
Foreword

Computer systems are becoming ubiquitous. Many of the most important and prevalent ones are reactive systems. Reactive systems include microprocessors, computer operating systems, air traffic control systems, as well as on-board avionics and other embedded systems. These systems are characterized technically by their ongoing, ideally infinite behavior; termination is impossible or aberrant behavior, in contrast to classical theories of computation. Reactive systems tend to be characterized in practice by having failure modes that can severely compromise safety, even leading to loss of life. Alternatively, errors can have serious financial repercussions such as expensive recalls. Reactive systems need to be correct before being deployed.

To determine whether such reactive systems do behave correctly, a rich mathematical theory of verification of reactive systems has been developed over the last two decades or so. In contrast to earlier work emphasizing the role of proofs in deductive systems to establish correctness, the alternative suggestion is to take a model-theoretic view. It turns out that this permits the process of reasoning about program correctness to be fully automated in principle and partially automated to a high degree in practice.

It is my pleasure to introduce Klaus Schneider's excellent book *Verification of Reactive Systems: Formal Methods and Algorithms*. This book is the story of reactive systems verification, reflecting Klaus's broad expertise on the subject. It addresses both applications and theory, providing especially strong coverage of basic as well as advanced theory not otherwise available in book form. Key topics include Kripke and related transition structures, temporal logics, automata on infinite strings including Safra's determinization construction, expressiveness and Borel hierarchies of ω -languages, as well as monadic predicate logics. An underlying theme is the use of the vectored μ -calculus to provide an elegant "Theory of Everything". *Verification of Reactive Systems* belongs on the bookshelf of every serious researcher into the topic. It should also serve as a valuable text for graduate students and advanced undergraduates.

April 2003

E. Allen Emerson,
Endowed Professor of Computer Sciences
University of Texas at Austin

Preface

The design of modern information processing systems like digital circuits or protocols is becoming more and more difficult. A large part of the design costs and time (about 70%) is currently spent on methods that try to guarantee the absence of design errors. For this reason, designing systems is now more and more a synonym for verifying systems.

The research into the verification of reactive systems, in particular, into model checking, is one of the most impressive successes of theoretical computer science. Two decades after the publication of the basic papers on the formal foundation, the methods became mature enough for industrial usage. Nowadays, the hardware industry employs hundreds of highly specialized researchers working with formal methods to detect design bugs.

When I entered this field, it was an enormous effort to read hundreds of papers to understand the relationships between the different formal methods that are currently in use. It was surprising to me that there was no book covering all these methods, even the basic ones, although there is such a huge interest in them. For this reason, I decided to write this book to provide newcomers and researchers with a textbook that covers most of the relevant logics, with a particular emphasis on (verification and translation) algorithms.

The book is intended for graduate students as well as for researchers already working in this area. It is self-contained and gives proofs and algorithms for all important constructions. For a less detailed and formal introduction, I want to recommend the book of Clarke, Grumberg, and Peled [111]. Supplemental material on actual tools is found in [38], and further topics on the μ -calculus and infinite games are found in [221].

There are many persons I have to thank for helping me to write this book. In particular, I want to thank Detlef Schmid and the hardware verification group at the University of Karlsruhe, in particular Jorgos Logothetis, Tobias Schüle, and Roberto Ziller. Many discussions with Moshe Vardi moved me to improve the book. Allen Emerson was soon interested in the project and also gave fruitful comments. Moreover, I want to thank Amir Pnueli, Wolfgang Thomas, and Peter Schmitt for comments on early versions of the manuscript. Last, but not least, it should be mentioned that the editors of the EATCS series, in particular, Prof. Brauer, and the team at Springer-Verlag helped me to publish this book.

Kaiserslautern, September 2003

Klaus Schneider

Introduction

Quo facto, quando orientur controversiae, non magis disputatione
opus erit inter duos philosophos, quam inter duos computistas.
Sufficiet enim calamos in manus sumere sedereque ad abacos,
et sibi mutuo (accito si placet amico) dicere: calculemus¹.
— *Gottfried Wilhelm Leibniz (1646-1716)*

1.1 Formal Methods in System Design

1.1.1 General Remarks and Taxonomy

The development of information processing systems, especially if these consist of concurrent processes, is a very complicated task. In fact, modern computer systems are the most complicated structures mankind has ever built in its history: Modern microprocessors are implemented by millions of transistors; operating systems and software applications consist of millions of lines of code. Therefore, it is no wonder that these systems often have errors that lead to serious malfunctions, even if the systems have been extensively tested to validate their correctness.

While the construction of erroneous systems has never been tolerated, the need to avoid errors in design is becoming more and more important. One reason for this is that system sizes are rapidly growing. If we assume that the number of errors in these systems grows with their size, it becomes clear that the larger the systems are, the more troublesome they will be, and the probability that they will actually work is reduced. Very large systems

¹ The translation is roughly as follows: whenever there are different opinions about certain facts, one should not discuss them like philosophers usually do; instead one should 'calculate' the truth. Leibniz and Newton were the first who tried to replace at least some parts of the creativity used by mathematicians by rules of a calculus that could be implemented by machines.

could contain so many errors that they will never successfully run. A couple of large software projects have demonstrated this already.

But even if the system's development can be successfully completed, the time spent for debugging and testing grows rapidly with the system's size. It is not unusual that more than 70% of the design time is spent on simulation. Errors found late in the design may lead to expensive redesigns that in turn lead to delays in the time-to-market. Even if this delay is only about two or three months, the overall economic success of the product may be endangered.

Another reason for the increased pressure to avoid design errors is that computer systems are used more and more in applications where their malfunctioning can cause extensive damage or could even endanger human lives. Nuclear plants, aircrafts, and automobiles are more and more controlled by so-called *embedded systems*. These embedded systems are in principle complete computer systems, since they usually consist of a microprocessor that runs special software. Often more than one task has to be performed by the system, and for this reason, it is often divided into several processes or threads that implement the desired tasks. For reasons of efficiency, specialized hardware is often used to increase the throughput. For example, image processing systems often have special processing units like JPEG and MPEG decoders for compressing and decompressing image data to increase the throughput.

While the complexity of systems steadily increases, the design methods to guarantee correct systems have not emerged at the same speed. In software design, the introduction of object-oriented analysis and design has helped to structure the designs to obtain reusable designs. In hardware design, special design flows and design tools are used to translate a high-level algorithmic description given in hardware description languages like VHDL or Verilog to a circuit netlist. This way, it is possible to design quite large systems, much larger and faster than one expected some years ago.

However, the design of embedded systems does not only involve the design of hardware or software. Instead, the combination of both is a critical problem since the tasks that are implemented in hardware or software must be carefully selected. For this reason, hardware-software codesign methods are in use that start with a realization-independent description of the system, which may however already be partitioned into several tasks. In an analysis phase, modules are detected whose functions influence the speed of the system. Moreover, estimates for implementation costs also need to be considered. Dependent on these facts, a partition into hardware and software is made and the design of the hardware and software parts is done in a conventional manner.

So, there are already design flows that allow hardware and software development and also hardware/software codesigns. However, these design flows are very complex. If an error is detected late in the design, its correction may influence even the hardware-software partitioning so that a com-

plete redesign has to be made. It is therefore mandatory that design errors are detected as soon as possible, i.e., at the level of the realization-independent description.

One way to detect design errors is clearly to test the design by means of implementing prototypes. This is a standard task in software design, but is not so simple for the design of hardware. Here, one is usually forced to simulate the circuits as it is too expensive to fabricