systems allow the following abbreviations, including the above mentioned '*' and '.*' notation of Word and GNU Emacs. In this book, the symbol "$\triangleq$" stands for "equal by definition" or "is defined as".

- $\varepsilon \triangleq \emptyset^*$ ('the empty word'),
- $\varphi^+ \triangleq (\varphi\ \varphi^*)$ ('one or more repetitions of φ')
- $\boldsymbol{.} \triangleq (((a_1 + a_2) + \ldots) + a_n)$ ('any letter'),
- $* \triangleq \boldsymbol{.}^*$ ('any word'),
- $\varphi? \triangleq (\varepsilon + \varphi)$ ('maybe one φ'),
- $\varphi^0 \triangleq \varepsilon$ and $\varphi^n \triangleq (\varphi\, \varphi^{n-1})$ for any $n > 0$ ('exactly n times φ'),
- $\varphi^n_m \triangleq (\varphi^m \varphi?^{n-m})$ for $0 \leq m \leq n$ ('at least m and at most n φ')

  (Here we assume that $\{\emptyset, +, {}^*, (,), \varepsilon, {}^+, \boldsymbol{.}, *, ?, {}^n, {}^n_m\} \cap \mathcal{A} = \emptyset$).

Semantics

In the context of Formal Methods, a formal language comes with a formal semantics which explains the meaning of the syntactical objects (words or graphs) under consideration by interpreting it in some domain.

The semantics identifies for each syntactical object a unique object in the chosen interpretation domain. Probably the fundamental question which can

be answered by a semantics is: when can two different syntactical objects be considered equal? For our regular expression case study this means to determine when two different expressions are to be the same. As another example from computer science, we would like to know whether two different programs compute the same result.

Other questions include whether one object entails another one. For instance, we would like to know whether one regular expression includes another one, one program extends another one, or one specification refines another one.

In contrast to syntax, the semantics of Formal Methods is not always decidable. That is, membership of a word or model in the formal language defined by a grammar or metamodel is usually trivial to decide. Semantical equality, however, is often undecidable as can be seen by the example of program equivalence.

There are three main ways of defining a semantics for formal languages: denotational, operational, and axiomatic.

In denotational semantics the *denotation* of an object is defined. That is, a denotational semantics is a function defining for each syntactic object an object in some semantical domain. For example, a regular expression denotes a language (a set of words) over the alphabet. That is, the semantical domain for regular expressions is the set of all languages. As another example, a program in a functional language denotes a function (set of tuples) from the input parameters to the output type. In Ch. 3 we will discuss three different denotational semantics for the process algebra CSP.

Example 2.4: Denotational Semantics of Regular Expressions

For any regular expression φ, we define the denoted language $[\![\varphi]\!]$ by the following clauses:

- $[\![a]\!] \triangleq \{a\}$ for any $a \in \mathcal{A}$. That is, the regular expression 'a' defines the language consisting solely of the one-letter word 'a'.
- $[\![\emptyset]\!] \triangleq \{\}$. That is, $\emptyset$ denotes the empty language.
- $[\![(\varphi\ \psi)]\!] \triangleq \{xy \mid x \in [\![\varphi]\!], y \in [\![\psi]\!]\}$. That is, $(\varphi\ \psi)$ denotes the language of all words which can be split into two parts, such that the first part is in the denotation of φ and the second in the denotation of ψ.
- $[\![(\varphi + \psi)]\!] \triangleq [\![\varphi]\!] \cup [\![\psi]\!]$. That is, $(\varphi + \psi)$ denotes the union of the denotations of φ and ψ.
- $[\![\varphi^*]\!] \triangleq \{x_1 \ldots x_n \mid n \geq 0,$ and for all $i \leq n$, $x_i \in [\![\varphi]\!]\}$.
  That is, φ^* denotes the language of all words which can be split into a finite number of n parts, such that each part is in the denotation of φ.
  With the special case $n = 0$, this definitions entails that for any φ, the empty word (consisting of zero letters) is in $[\![\varphi^*]\!]$.

Operational semantics describes the execution of the syntactic object by some virtual machine. In our example, from each regular expression we can construct an automaton accepting its language. For an imperative or object-oriented programming language, the operational semantics defines, for instance, the change of the memory content induced by an assignment. In Ch. 3 we will discuss an operational semantics for the process algebra Csp.

Example 2.5: Operational Semantics of Regular Expressions

For any regular expression φ, we define an *automaton* $\mathbf{A}(\varphi)$, that is, a graph (N, E, s_0, S_F), where N is a nonempty set of *nodes*, $E \subseteq (N \times \mathcal{A} \times N) \cup (N \times N)$ is a set of (labelled) *edges*, $s_0 \in N$ is the *initial node* and $S_F \subseteq N$ is the set of *final nodes*.

- $\mathbf{A}(a) \triangleq (\{s_0, s_1\}, \{(s_0, a, s_1)\}, s_0, \{s_1\})$ for any letter $a \in \mathcal{A}$.
- $\mathbf{A}(\emptyset) \triangleq (\{s_0\}, \{\}, s_0, \{\})$.
- If $\mathbf{A}(\varphi) = (N_\varphi, E_\varphi, s_{0,\varphi}, S_{F,\varphi})$ and $\mathbf{A}(\psi) = (N_\psi, E_\psi, s_{0,\psi}, S_{F,\psi})$ (where we assume all elements to be disjoint), then $\mathbf{A}((\varphi\ \psi)) \triangleq (N_\varphi \cup N_\psi, E_\varphi \cup E_\psi \cup \{(s, s_{0,\psi}) \mid s \in S_{F,\varphi}\}), s_{0,\varphi}, S_{F,\psi})$.
- $\mathbf{A}((\varphi + \psi))$ is constructed from $\mathbf{A}(\varphi)$ and $\mathbf{A}(\psi)$ by $\mathbf{A}((\varphi + \psi)) \triangleq (N_\varphi \cup N_\psi \cup \{s_0\}, E_\varphi \cup E_\psi \cup \{(s_0, s_{0,\varphi}), (s_0, s_{0,\psi})\}, s_0, S_{F,\varphi} \cup S_{F,\psi})$, where s_0 is a new node not appearing in N_φ or N_ψ.
- If $\mathbf{A}(\varphi) = (N, E, s_{0,\varphi}, S_F)$, then $\mathbf{A}(\varphi^*) \triangleq (N \cup \{s_0\}, E \cup \{(s_0, s_{0,\varphi})\} \cup \{(s, s_0) \mid s \in S_F\}, s_0, \{s_0\})$, where again s_0 is a new node not appearing in N_φ.

A word w is *generated* or *accepted* by an automaton, if there is a path from the initial node to some final node which is labelled by w. It is not hard to see that for every regular expression φ the automaton $\mathbf{A}(\varphi)$ accepts exactly $[\![\varphi]\!]$. That is, denotational and operational semantics coincide.

An axiomatic semantics gives a set of proof rules from which certain properties of the syntactical object can be derived. For example, for regular expressions an axiomatic semantics might consist of a list of rules allowing to prove that two expressions are equal. For logic programming languages, the axiomatic semantics allows to check if a query is a consequence of the facts stated in the program. In Ch. 2 we will discuss a Hilbert-style proof system for propositional logic, which – thanks to its correctness and completeness – can also serve as axiomatic semantics of propositional logic.

Example 2.6: Axiomatic Semantics of Regular Expressions

Axiomatic systems for equality of regular expressions were given by various authors [Sal66, Koz94, KS12]. We call an equation $\alpha = \beta$ *derivable* and write $\vdash \alpha = \beta$, if it is either an instance of one of the axioms below

or follows from a set of such instances by a finite number of applications of the below rules. Salomaa gives the following axioms

- $\vdash (\alpha + (\beta + \gamma)) = ((\alpha + \beta) + \gamma)$, $\vdash (\alpha\ (\beta\ \gamma)) = ((\alpha\ \beta)\ \gamma)$
- $\vdash (\alpha\ (\beta + \gamma)) = ((\alpha\ \beta) + (\alpha\ \gamma))$, $\vdash ((\alpha + \beta)\ \gamma = ((\alpha\ \gamma) + (\beta\ \gamma))$
- $\vdash (\alpha + \beta) = (\beta + \alpha)$, $\vdash (\alpha + \alpha) = \alpha$, $\vdash (\varepsilon\ \alpha) = \alpha$
- $\vdash (\emptyset\ \alpha) = \emptyset$, $\vdash (\alpha + \emptyset) = \alpha$
- $\vdash \alpha^* = (\varepsilon + (\alpha^*\ \alpha))$, $\vdash \alpha^* = (\varepsilon + \alpha)^*$

and derivation rules

- If $\vdash \alpha = \beta$ and $\vdash \gamma = \delta$, then $\vdash \gamma[\alpha := \beta] = \delta$ and $\vdash \gamma[\alpha := \beta] = \gamma$
- If $\vdash \alpha = ((\alpha\beta) + \gamma)$ and not $\varepsilon \in \beta$, then $\vdash \alpha = (\gamma\beta^*)$

Here, $\gamma[\alpha := \beta]$ means γ with one or more occurrences of α replaced by β. For the second rule, $\varepsilon \in \beta$ means that

1. β is of form ρ^* for some regular expression ρ, or
2. β is of form $(\rho_1 + \rho_2)$ where $\varepsilon \in \rho_1$ or $\varepsilon \in \rho_2$, or
3. β is of form $(\rho_1\ \rho_2)$ where $\varepsilon \in \rho_1$ and $\varepsilon \in \rho_2$.

Without the restriction "not $\varepsilon \in \beta$" the rule would not be correct: $a^* = (a^*a^*) + \emptyset$, but not $a^* = (\emptyset\ a^{**})$, since $(\emptyset\ a^{**}) = \emptyset$

It can be easily proven that $[\![\alpha]\!] = [\![\beta]\!]$ if $\vdash \alpha = \beta$, that is, the system is *correct* with respect to the denotational semantics. The proof proceeds by showing that all axioms are correct, and that the rules allow only to derive correct equations from correct ones. In passing we mention that the system also can be proven to be *complete*, that is, if $[\![\alpha]\!] = [\![\beta]\!]$ then $\vdash \alpha = \beta$. Completeness usually is much harder to show than correctness.

Methods

A formal language is described by an unambiguous syntax and a mathematical semantics. For a Formal *Method* (as opposed to a formal language) it is essential that there are some *algorithms* or *procedures* which describe what can be done with the syntactic objects in practice. According to the Oxford dictionary, a method is a particular procedure for accomplishing or approaching something, especially a systematic or established one. A Formal Method describes how to 'work with the language', that is, perform some activity on its elements in order to achieve certain results. In general, this information processing is some form of transformation (metamorphosis, *Gestaltwandlung*), where the syntactic objects are modified from one form to another.

Usually, a formal language is designed for a specific purpose. For example, a logical language is supposed to formalise human reasoning. A specification language should allow to describe the functionality of a system. A program formulated in a programming language should be executable on a machine.

A model expressed in some modelling language should help humans to understand a concept or design.

The methods associated with a formal language usually are constructed to support this purpose. For a logical language, the transformation can be a calculus with which to derive theorems from axioms. For a specification language, it can be a set of rules to transform a specification into an implementation. For programs written in any programming language, the execution on a virtual machine can be seen as a form of transformation. For modelling languages, model transformations allow to change between different levels of abstraction.

In a Formal Method, the transformation must be according to fixed rules; these rules operate on syntactical objects of the formal language under discussion, and result in some 'insight' about them. Such an insight might be the result of the transformation, or the realisation that the (repeated) transformation does not come to an end. Other insights we might want to achieve are whether a program is correct with respect to its specification, or whether one model refines another one.

Continuing our example, we show how regular expressions can be used in everyday text processing.

Example 2.7: Regular Replacements

A frequent task while writing scientific articles is to consistently replace certain text passages by others in the whole text. A replacement $[\alpha := \beta]$ consists of a regular expression α and a word β over $\mathcal{A}$. The word δ is the result of the replacement $[\alpha := \beta]$ on a word γ, denoted as $\delta = \gamma[\alpha := \beta]$ if one of the following holds:

1. either there exist γ_1, γ_2 and γ_3 such that
   - 1.1. $\gamma = \gamma_1\gamma_2\gamma_3$,
   - 1.2. $\gamma_2 \in [\![\alpha]\!] \setminus [\![\varepsilon]\!]$,
   - 1.3. γ_1 is of minimal length, that is, there are no γ_1', γ_2' and γ_3' such that $\gamma = \gamma_1'\gamma_2'\gamma_3'$, $\gamma_2' \in [\![\alpha]\!] \setminus [\![\varepsilon]\!]$ and $|\gamma_1'| < |\gamma_1|$,
   - 1.4. γ_2 is of maximal length, that is, there are no γ_2' and γ_3' such that $\gamma = \gamma_1\gamma_2'\gamma_3'$, $\gamma_2' \in [\![\alpha]\!] \setminus [\![\varepsilon]\!]$ and $|\gamma_2'| > |\gamma_2|$,

   and $\delta = \gamma_1\beta(\gamma_3[\alpha := \beta])$, or
2. there are no γ_1, γ_2 and γ_3 satisfying the above 1.1–1.4., and $\delta = \gamma$.

The definition of the first case is recursive; it is well-defined because condition 2. requires that γ_2 is a nonempty string. Therefore, $|\gamma_3| < |\gamma|$, and the recursion must terminate.

As an application of regular replacement, we note that $(p \Rightarrow (q \Rightarrow p))[(p + q) := (p \Rightarrow q)] = ((p \Rightarrow q) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow q)))$.

Coming back to our introductory tool experiments in Example 2.1 on page 6, the above definition determines:

- if the wildcard sign '*' has been defined to express the iteration of the empty word, then $(abc)[\varepsilon^* := x] = abc$ as condition 1.2 of Example 2.7 can not be fulfilled;
- if the wildcard sign '*' stands for 'any word', then $(abc)[* := x] = x$, because the wildcard sign matches exactly 'abc' and therefore condition 1 of Example 2.7 is fulfilled with $\gamma_1 = \varepsilon$, $\gamma_2 =' abc'$, and $\gamma_3 = \varepsilon$.

Thus, the formal treatment allows to calculate a reliable result which is independent from the particular text editor being used, and against which the tools can be verified.

We now have discussed all ingredients of what constitutes a Formal Method, and thus are in a position to give a definition.

Definition 1. A *Formal Method* $\mathbb{M}$ consists of three components:

- syntax,
- semantics, and
- method.

The syntax gives a precise description of the form of objects (strings or graphs) belonging to $\mathbb{M}$. The semantics describes the 'meaning' of the syntactic objects of $\mathbb{M}$, in general by a mapping into some mathematical structure. The method describes algorithmic ways of transforming syntactic objects, in order to gain some insight about them.

1.2 Formal Methods in Software Development

Having developed an understanding of what Formal Methods are, we now consider their role in software development. To this end, we briefly recall the notion of the software life cycle, describe how this cycle is realised, and discuss where to use Formal Methods in the life cycles. While life cycle models describe development activities and their order, software development standards give a legal framework prescribing which activities have to be performed, including Formal Methods. This leads to a discussion of the main purposes for the use of Formal Methods in systems development.

1.2.1 The Software Life Cycle

A *software life cycle (SLC)* (also referred to as "software development life cycle", "software development process", and "software process") is a structure describing development, deployment, maintenance, and dismantling of a software product. There are several models for such processes, describing

different approaches on how to develop software (waterfall, spiral, iterative, agile development, etc.).

Our objective is to discuss the use of Formal Methods in the software development process, independently of the model used, rather than to provide a survey on such different models. For that reason we concentrate on the *V-model*, and the general ideas behind *agile* methodologies.

The V-model

Models of software development often describe the development process as being composed of separate phases. For example, there usually are a project definition phase, an architectural and software design phase, a coding phase and a testing phase. Traditionally, these phases are ordered sequentially, which leads to the so-called waterfall model. In this model, the results of one phase are starting points for the subsequent phase, like water falls from one level to the next in a cascade. The waterfall model has several deficits and today is considered to be archaic. Mainly, it does not pay respect to the fact that quality assurance takes place on several levels. For instance, system testing is considered with the systems specification, whereas in unit testing individual units (methods, procedures, functions etc.) rather than the overall system are checked.

Traditionally, the V-model usually has been depicted like the waterfall model, however in the shape of a big V. The V-model has been developed over many years in various versions. A newer version is the V-model XT (for "eXtreme Tailoring") [dBfI12], see Fig. 1.2. In particular, the German federate office for information security (BSI) was a driving force in its elaboration. In Germany, the V-model is mandatory for safety-critical governmental projects. Instead of describing phases, the V-model XT describes states in the development process. It refrains from prescribing a specific order to these states.

Fig. 1.3 hints at where and how validation and verification could be used in the software development process according to the V-model XT. It shows four design levels (from top to bottom: requirements, design specification, architecture, and implementation). At each level, the realisation and integration artefacts (on the right) should comply with the corresponding specification and subdivision artefacts (on the left).

Many researchers and practitioners in software engineering differentiate between 'validation' and 'verification' in the following way: *validation* tries to answer the question

"Are we building the right product?",

whereas *verification* is concerned with the question

"Are we building the product right?"

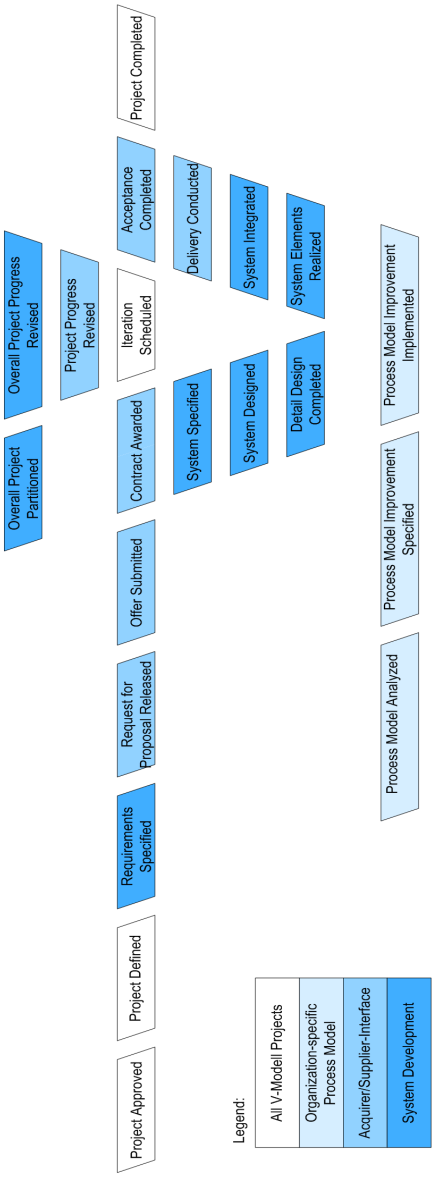

Fig. 1.2 V-model XT [dBfI12].

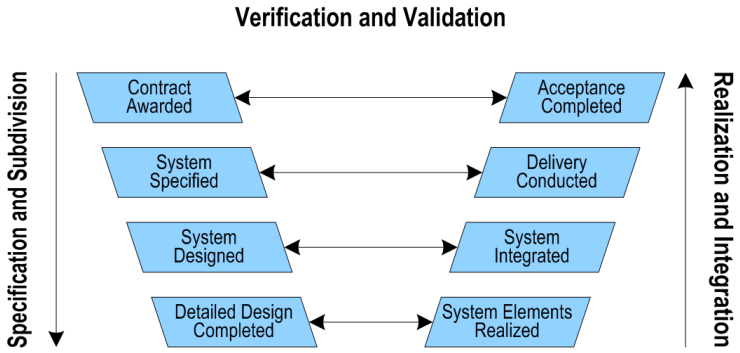

Fig. 1.3 Verification and Validation in the V-model XT [dBfI12].

Thus, validation refers to the user's needs according to the requests, while verification checks that the specification is correctly implemented. In Fig. 1.3 this means that validation could be associated with the compliance between 'contract awarded' and 'acceptance completed', whereas verification concerns the other compliances.

Note, however, that many researchers use a slightly different definition, where *validation* is a general term covering all activities for checking that the product is correct, while *verification* is used for the process of formally proving that a program meets its specification. Since such a formal proof requires formal languages, verification is only applicable at the three lower levels of Fig. 1.3. This can be for establishing the horizontal compliances, for refinement between different levels, or for proving properties about formal artefacts.

Agile Development

Phase-based models of software development, such as the waterfall model, have been criticised for a number of reasons. A main point is that each new phase has to wait until the previous one is completed. This can lead to delays in the project. Moreover, if an error is detected it might be necessary to go back to an earlier phase, causing further delays. Finally, the waterfall model assumes that all system requirements are known from the very beginning of the development. There is no provision to modify or extend requirements during the process. This can be a severe restriction.

Therefore, many other process models have been proposed. A current trend is to develop software in a manner where teams are small, the phases are not

clearly identified, and the user is represented in the whole process of software development. This procedure has been called *agile development* [Coc00]. Design, development and testing are done almost simultaneously and in short iterations.

In an agile development process, a natural way to work is to follow a test-driven development approach [Bec02]. That is, before starting to write code, tests for the system are produced. These tests represent user scenarios and requirements for the system. As long as the code is non-existent or erroneous, the tests will fail. Then the code is written in order to make the tests pass. When all tests pass, one system development cycle is completed.

The Scrum management methodology identifies roles and responsibilities in an agile development process [SB01]. It also defines activities like daily and weekly meetings, where the basic unit of development is organised in project time slots (*sprints*). Each time slot should produce a potentially deliverable result (e.g., a piece of software).

In an agile process, roles (manager, analyst, programmer, tester, verifier etc.) are frequently swapped amongst group members. Therefore, each developer should in principle have knowledge of all development activities in the project. In particular, if Formal Methods are used, the group members should know the capabilities and limitations of the available formal development tools.

1.2.2 Formal Methods: When and Where

While process models describe development phases and their order, software development standards give a legal framework prescribing which activities have to be performed. For example, a standard might prescribe that "for each phase of the overall ...system and software safety lifecycles, a plan for the verification shall be established concurrently with the development for the phase. The verification plan shall document or refer to the criteria, techniques, tools to be used in the verification activities." [IEC10].

There are various standards on the development of software, e.g., EN-50128, IEC-61508 and DO-178 B/C. Some of these standards prescribe that Formal Methods are being used for high integrity levels. For example, IEC 61508, the international standard on functional safety of electrical/electronic/programmable electronic safety-related systems, is a 'meta-standard', from which several other domain-specific standards are derived. In Part 7 (2010), section C.2.4.1 it defines the aim of Formal Methods as "the development of software in a way that is based on mathematics. This includes formal design and formal coding techniques. ... A Formal Method will generally offer a notation (generally some form of discrete mathematics being used), a technique for deriving a description in that notation, and various forms of analysis for checking a description for different correctness properties." In

Section B.2.2. it states that "Formal Methods ... increase the completeness, consistency or correctness of a specification or implementation".

However, the IEC 61508 standard also states that there can be disadvantages of Formal Methods, namely: "fixed level of abstraction; limitations to capture all functionality that is relevant at the given stage; difficulty that implementation engineers have to understand the model; high efforts to develop, analyse and maintain model over the lifecycle of system; availability of efficient tools which support the building and analysis of model; availability of staff capable to develop and analyse model."

Several Formal Methods are described in the standard (CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z). The use of Formal Methods is recommended for achieving the highest safety integrity level (SIL 4), where the average frequency of a dangerous failure of the safety function must be provably less than 10^{-8}/h, i.e., a failure may occur on average at most once in 10,000 years of operation.

DO-333 is the Formal Methods supplement to DO-178C and DO-278A for safety-critical avionics software. It defines Formal Methods as "mathematically based techniques for the specification, development and verification of software aspects of digital systems". Formal Methods can be used to "improve requirements, reduce error introduction, improve error detection, and reduce effort". The supplement further states that "the extent to which Formal Methods are used [in the software development] can vary according to aspects such as preferences of the program management, choice of technologies, and availability of specialised resources".

Use of Formal Methods

The use of Formal Methods in software development is not constrained to a specific process and life cycle model followed by a company. That is, Formal Methods can be used with traditional as well as agile models. Moreover, Formal Methods should not constitute separate phases or sprints, but should rather be integrated as part of the general verification activities.

Mishra and Schlingloff [MS08] evaluate the compliance of Formal Methods with process areas identified in CMMI-DEV, the capability maturity model integration for development. Their result is that out of 22 process areas from CMMI, six can be satisfied fully or largely with a formal specification-based development approach. Notably, the process areas requirements management, product integration, requirements development, technical solutions, validation and verification are supported to a large extent. They also show the possibility of automation in process compliance, which reduces the effort for the implementation of a process model.

Formal Methods are used in system development for two main purposes:

1. as a means to make descriptions precise, and
2. to help in different kinds of analysis.

Concerning the first purpose, descriptions of interest include requirements, specifications, and models, which appear at different levels and moments in the life cycle. It is common practice to write software descriptions in natural language. In spite of the apparent advantage of being written in a language understandable to everybody, its inherent ambiguity and lack of precision makes the realisation of such descriptions problematic.

The literature distinguishes between linguistic and domain-specific ambiguities. Kamsties et al. [KBP+01] provide the following examples: the 500 most used words in English have on average 23 meanings; the sentence "The product shall show the weather for the next 24 hours" exhibits the linguistic ambiguity if the phrase 'for the next twenty-four hours' is attached to the verb 'show' or to the noun 'weather'; the sentence "Shut off the pumps if the water level remains above 100 meters for more than 4 seconds" is ambiguous as in the given domain the term 'water level' can refer to the mean, the median, the root mean square, or the minimum water level. An attempt to address such problems is the use of controlled natural language. Here, the grammar and vocabulary of natural language is restricted in order to avoid or reduce ambiguity of sentences. Present day controlled languages, however, are often felt to be either too restrictive or too informal to be practical. In this book we advocate formal languages, which have a well-defined syntax and semantics. They may be used to resolve such ambiguities and to achieve the required level of precision.

To illustrate such a process of removing ambiguities, consider the regular replacements discussed in Example 2 on page 5. We showed that the informal description is ambiguous and can lead to unexpected results. In contrast, in Example 2.7 on page 11 we formally defined $\gamma[\alpha := \beta]$ to be the word resulting from the word γ by the replacement of a regular expression α with the word β. The formal definition cares for all special cases; there is no need for explaining what, e.g., the replacement of the empty language by the empty string in a one-letter word is. $a[\emptyset := \varepsilon]$ has a well-defined meaning which can be derived from the definition.

This book focuses largely on using Formal Methods in the second way, i.e., to assist with analysis. The use of Formal Methods in Verification and Validation (often abbreviated as V&V) is wide and includes techniques such as static analysis, formal testing, model checking, runtime verification, and theorem proving. All the above are complementary techniques to standard methods such as code review, testing and debugging. They increase the confidence in the correctness of the software under consideration.

This is shown in the software development for the ISS as described in Example 1. It illustrates how Formal Methods can help to analyse a system. Given a system model, the model checker FDR could prove the presence of several deadlocks and livelocks. This helped to improve the quality of the safety-critical system.

Aligned with current practices in software development, Formal Methods may be used from the very beginning (when a system is initially conceived) up

to the end (when the final product is shipped). Model checking, for instance, does not require that a single line of code has been written: it may already be used when the first formal specifications and high-level models are available. As another example, runtime verification can be used in a pre-deployment phase, when (part of) the code is ready to run, and even after deployment to control/enforce desirable properties at runtime.

1.2.3 A Classification Scheme for Formal Methods

In Def. 1 on page 12, we said that a Formal Method consists of syntax, semantics and specific methods or algorithms. Thus, e.g., "CTL model checking" or "Z theorem proving with HOL" are particular Formal Methods.

Although there has been quite a debate in the Formal Methods community on the 'right' syntax, the 'best' semantics and the 'most effective' algorithms, these aspects can be subsumed within other categories in a taxonomy of Formal Methods.

In order to give an orientation, we provide a classification scheme which allows to categorise each Formal Method along the following dimensions.

- **Method definition** – syntax, semantics and procedures as described above.
  - Syntactic aspects, e.g., whether the language allows user-defined mixfix operators, linear or non-linear visibility, graphical or textual notation, etc., are related to the usability-aspect of a Formal Method.
  - Semantic aspects – which semantic domains are employed and how they are characterised (denotational, operational, axiomatic semantics) – determine the application range and underlying technology.
  - Algorithmic aspects (describing what can be done with the method) dominate the underlying technology and properties of concern. Typical procedures include simulation and symbolic execution, model checking, automated or interactive theorem proving, static analysis, refinement checking, etc.
- **Application range** – this dimension determines the application domain (e.g., avionics, railway, finance) and the specific needs of this domain (whether the systems are mainly reactive, interactive, real time, spatial, mobile, service oriented, etc.)
- **Underlying technology** – this dimension notes how the method can be realised. Technologies are, for example, SAT solving, logical resolution, term rewriting, symbolic representation, etc.
- **Properties of concern** – This dimension categorises properties of the systems which are the subject of the Formal Method and which are supported

by the method (safety, liveness, fairness, security, consistency, functional correctness, etc.)
- **Maturity and applicability** – this dimension describes how fit the method is for actual use (universality, expressivity, usability, learning-curve, intuitive level, tool support, etc.)

Each particular application of Formal Methods can be located within the space that these dimensions span. For illustration, consider Example 1 on page 2: the language used in this example is the process algebra CSP, with its failures semantics, and automated refinement checking as a procedure (see Fig. 1.1). The application domain is that of fault-tolerant algorithms in aerospace. The technology used in the FDR tool is the hierarchical compression of the state space, a technique specific for this tool. Properties of concern are livelock and deadlock. The case study was conducted in a collaboration between industry and academia, since the abstraction process from Occam to CSP and the use of FDR was outside the standard routine of the aeronautic engineers.

The second example from this chapter, regular replacement, can be classified as follows: the syntax is the language of regular expressions, with the usual denotational (set-theoretic) semantics, and text transformation as a procedure. Application domain are text editors or macro processors. There is no specific technology involved with this example, as we refrain from giving an implementation; one possibility would be to use list processing in a functional programming language. The property to be achieved is to give a well-defined transformation, open to formal argument about the correctness of any implementation. The Formal Method of regular expressions belongs to the standard knowledge of computer science and is accessible at an undergraduate level.

Of course, there are other dimensions which could be added to this classification scheme. These include specification focussed vs. analysis focussed, correctness-by-construction vs. design-and-proof-methodology, lightweight vs. heavyweight Formal Methods, etc.

1.2.4 Tool Support for Formal Methods

Formal Methods usually start on the 'blackboard': toy examples are treated in an exemplary way. With paper and pen one checks if a method works out. In the long run, however, Formal Methods need tool support in order to become applicable. This is the case as software systems are fundamentally different compared to mathematical theories:

Numbers of axioms involved. In Ch. 4, we will formalise and verify control programs written in Ladder Logic. Here, each line of code is represented by one logic axiom in the specification language CASL. The toy

example presented, a traffic light controller, cf. Example 44, has about 10 lines of code, i.e., the model consists of about 10 CASL axioms. Our approach scales up to actual railway interlockings. Such interlockings describe for a railway station how to position the points and how to light the signals in such a way that trains can move safely. A typical interlocking program written in Ladder Logic consist out of 500–1500 lines of code, i.e., its model has 500–1500 CASL axioms in its formalisation.

In contrast, the whole realm of group theory is based on three axioms only, namely that the operation + is associative (axiom 1), that + has the unit 0 (axiom 2), and, finally, that every element has an inverse (axiom 3).

These example indicate that the number of axioms when applying Formal Methods is by magnitudes larger than the number of axioms involved in mathematical theories. Consequently, tool support is needed in Formal Methods for sheer book keeping.

Ownership and interest. The interlocking program for a railway station is commissioned by a rail operator. Intellectual property rights ensure that, besides the rail operator, only the company writing the code and the railway authorities have access to design documents, code, and verification documentation, etc. These artefacts are studied only when the software lifecycle dictates it:

- during production by the company programming it,
- for acceptance by the company running the train station,
- for approval by the railway authorities, and
- when maintaining the code by a possibly different company hired for the task.

Thus, any verification of a ladder logic program, say in CASL, will be studied only at few occasions.

In contrast, group theory is public, its theorems and their proofs are published in books and journals, everyone has access to them. The proofs of group theory are taught for educational purposes at universities. Every year, the fundamental theorems of group theory are proven and checked in lecture halls all over the world.

Many software systems are the intellectual property of a company. This restricts access to the actual code; interest in their design is limited. Mathematical theories are part of the scientific process and publicly available. There is scientific interest in them. Therefore, proofs related to a specific software system are studied by few people only, and only when necessary – while mathematical proofs are studied by many, over and over again. Consequently, tools play the role of 'proof checkers' for quality control in Formal Methods.

Change of axiomatic basis. Every ten to fifteen years, the design of a railway station changes. New safety regulations have to be implemented, the station shall deal with more trains, new technology shall be introduced

such as the European Train Control System (ETCS). This requires changes to the interlocking program and, consequently, to the proofs on it.
In contrast, mathematical theories are stable. Already in the 1830s Galois worked with the axioms of group theory, which have not changed ever since.
Requirements of software systems are bound to change in small time intervals. This means that design steps involving Formal Methods need to be repeated several times, sometimes already during the design phase, certainly when maintaining the system. Mathematical theories, however, are stable over centuries. Consequently, tools are needed to help with management of change in Formal Methods.

The technology underlying tools for Formal Methods is generic. The Heterogeneous Tool Set HeTS – to be discussed in Ch. 4 "Algebraic Specification in CASL" – for example is a 'broker' which offers, amongst other functionalities, translations from the language CASL to various tools. Yet another example is the Process Analysis Toolkit PAT, which supports reasoning about concurrent and real-time systems. Other tools have been built specifically for one Formal Language. An example is the model checker FDR which has been designed specifically for the process algebra CSP – see Ch. 3 "The process algebra CSP". The current trend in Formal Methods is to offer (integrated) work environments for different Formal Languages and Methods, e.g., HeTS.

Tool Qualification

When software tools are used to validate software, the questions is, who is validating the tools? In other words, for highly safety-critical systems there needs to be evidence why the tools which are used in their development should be trusted. There are two kinds of tools: for artefact generation and for artefact validation. This holds for all artefacts occurring in the software design cycle, e.g., binary code, program text, formal model, specification, or even the user manual. In industry, there are contradicting views concerning the importance of these tool classes. In some areas, generating tools are considered to be more critical than validating tools, since they directly affect the behaviour of the system. In other areas it is argued that faulty behaviour of a generator can anyway be found by the validation tools, which therefore are more critical.

If a generating tool is faulty, then the generated artefact will not correspond to its source. In the case of executable code, e.g., this may mean that the runtime behaviour is not as expected. In the case of a model transformation, the generated model might miss out on properties already established for the source model.

Example 3: Public-Domain C Compilers

Yang et al. [YCER11] found more than 325 errors in public-domain C compilers using a specialised compiler testing tool. They report:

> "Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input."
>
> "A version of GCC miscompiled this function:

```
1   int x = 4;
2   int y;
3
4   void foo (void) {
5       for (y = 1; y < 8; y += 7) {
6          int *p = &y;
7          *p = x;
8       }
9   }
```

> When `foo` returns, `y` should be `11`. A loop-optimisation pass determined that a temporary variable representing `*p` was invariant with value `x+7` and hoisted it in front of the loop, while retaining a dataflow fact indicating that `x+7==y+7`, a relationship that no longer held after code motion. This incorrect fact led GCC to generate code leaving `8` in `y`, instead of `11`."

If a validation tool is inaccurate or faulty, there are two cases: the tool might report an error where there is none (false positive), or the tool might miss to report an error where there is one (false negative). For example, an erroneous program verifier might fail to verify a correct program (false positive), or it might claim to have found a proof for an incorrect program (false negative). Often, false negatives are more critical than false positives since they convey a deceptive certainty. False positives are a hassle, because they need to be dealt with manually.

To make this more concrete, consider the tool Cppcheck for static analysis of C programs:

Example 4: Static Analysis of C Programs

When Cppcheck (Version 1.59) checks the following program, it issues for line 5 the error message

```
Array 'x[7]' accessed at index 13, which is out of bounds.
```

```
1   int main() {
2     int x[7];
3     int i = 13;
```

```
4     int flag; if (i<7) flag = 1; else flag = 0;
5     if (flag) x[i] = 0;
6     x[3] = 33; x[x[3]] = 0;
7   }
```

Since the assignment in line 5 is never executed, this is a false positive. Surprisingly, this false positive disappears when we replace line 4 by the equivalent

```
4     int flag = (i<7)?1:0;
```

Cppcheck does not issue an out-of-bounds warning for line 6. This is a false negative, since the assignment `x[33] = 0;` clearly might cause problems.

What are now the possibilities for the validation of tools? The usual approach is to resort on tools which are *proven-in-use*, i.e., where experience from many previous projects suggests that the tool is 'correct'. This is especially the case for certain public-domain tools which have been applied by many users for a long period of time. In order to claim that a tool is proven-in-use, it is necessary to provide evidence in which comparable projects it was used. As the above example of the GCC compiler error shows, proven-in-use is no guarantee against subtle, hidden errors.

For new or newly introduced methods and tools, the proven-in-use principle poses the problem of how to begin such a chain of trust. So, what to do when proven-is-use is not applicable? In order to be allowed to use tools which are not proven-in-use in a safety-oriented development, at least one has to perform a *tool qualification*. That is, the tool has to be applied to a number of selected examples under controlled conditions, where the tool's behaviour must be analysed and documented in detail. *Tool certification* is the process of confirming the qualification by a designated authority. Usually this is done only for tools to be applied in several different projects. The software development standard DO-333, e.g., prescribes in detail how to qualify and certify tools for different safety integrity levels in aerospace.

Of course there is still a possibility that even certified tools might contain errors. There are further methods that can improve the reliability of Formal Methods tools.

In the case of generating tools, one possibility is to verify the generator itself. For instance, there are a number of research projects and results dealing with compiler verification. One challenge here is that the correctness argument needs to deal with several different languages: the language the generator is written in, the source language, and the target language. Due to the sheer size of the problem, often compiler verification is supported by automated tools. Yet another possibility is to generate, besides the code, also certain proof conditions from program annotations, which can be checked automatically in the generated code with a suitable verification tool. This

way, if the compiler is faulty in its code generation part, this will be detected by the following verification tool.

Both these suggestions to improve the dependability of generating tools rely on the existence of correct verification tools. In order to increase the trust in the correctness of verification tools themselves, one can run several different provers on the same problem and compare their results. If at least two of them agree, then, under the assumption that different tools do not make the same mistake, the result is as given. If one prover claims to have found a proof, while another one claims that the property is not valid, one of them must be faulty.

Another approach to increase the trust in theorem provers is to augment them with a proof checking component. For this, the prover must not only give a boolean result but also produce some term which allows to check whether the proof is a valid one.

1.3 Formal Methods in Practice

We present various case studies on the application of Formal Methods. These come in two flavours: comparative case studies compiled by academics, and the application of Formal Methods in industry.

1.3.1 Comparative Surveys and Case Studies

The Formal Methods community has compiled several surveys with the aim of comparing different approaches for the application of Formal Methods in various areas. The characteristic of these surveys is to discuss *one* coherent example in the context of *several* Formal Methods.

- Lewerentz and Lindner [LL95] discuss a production cell. This cell is considered to be safety critical, i.e., a number of properties must be enforced in order to avoid injuries of people. The system is reactive, i.e., it has to react permanently to changes of the environment. In principle this is a real-time problem, as the reaction of the control software must be guaranteed within a certain interval of time.

Example 5: Production Cell

"The production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a travelling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There,

they are forged and then brought out of the cell via the other belt and the crane." [LL95].

This case study reflects a typical scenario as it arises in industrial automation. For this case study, various safety and liveness requirements are to be established. Efficiency (w.r.t. production time) and flexibility (w.r.t. the effort it takes to adapt a solution to changed requirements) should also be taken into account. Besides presenting 18 contributions, the book includes a summary and evaluation of the different solutions.

- Broy et al. [BMS96] study a memory cell which can be accessed by remote procedure calls. Such a call is an indivisible action. A return is an atomic action issued in response to a call. There are two kind of returns, *normal* and *exceptional*. A return is issued only in response to a call.

Example 6: RPC Memory Cell

"The component to be specified is a memory that maintains the contents of a set **MemLocs** of locations. The content of a location is an element of a set **MemVals**. This component has two procedures... [Procedure] **Read** returns the value stored in address **loc**. [Procedure] **Write** stores the value **val** in address **loc.** The memory must eventually issue a return for every **Read** and **Write** call." [BMS96].

Broy et al. [BMS96] collect fifteen solutions in various Formal Methods, including Petri nets, temporal and higher-order logics, various forms of transition systems or automata, and stream-based approaches.

- Abrial et al. [ABL96] study a classical control problem:

Example 7: Steam-Boiler Controller

"[The steam-boiler control program] serves to control the level of water in a steam-boiler. The program communicates with the physical units through messages which are transmitted over a number of dedicated lines connecting each physical unit with the control unit. ... The program follows a cycle and a priori does not terminate. This cycle takes place each five seconds and consists of the following actions:
– Reception of message coming from the physical units.
– Analysis of informations which have been received.
– Transmission of messages to the physical units." [Abr94].

Abrial et al. [ABL96], use Formal Methods for various purposes: formal requirement specifications, intermediate refined models, analysis of system properties, proofs, automated synthesis of conditions implying safety for parameters of the controller, design or generation of executable code. The overall twenty-one contributions used algebraic, logical, and operational languages similar to those treated in the subsequent chapters of this book.

- Frappier and Habrias [FH01] discuss a classical commercial software application:

Example 8: Invoicing Software

"To invoice is to change the state of an order (to change it from the state "pending" to "invoiced"). On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders. The same reference can be ordered on several different orders. The state of the order will be changed into "invoiced" if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product." [FH01].

The volume collects specifications in the state-based methods Z and B, in the event-based methods Action Systems, UML with a behaviour-driven method, VHDL, Estelle, SDL, and E-Lotos, and in other formal approaches as CASL, Coq, and Petri Nets.

- Jones and Woodcock [JW08] collect approaches to mechanise the proof of correctness of the Mondex smart-card for electronic finance. This was one of the first comparative case studies dealing with security issues.

Example 9: Electronic Purse Mondex

"The system consists of a number of electronic purses that carry financial value, each hosted on a Smartcard. The purses interact with each other via a communication device to exchange value. Once released into the field, each purse is on its own: it has to ensure the security of all its transactions without recourse to a central controller. All security measures have to be implemented on the card, with no real-time external audit logging or monitoring." [SCW00].

The methods applied to the Mondex case study are the Alloy model-finding method, the KIV system, Event-B, UML and OCL, RAISE, and Z/Eves.

- Rausch et al. [RRMP08] document a competition on the so called Common Component Modelling Example (CoCoME). Given a prescribed architecture, the challenge lies in using a specific formalism for modelling and analysing the CoCoME according to this architecture.

Example 10: CoCoME Trading System

The CoCoMe case study concerns a trading system as it can be observed in a supermarket handling sales. At a Cash Desk the Cashier scans the goods the Customer wants to buy and the paying (either by credit card or cash) is executed. The central unit of each Cash Desk is the Cash Desk PC which wires all other components with each other. Also the software which is responsible for handling the sale process and amongst others for the communication with the Bank is running on that machine. A Store itself consists of several Cash Desks organised in Cash Desk Lines. A Cash Desk Line is connected to a Store Server which itself is also connected to a Store Client. A set of Stores is organised as an Enterprise where an Enterprise Server exists to which all Stores are connected. (Formulated closely following Herold et al. [HKW+07].)

Rausch et al. [RRMP08] documents more than ten formal component models and their use in verification and quality prediction. A jury evaluation concludes the volume. We discuss similar examples in Ch. 6 on "Specification and Verification of Electronic Contracts".

- Cortier and Kremer [CK11] collect several popular symbolic approaches to formal security. Different research groups demonstrate their techniques, taking a flawed public key protocol (and its correction) as a common example:

Example 11: Handshake Security Protocol

"The aim of the protocol is that A and B share a secret key s at the end. Participant A generates a fresh session key k, signs it with his secret key $sk(A)$ and encrypts it using B's public key $pk(B)$. Upon receipt B decrypts this message using the private secret key, verifies the digital signature and extracts the session key k. B uses this key to symmetrically encrypt the secret s." [CK11].

This protocol shall provide secrecy: the secret s shall only be known to A and B. The above handshake protocol, however, is vulnerable to a 'man in the middle' attack. Various techniques are used to (automatically) find this flaw and to prove that adding the identities of the intended participant

changes the protocol into a correct one. These techniques include rewrite rules, Horn clauses, strand spaces, constraint systems, process algebra, and Floyd-Hoare style logics. Further protocols are consider in order to demonstrate the strengths of individual techniques. In Chapter 8 on the "Formal Verification of Security Protocols" we verify a protocol for authentication.

For good reason, the above compilations refrain from giving a concrete recommendation which method is 'the best'. Similar to the problem of selecting a suitable programming language for a particular project, the suitability of a method strongly depends on the context. For example, a method for analysing functional correctness might not be suitable for deadlock analysis of protocols.

1.3.2 Industrial Practice

Formal Methods play an increasing role in industrial practice: "yesterday's Formal Methods are today's best practice". For example, the theory of static program analysis and abstract interpretation has been developed since the mid-1970's. A first tool, Lint, for checking source code of C programs has been released in 1979. Subsequently, more specialised tools for safety-critical applications based on this theory were developed. Today, more than one hundred tools for static code analysis exist, with varying strengths and application ranges. However, static analysis is also performed in ordinary compilers: the Java specification (for version 7 in section 14.21.) requires that "it is a compile-time error if a statement cannot be executed because it is unreachable" [GJBB13]. There are detailed instructions in the language definition on how to figure out whether a statement is reachable; in general, this is a static analysis task which is performed by the Java compiler.

In hardware design, modelling a chip lay out and checking it with model checking and theorem proving (both techniques were developed during the 1990s) is an established practice today. Graph grammars originated in the late 1960s; the theory of graph transformation provides the mathematical foundation for code generators in current model-based development environments. Existing standards, such as DO-333 (see Sect. 1.2.2), which was released in 2012, allow to replace informal validation steps such as code inspections, code reviews, and testing by Formal Methods.

Knowledge transfer from academia, however, is a slow process in general. The Formal Methods community itself reflects on this topic. We document this reflection within the last twenty years. In 1990, Hall [Hal90] identifies "Seven Myths of Formal Methods": he argues that unrealistic expectations can lead to the rejection of Formal Methods in industry, and presents a more realistic picture of what Formal Methods can achieve. Five years later, Bowen and Hinchey [BH95a] formulate "Seven more myths of Formal Methods", identify them as misconceptions, and conclude that "Formal Methods

are not a panacea, but one approach among many that can help to improve system reliability". Complementing this work, Bowen et al. [BH95b] formulate "Ten Commandments of Formal Methods" which give guidelines of how to make good use of Formal Methods. Ten years later, Bowen and Hinchey [BH06] observe that the "application of formal methods has been slower than hoped by many in the Formal Methods community" and conclude that, at least for highly safety-critical systems, Formal Methods have found a niche. Over the years, the perception of Formal Methods has become more positive. In 2011, e.g., Barnes states that "the application of Formal Methods is a cost effective route to the development of high integrity software" [Bar11]. The 2020 white paper "Rooting Formal Methods within Higher Education Curricula for Computer Science and Software Engineering" [CRD+20] argues:

- Current software engineering practices fail to deliver dependable software.
- Formal Methods are capable of improving this situation, and are beneficial and cost-effective for mainstream software development.
- Education in Formal Methods is key to progress things.
- Education in Formal Methods needs to be transformed.

The "2020 Expert Survey on Formal Methods" [GBP20] compiles a collective vision on the past, present, and future of FMs with respect to research, industry, and education. They report:"A huge majority of 90% thinks the use of Formal Methods will likely become more widespread in industry, while only nine experts [out of 130] doubt this and four have no opinion."

Success Stories

In order to support positive views on Formal Methods, we report on a number of industrial experiments and experiences.

Example 12: Model Checking at Intel

In 1993, Intel released the first Pentium® processor. Shortly afterwards, a bug in the floating point arithmetic was detected, which caused wrong computation results with certain division operations. As a consequence, Intel had to exchange more than one million faulty processors, with cost of more than 475 million dollars. Subsequently, Intel initiated major changes in its validation technology and methodology. "Since 1995 Intel engineers have been using formal verification tools to verify properties of hardware designs" [Fix08]. In hardware design, bugs have traditionally been detected by extensive testing, including pre-silicon simulation. However, this procedure is rather slow, and there are too many input combinations for an exhaustive testing. Therefore, Intel now employs temporal logic model checking for this task. Here, a model of the system (i.e., the hardware design) is compared with a formal specification

of system properties (see Ch. 2 on Logics). For describing hardware properties, Intel developed the specification language ForSpec, which was later made into the IEEE 1850 standard PSL (property specification language). In this language, properties of floating point arithmetic as required by the relevant IEEE 754 standard were formulated. As a model, the register transfer level description of the design is used. Thus, the verification is done with the same gate-level design that is used for traditional dynamic validation. Given suitable model checking tools, the verification is fast and can be easily done within the development timeframe. Therefore, such a full formal verification of floating-point processing units is now standard practice at Intel, see also the work of Harrison [Har03b].

Regarding this case study, L. Fix [Fix08] of Intel remarks:

> The barrier to moving from a limited deployment to wide spread deployment of formal property verification in Intel was crossed mainly due to two developments: the first was the introduction of ForSpec assertions inside the Verilog code, thus allowing the designers to easily code and maintain the properties (assertions). The second was the integration of the formal verification activity with other validation efforts. In particular, the RTL designer had two reasons to annotate his/her code with assertions. The assertions were always checked during simulation and in addition the assertions served as assumptions and properties for formal verification. In case an assertion was too complex to be verified formally it was still very useful as a checker in simulation.

Example 13: Microsoft's Protocol Documentation Program

Due to legal negotiation with the U.S. Department of Justice and the EU, Microsoft decided to make available to competitors the interfaces of certain client-server and server-server communication protocols used in the Windows operating system. In order not to disclose the source code of the implementation, a series of technical documents were written describing the relevant protocols. This documentation was quite extensive, consisting of more than 250 documents with approximately 30,000 pages in all. The actual implementation of the protocols had previously been released on millions of servers, as part of the Windows operating system. To ensure that the informal specifications conform to the actual code, in the Winterop project a formal model of the specification was produced [GKB11]. For this, the specification language Spec# was used, which is based on the notion of abstract state machine, with C# syntax. The effort took more than 50 person-years to complete. From these specifications, test cases were automatically generated by the tool SpecExplorer. These test cases could be executed with the existing implementation, exposing over 10,000 "Technical Document Issues" in the specification [GKSB10]. The endeavour was such a big success, that SpecExplorer was turned into a product which is now distributed as

a 'power-tool' add-on to the software development environment Visual Studio.

Several other formal specification and verification project within Microsoft have been done. Hackett et al. [HLQB] use the modular checker HAVOC to check properties about the synchronisation protocol of a core Microsoft Windows component in the NT file system with more than 300,000 lines of code and 1500 procedures. The effort found 45 serious bugs (out of 125 warnings) in the component, with modest annotation effort.

Das [Das06] writes on Formal Methods at Microsoft:

> "Today, formal specifications are a mandated part of the software development process in the largest Microsoft product groups. Millions of specifications have been added, and tens of thousands of bugs have been exposed and fixed in future versions of products under development. In addition, Windows public interfaces are formally specified and the Visual Studio compiler understands and enforces these specifications, meaning that programmers anywhere can now use formal specifications to make their software more robust."

Example 14: Electronic Voting

Secure communications have become fundamental to modern life for the purposes of electronic commerce, online banking and privacy over the Internet to name but a few applications. As a design problem, security protocols have inspired the use of Formal Methods for well over two decades. The distributed and parallel nature of communications facilitated by protocols, along with various assurances desired, means that designing secure message exchange is not straightforward. A good example of this problem is electronic voting, which has a complex set of security and privacy requirements all of which must be guaranteed if digital democracy is to be truly realised in the modern world.

An electronic voting system subject to formal scrutiny is the Prêt à Voter system [RBH+09], which is essentially a multi-party cryptographic protocol offering privacy, voter verifiability, coercion-resistance, and receipt-freeness. Some of these properties have been subject to formal examination, using various methods including process algebra and refinement checks [HS12], and zero-knowledge proofs [KRT13], with the ultimate goal to providing a formal proof of the relevant property.

An implementation of the Prêt à Voter system has been demonstrated for the state of Victoria in Australia [BCH+12].

Undoubtedly some legal [DHR+12] and usability [SSC+12, SLC+11] challenges exist for such electronic voting systems. However, the above case study demonstrates considerable progress for providing assurances to the government and public to ensure confidence and trust in the election system.

Example 15: The Operating Systems seL4 and PikeOS

Formal verification of operating systems remains a difficult task to achieve given the scale and complexity of the software involved. One such attempt stands out to provide a benchmark of how Formal Methods have been effectively applied towards achieving such a goal. The L4 family of microkernels [Lie96] for embedded systems serves as an operating system with typical features of concurrency in terms of threading and inter-process communication, virtual memory, interrupts and process authorisation features.

A secured version of such an operating system, known as seL4 [KAE+10], has been established through formal specification and verification. Formal Methods have been applied at various levels of the development of seL4. Starting with an abstract specification a prototype is generated in Haskell [KDE09], which is a functional and executable language. This has the advantage of translating all data structure and implementation details desired for the final implementation. The Haskell prototype is formalised using Isabelle/HOL, an interactive-theorem prover allowing for machine-checking of proofs, and functional correctness is demonstrated using refinement. A C implementation is manually achieved from Haskell with a view to optimising the code for better performance. The implementation is then translated into Isabelle (using a formal semantics defined for a subset of C) for checking.

The methods used for seL4 have influenced the verification of PikeOS [BBBB09], which is a commercial microkernel operating system based on L4. Core parts of the embedded hypervisor, and, in particular, the memory separation functionalities, have been formally verified using the VCC verification tool. PikeOS is certified according to various safety standards and is used in several critical real-time applications, e.g., in the integrated modular avionics modules of Airbus planes.

The German Verisoft project [Ver07] demonstrates that with present Formal Methods it is not only possible to verify an operating system, but that the systematic use of computer-aided verification tools is possible throughout all layers of abstractions.

Example 16: Model-Based Design with Certified Code Generation

Lustre is a synchronous data-flow programming language which evolved in the 1980s from academic concepts similar to the ones existing in algebraic specification languages (see Ch. 3) [Hal12]. Its main focus was programming reactive real-time systems such as automatic control and monitoring devices. From the beginning, it had a strict denotational and operational semantics. The formalism was very similar to temporal log-

ics (see Ch. 2) which allowed the language to be used for both writing programs and for expressing program properties. In the mid-1980's, the company Merlin Gerin (now Schneider Electric) in collaboration with researchers from VERIMAG developed an industrial version of Lustre for the development of control command software in nuclear power plants. This version was called SAGA and provided a mixed textual / graphical syntax for the language as well as a simple code generator. In order to further industrialise the tool, the company Verilog took over SAGA, renamed it SCADE (for "Safety Critical Application Environment Development") and adapted it to the needs of Aerospatiale (now part of Airbus). In the aerospace domain, any tool used for the development of a critical equipment must have at least the same quality as the equipment itself. Therefore, the SCADE code generator KCG was qualified according to the highest criticality level A. (In this qualification, it was shown that the development processes for KCG conform to the requirements of the standard; note that this does not amount to a full compiler verification!) Verilog itself was acquired in 1999 by Telelogic, a Swedish telecommunications tool provider (now IBM). In 2001, Esterel Technologies bought SCADE from Telelogic for 1.4 million Euro. It extended SCADE by various additional components, e.g., the tool IMAGE by Thales for the design of the cockpit of the A380 aircraft, as well as formal verification technology, SysML support, and software lifecycle management. In 2012, Esterel Technologies was taken over by Ansys Inc. for the sum of 42 million Euro. Ansys plans to integrate SCADE with its own tool Simplorer for modelling and simulating physical systems.

Today, more than 230 companies in 27 countries use SCADE to develop safety-critical control components. Success stories include the use in the primary flight control system of the Airbus A380, the autopilot for several Eurocopter models, several nuclear power plants as well as the interlocking and train control system of the Eurostar trains between London and Paris.

Example 16 demonstrates that tools for Formal Methods not only significantly contribute to system safety, but also can have a considerable market.

Example 17: Transportation Systems in France

The Paris Métro line 14 (Est-Ouest Rapide) was opened in 1998. It is the fastest and most modern line in the Paris subway network, being operated driverless, with a high train speed and frequency. For ensuring the correctness of the control and signalling software, it was decided to use the B method and the associated Atelier B programming tool.

The B method is based on the idea of refinement of abstract machines. Mathematical specifications written in first-order logic are stepwise re-

fined, and it is proven that the refinement is coherent and includes all the properties of the abstract machine. Throughout all of the development steps the same specification language (B notation) is used. The process of refinement is repeated until a deterministic, fully concrete implementation is reached, which is then automatically translated into Ada code.

In the above mentioned Météor project, over 110,000 lines of B specifications were written, generating 86,000 lines of safety-critical Ada code. With this model, 29,000 proofs were conducted. No bugs were detected after the proofs, neither during the functional validation of the software, during its integration in the train, during the on-site tests, nor since the metro lines operate. The software is still operated in version 1.0 today, without any bug detected so far [LSGP07].

Other uses of the B method include the automatic train protection system for the French railway company SNCF, which was installed on 6,000 trains since 1993. For the verification, 60,000 lines of B specifications and approximately 10,000 proofs have been written. In the Roissy VAL project, an automatic pilot for a driverless shuttle in the Paris-Roissy airport has been developed and verified with 180,000 lines of B specification and 43,000 proofs.

In the report [BA05] of the Roissy VAL project mentioned in Example 17 the authors conclude:

> The process described here is suitable for any industrial domains, not only for railways command/control software. Actually this process deals with designing procedural software based on logical treatments, not based on real or floating-point numbers. It is all the more suitable that software specification can be easily formalised into set-theoretical expressions.
>
> From the management point of view, the project went off according to the initial schedule, although the software produced is quite large, thanks to a straightforward process and efficient tools.
>
> Every verification stage throughout the process was useful and led to early error detection: analysis of software document specification, type checking, inspections, proof of abstract model safety properties, refinement proof of correct implementation.

Section 1.3.4 of Garavel's report [GG13] provides a collection of further success stories.

1.3.3 How to Get Started

The previous sections give the right impression that the variety of Formal Methods is overwhelming. This might leave the beginner or potential user to be lost in the field. Which method shall be selected in a given context? We discuss this question in two different scenarios. One possible scenario is that

of a research student learning a Formal Method. The other scenario is that of a Formal Method to be selected in a specific industrial project.

Learning a Formal Method

For the first scenario, this book provides a good starting point. It offers a non-representative selection of methods, where each chapter provides a solid introduction to one method. Specialisation in one method is unproblematic, as the foundations of Formal Methods are well connected. Concepts studied, say, in the context of process algebra, are also to be found in temporal logics, which again are closely connected to automata theory, and are applied, e.g., in testing. Within a discipline, there are often attempts to unify structural insights. In logics, for example, the theory of institutions provides a general framework in which logical properties can be studied in a uniform way (see Ch. 2). The methodological approach to different Formal Methods often is comparable. Consequently, one should not be afraid of intensively studying one specific method, even if it is not in the direct line of one's own research.

The best approach of studying a specific method is by example. One should select a suitable case study of medium complexity (this book is full of these). The first step is to formalise the case study, i.e., to transfer it into the language of the chosen method. Already in this step one might find limitations or restrictions that one would like to study further. The next step is to check if the formalisation is an adequate representation of the case study. The modelling of systems and of proof obligations needs to be faithful.

Now, it is time for reflection: what insight can be gained into the formal representation with the chosen Formal Method? Here, one can try to derive properties manually – using a calculus, or even directly applying the semantics. Only in the next step one should reproduce the manual results in tools, if available. This order of first thinking and then using tools is important for keeping one's mind open to the possibilities of a Formal Method. Tools usually cover only certain aspects, namely those most relevant to their developers. Experience suggests that such a case study-driven approach creates good research topics.

Choosing a Formal Method in an Industrial Project

In the industrial scenario, it is often a challenge to choose an appropriate Formal Method for a particular industrial problem. Factors to be considered include

- the qualification and availability of staff,
- the degree of formalisation of existing documents,
- the development processes and capability maturities within the company, and

- the available budget in relation with the expected benefits.

Moreover, for each Formal Method to be considered, the availability of industrial strength tools is a decisive factor. In order to be usable for an industrial project, a tool has to satisfy certain criteria.

- It needs to be supported: that is, during a certain amount of time (usually, a time period well beyond the lifespan of the product, which can be several years) there must be a reliable partner offering maintenance, error correction, adaptation to evolving platforms, further development, and advice to users.
- It needs to be documented: that is, there must exist user manuals, online help, training material, and coaching resources for the engineers who shall use the tool. To this end, competences and skills profiles need to be established.
- It needs to integrate smoothly into the existing development processes. That is, exchange formats need to be available and translations between different representations should exist or be easily implementable.
- Its use should be predictable: there need to be good arguments that the intended task can be accomplished with the help of the tool, within a time frame which is allocated in advance.
- In some cases, it even needs to be qualified: for the development of safety-critical systems, it is not permitted to use an arbitrary tool; at least, an analysis of capabilities and alternatives must be conducted.

One risk in selecting a Formal Method is that most practitioners tend to favour their own area of expertise. Other approaches, which actually might be better suited, are easily overlooked. Thus, it is a good idea to consult several experts covering different areas.

Having identified a suitable Formal Method, the next step is to carry out a pilot project. Here, a small but relevant part of the problem is solved in an exemplary way using the selected Formal Method. This involves thorough time measurement and documentation for all activities that the Formal Method incurs: modelling, installing the tools, using the tools, integrating the tools, interpreting the results, etc. Reflecting upon this allows to check if the chosen approach is indeed feasible. On the management level, the pilot project then has to be evaluated as to whether it is an improvement with respect to current practice. It should be both more effective, in that it allows to achieve better results than previous methods, and more efficient, i.e., in the long run it should offer a better cost/result ratio.

1.4 Closing Remarks

In this chapter we developed an understanding of the key ingredients of Formal Methods: syntax, semantics and method. The syntax is usually given in Backus-Naur-Form; the semantics is mostly presented in either operational, denotational, or axiomatic style; the method says how to work with the language. Formal Methods are useful in classical as well as in agile software development processes. They are used to achieve precision in design documents and to support various forms of system analysis. International standards recognise and recommend the use of various Formal Methods. In practise, Formal Methods require tool support. As several academic and industrial success stories demonstrate, Formal Methods play an increasing role in industrial practice.

1.4.1 Current Research Directions

In this section we point out several challenges and current research topics for Formal Methods.

Advancement. An account of the historical development of Formal Methods is given in Part IV. In this context, the question is whether there still is a need for Formal Methods to evolve further. Considering computational systems, we see that their size and complexity is ever increasing. Also, computers for executing Formal Methods tools become more powerful. However, the increase of the problem size often outgrows what tools can handle. This is due to the fact that most algorithms in Formal Methods are complex. As a consequence, there is a constant need to improve methods and tools. Therefore, the questions of how to develop 'good' Formal Methods, i.e., Formal Methods which are efficient and usable, will stay.

Integration. As was shown above, various formal software modelling techniques have been developed. In systems' design, these can be used to describe different aspects of the very same system. In Example 1 (see page 2) concerning the ISS, for instance, the correctness of the fault management layer was analysed using the process algebra CSP. In order to guarantee a minimal throughput on the station's MIL bus, Schlingloff performed a stochastic analysis using Timed Petri-Nets [THSS98]. Generally, in such circumstances the question arises whether different models provide a consistent view of the system, and whether analysis results for one aspect can be re-used and integrated into the analysis of other aspects. Here, UML provides an integration of various modelling frameworks. However, this integration is on the syntactical level only. Semantical and methodological integrations are still being researched [KM18].

Industrial Practice. The long standing question of how to turn Formal Methods into good industrial practice still remains a challenge. For example, the aerospace standard DO-333, published in 2012, allows Formal Methods to replace traditional engineering practice, e.g., in testing, code inspection, and code review. However, there are not yet sufficiently many qualified tools available. Moreover, it is not always clear where Formal Methods offer better results than the established processes.

Parallelisation. Another current research trend is that the impending multi/many core revolution poses the question of how to develop efficient parallel algorithms. In Formal Methods, e.g., for model checking, SAT and SMT solving, and automated theorem proving there are first proposals of algorithms tailored towards the execution on multi/many core machines.

Re-use. Nowadays, systems are rarely constructed from scratch. New, functionally increased and more complex software products are built on top of existing ones. Systems are rather improved than newly developed, i.e., there is a constant software evolution. Like other industrial products, also software is designed in product lines. Formal Methods have not yet come up with adequate techniques to reflect these development processes by evolutionary modelling and verifying of systems. The main challenge is how to re-use verification artefacts.

Compositionality. As systems become more and more complex and spatially distributed, there is an increasing need to verify large, parallel systems. For example, there are Formal Methods being developed to deal with service-oriented architectures, where autonomous software agents in the Internet cooperate in order to achieve a certain task. Questions include the interaction with an unknown, non-predictable environment, functional correctness, quantitative analysis, verification of service level agreements, and security (see Ch. 8).

Cyber-physical agents. Yet another challenge concerns the application of Formal Methods in cyber-physical systems. These are 'agent-based systems in the physical world', i.e., intelligent sensor networks, swarms of robots, networks of small devices, etc. Part of the problem is the combined physical and logical modelling. For modelling systems which have both discrete and continuous state changes, hybrid automata have been suggested as a formal framework. However, current methods are not yet sophisticated enough to allow the verification of industrial strength applications. Additionally, cyber-physical systems have to deal with unreliable communication, imprecise sensors and actors, faulty hardware etc. For each of these problems, initial individual approaches are being developed, however, it is an open research topic to develop convincing Formal Methods tackling them in combination.

Artificial intelligence and machine learning. Techniques based on artificial intelligence (AI) in general, and machine learning (ML) in particular, are massively being used in deployed software. Many applications using AI/ML are safety-critical (e.g., autonomous cars), so correctness is

paramount. But the interaction between Formal Methods and AI/ML goes beyond the standard 'let us use a Formal Methods technique to prove the correctness of this algorithm – which happens to use AI/ML.' Indeed, the use of AI/ML introduces new challenges for formal verification, in particular in the area of deep neural networks where sometimes an algorithm has been learned without a clear understanding of the process of its creation. This makes it difficult to assert the correctness of the outcomes of the algorithm, which might require *transparency* in the underlying models and the used techniques and methods for learning algorithms, that is to get 'explainable' AI [Mol19]. Other interesting research directions are the use of machine learning to improve Formal Methods [ALB18], and the application of Formal Methods to AI/ML [HKWW17, SKS19, WPW+18].

Finally, we briefly mention further applications of Formal Methods beyond software engineering, such as biological systems and, more recently, ecology, economics and social sciences.

Biological systems. Formal Methods started to be used to model *biological systems* following 1998 Gheorghe Păun's definition of P systems [Pău98, Pău00], a computational model inspired from the way the alive cells process chemical compounds in their compartmental structure. Variations of this model led to the birth of a research field known as *membrane computing* [PRS09]. Although P systems were originally intended as a biologically-inspired computational model, it was soon understood that they could provide a modelling language to formally describe biological systems and on which to base tools to reason about their evolution. The Grand Challenge for computing that David Harel proposed in 2002 [Har03a] to model a full multicellular animal, specifically the *C. elegans* nematode worm, as a reactive system led to the extension of various Formal Methods, traditionally used in computer science, to make them suitable to the modelling of biological systems. For example, the Performance Evaluation Process Algebra (PEPA) [GH94] was extended in order to handle some features of biochemical networks, such as stoichiometry and different kinds of kinetic laws, thus resulting in the Bio-PEPA language [CH09a], whose models can be fed to PRISM [KNP10] for (stochastic) model checking. Another development in this application area has been to move from the modelling of a single organism to the modelling of population dynamics. Some formal notations, such as the process algebra-based BlendX language [DPR08], have been developed to model *ecological systems* consisting of various populations (or *ecosystems*, in a wider context), aiming at overcoming the technical and cultural gap between life scientists and computer scientists [CS15]. The final objective of this modelling approach is not only to understand the functioning of the ecosystem but also to test possible control interventions on some of the system components aiming at performing adjustments to the system behaviour and evaluate the impact of such intervention on the entire ecological system [CS15]. Exam-

ples are: pest eradication [BCB+10], preservation/reintroduction of species [CCM+10], disease control [CH09b] and even tumour control (a tumour can be seen as an 'ecosystem' consisting of various populations of normal and mutant cells) [SBMC15].

Economics. The most successful application of Formal Methods to economics is in the area of business process management. Will van der Aalst has been using variants of Petri nets to model enterprise resource planning systems, cooperative work, resource allocation and inter-organisational business processes [VDAS11]. It is in this application area that the two analytical philosophies of the Formal Methods community and the data mining / big data community are getting closer and closer. Rozinat and Van der Aalst developed methodologies to perform *conformance checking*, also called *conformance analysis*, that is, the detection of inconsistencies between an *a priori* process model and an *a posteriori* model produced by applying process mining to the corresponding execution log [Aal11, RA08]. The future of this approach goes well beyond the specific application to business process management, in particular in humanities.

Social Sciences. In fact, conformance checking seems appropriate for the analysis of social networks and peer-production systems, and the first attempts in this direction have being done in the areas of collaborative learning and OSS (Open Source Software) development [MCT15]. More in general, data mining, text mining and process mining, through conformance checking, can provide appropriate and effective validation tools for formal models of social systems, opening the application of Formal Methods to the vast area of social sciences.

Another promising use of Formal Methods in social sciences is the modelling of privacy. For example, privacy is an issue in sociology, politics, and legislation. Formalising privacy policies and realising enforcing mechanisms is not easy. The challenges of privacy for Formal Methods have been discussed for instance in [TW09]. Also, there is an increasing need for technology-based solutions to help lawyers to draft and analyse contractual documents, and citizens to understand the huge amount of different kinds of agreements and terms of services on paper and digital devices. Formal Methods can play a crucial role in providing solutions to help handling such complex documents (see Ch. 6).

The above items present opportunities for research on topics which are both scientifically exciting and have a large impact on society. In order to start such research, one can build upon the material presented in the subsequent chapters.

References

[Aal11] Wil M. P. Van Der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[ABL96] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grew out of a Dagstuhl Seminar, June 1995)*, LNCS 1165. Springer, 1996.

[Abr94] Jean-Raymond Abrial. Steam-boiler control specification problem. `https://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html`, 1994.

[ALB18] Moussa Amrani, Levi Lúcio, and Adrien Bibal. ML + FV = ♡? A survey on the application of machine learning to formal verification, 2018. `http://arxiv.org/abs/1806.03600`.

[BA05] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *ZB 2005*, LNCS 3455, pages 334–354. Springer, 2005.

[Bac59] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *Proceedings of the International Conference on Information Processing*. UNESCO, 1959. Available via the web site of the Computer History Museum's Software Preservation Group, `http://www.softwarepreservation.org`.

[Bar11] Janet Elizabeth Barnes. Experiences in the industrial use of formal methods. In *AVoCS'11*. Electronic Communications of the EASST, 2011.

[BBBB09] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In *SAFECOMP 2009*, LNCS 5775, pages 187–200. Springer, 2009.

[BCB+10] Thomas Anung Basuki, Antonio Cerone, Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, and Elisabetta Rossi. Modelling the dynamics of an aedes albopictus population. In *AMCA-POP 2010*, volume 33 of *Electronic Proceedings in Theoretical Computer Science*, pages 18–36. Open Publishing Association, 2010.

[BCH+12] Craig Burton, Chris Culnane, James Heather, Thea Peacock, Peter Y. A. Ryan, Steve Schneider, Sriramkrishnan Srinivasan, Vanessa Teague, Roland Wen, and Zhe Xia. A supervised verifiable voting protocol for the Victorian electoral commission. In *EVOTE 2012*, volume 205 of *LNI*, pages 81–94. GI, 2012.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[BH95a] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[BH95b] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, 1995.

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods ... Ten years later. *IEEE Computer*, 39(1):40–48, 2006.

[BKPS97] Bettina Buth, Michel Kouvaras, Jan Peleska, and Hui Shi. Deadlock analysis for a fault-tolerant system. In *AMAST*, LNCS 1349. Springer, 1997.

[BMS96] Manfred Broy, Stephan Merz, and Katharina Spies, editors. *Formal Systems Specification, The RPC-Memory Specification Case Study*, LNCS 1169. Springer, 1996.

[BPS98] Bettina Buth, Jan Peleska, and Hui Shi. Combining methods for the livelock analysis of a fault-tolerant system. In *AMAST*, LNCS 1548. Springer, 1998.

[CCM+10] Mónica Cardona, M. Angels Colomer, Antoni Margalida, Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez, and Delfí Sanuy. A P system based model of an ecosystem of some scavenger birds. In *WMC 2009*, LNCS 5957, pages 182–195. Springer, 2010.

[CH09a] Federica Ciocchetta and Jane Hillston. Bio-PEPA: a framework for the modelling and analysis of biochemical networks. *Theoretical Computer Science*, 410:3065–3084, 2009.

[CH09b] Federica Ciocchetta and Jane Hillston. Bio-PEPA for epidemiological models. In *PASM 2009*, volume 261 of *Electronic Notes in Theoretical Computer Science*, pages 43–69. Open Publishing Association, 2009.

[CK11] Véronique Cortier and Steve Kremer, editors. *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press, 2011.

[Coc00] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2000.

[CRD+20] Antonio Cerone, Markus Roggenbach, James Davenport, Casey Denner, Marie Farrell, Magne Haveraaen, Faron Moller, Philipp Koerner, Sebastian Krings, Peter Ölveczky, Bernd-Holger Schlingloff, Nikolay Shilov, and Rustam Zhumagambetov. Rooting formal methods within higher education curricula for computer science and software engineering – A White Paper, 2020. `https://arxiv.org/abs/2010.05708`.

[CS15] Antonio Cerone and Marco Scotti. Research challenges in modelling ecosystems. In *SEFM 2014 Collocated Workshops*, LNCS 8938, pages 276–293. Springer, 2015.

[Das06] Manuvir Das. Formal specifications on industrial-strength code – from myth to reality. In *Computer Aided Verification*, LNCS 4144. Springer, 2006.

[dBfI12] Die Beauftragte der Bundesregierung für Informationstechnik. Das V-Modell XT. `http://www.v-modell-xt.de`, 2012.

[DHR+12] Denise Demirel, Maria Henning, Peter Y. A. Ryan, Steve Schneider, and Melanie Volkamer. Feasibility analysis of prêt à voter for german federal elections. In *VoteID 2011*, LNCS 7187, pages 158–173. Springer, 2012.

[DPR08] Lorenzo Dematté, Corrado Priami, and Alessandro Romanel. The BlenX language: a tutorial. In *Formal Methods for Computational Systems Biology*, LNCS 5016, pages 313–365. Springer, 2008.

[FH01] Marc Frappier and Henri Habrias, editors. *Software Specification Methods*. Springer, 2001.

[Fix08] Limor Fix. Fifteen years of formal property verification in Intel. In *25 Years of Model Checking*, LNCS 5000, pages 139–144. Springer, 2008.

[GBP20] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In *Formal Methods for Industrial Critical Systems*, pages 3–69. Springer, 2020.

[GG13] Hubert Garavel and Susanne Graf. *Formal Methods for Safe and Secure Computers Systems*. Federal Office for Information Security, 2013. `https://www.bsi.bund.de/DE/Publikationen/Studien/Formal_Methods_Study_875/study_875.html`.

[GH94] Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 794, pages 353–368. Springer, 1994.

[GJBB13] James Gosling, Bill Joy, Guy Steele Gilad Bracha, and Alex Buckley. The Java language specification, 2013.

[GKB11] Wolfgang Grieskamp, Nico Kicillof, and Bob Binder. Microsoft's protocol documentation program: Interoperability testing at scale. *Communications of the ACM*, 2011.

[GKSB10] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Softw. Test. Verif. Reliab.*, 2010.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[Hal12] Nicolas Halbwachs. *A Synchronous Language at Work: The Story of Lustre*, pages 15–31. Wiley, 2012.

[Har03a] David Harel. A grand challenge for computing: Towards full reactive modeling of a multi-cellular animal. *Bull. EATCS*, 81:226–235, 2003.

[Har03b] J. Harrison. Formal verification at Intel. In *18th Annual IEEE Symposium of Logic in Computer Science*, pages 45–54, 2003.

[HKW+07] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME - the common component modeling example. In *CoCoME*, LNCS 5153, pages 16–53. Springer, 2007.

[HKWW17] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV'17*, LNCS 10426, pages 3–29. Springer, 2017.

[HLQB] Brian Hackett, Shuvendu K. Lahiri, Shaz Qadeer, and Thomas Ball. Scalable modular checking of system-specific properties: Myth or reality?

[HS12] James Heather and Steve Schneider. A formal framework for modelling coercion resistance and receipt freeness. In *FM 2012*, LNCS 7436, pages 217–231. Springer, 2012.

[IEC10] IEC. `http://www.iec.ch/functionalsafety`, 2010.

[JW08] Cliff B. Jones and Jim Woodcock, editors. *Formal Aspects of Computing*, volume 20, No 1. Springer, 2008.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[KBP+01] Erik Kamsties, Daniel M. Berry, Barbara Paech, E. Kamsties, D. M. Berry, and B. Paech. Detecting ambiguities in requirements documents using inspections. In *First Workshop on Inspection in Software Engineering*, 2001.

[KDE09] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In *ICFP 2009*, pages 91–96. ACM, 2009.

[KM18] Alexander Knapp and Till Mossakowski. Multi-view consistency in UML: A survey. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, LNCS 10800, pages 37–60. Springer, 2018.

[KNP10] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic model checking for systems biology. In *Symbolic Systems Biology*, pages 31–59. Jones and Bartlett, May 2010.

[Koz94] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.

[KRT13] Dalia Khader, Peter Y. A. Ryan, and Qiang Tang. Proving prêt à voter receipt free using computational security models. *USENIX Journal of Election Technology and Systems (JETS)*, 1(1):62–81, 2013.

[KS12] Dexter Kozen and Alexandra Silva. Left-handed completeness. In *Relational and Algebraic Methods in Computer Science*, LNCS 7560, pages 162–178. Springer, 2012.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[LL95] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems – Case Study Production Cell*, LNCS 891. Springer, 1995.

[LSGP07] T Lecomte, T Servat, and G G Pouzancre. Formal methods in safety-critical railway systems. In *Brazilian Symposium on Formal Methods: SMBF*, 2007.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MCT15] Patrick Mukala, Antonio Cerone, and Franco Turini. Process mining event logs from floss data: State of the art and perspectives. In *SEFM 2014 Collocated Workshops*, LNCS 8938, pages 182–198. Springer, 2015.

[Mol19] Christoph Molnar. Interpretable machine learning, 2019. `https://christophm.github.io/interpretable-ml-book/`.

[MS08] Satish Mishra and Bernd-Holger Schlingloff. Compliance of CMMI process area with specification based development. In *Conference on Software Engineering Research, Management and Applications*, SERA '08, pages 77–84. IEEE Computer Society, 2008.

[Pău98] Gheorghe Păun. Computing with membranes. Technical Report 208, Turku Centre for Computer Science, November 1998.

[Pău00] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Science*, 61(1):108–143, 2000.

[PB99] Jan Peleska and Bettina Buth. Formal methods for the international space station ISS. In *Correct System Design*, LNCS 1710. Springer, 1999.

[PRS09] Gheorghe Păun, Grzegorz Rozemberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford Handbooks in Mathematics. Oxford University Press, December 2009.

[RA08] Anne Rozinat and Wil M. P. Van Der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.

[RBH+09] Peter Y. A. Ryan, David Bismark, James Heather, Steve Schneider, and Zhe Xia. Prêt à voter: a voter-verifiable voting system. *IEEE Transactions on Information Forensics and Security*, 4(4):662–673, 2009.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, LNCS 5153. Springer, 2008.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, January 1966.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[SBMC15] Sheema Sameen, Roberto Barbuti, Paolo Milazzo, and Antonio Cerone. A mathematical model for assessing KRAS mutation effect on monoclonal antibody treatment of colorectal cancer. In *SEFM 2014 Collocated Workshops*, LNCS 8938, pages 243–258. Springer, 2015.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.

[SKS19] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. Formal verification of neural network controlled autonomous systems. In *HSCC 2019*, pages 147–156. ACM, 2019.

[SLC+11] Steve Schneider, Morgan Llewellyn, Chris Culnane, James Heather, Sriramkrishnan Srinivasan, and Zhe Xia. Focus group views on prêt à voter 1.0. In *REVOTE 2011*, pages 56–65. IEEE, 2011.

[SSC+12] Steve Schneider, Sriramkrishnan Srinivasan, Chris Culnane, James Heather, and Zhe Xia. Prêt á voter with write-ins. In *VoteID 2011*, LNCS 7187, pages 174–189. Springer, 2012.

[THSS98] L. Twele, B-H. H. Schlingloff, and H. Szczerbicka. Performability analysis of an avionics-interface. In *Proc. IEEE Conf. on Systems, Man and Cybernetics*, 1998.

[TW09] Michael Carl Tschantz and Jeannette M. Wing. Formal methods for privacy. In *FM'09*, volume 5850 of *LNCS*, pages 1–15. Springer, 2009.

[VDAS11] Wil M. P. Van Der Aalst, , and Christian Stahl. *Modeling Business Processes: A Petri Net-Oriented Approach*. MIT Press, May 2011.

[Ver07] The Verisoft XT Project, 2007. `http://www.verisoft.de/StartPage.html`.

[WPW+18] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *USENIX'18*, pages 1599–1614. USENIX Association, 2018.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.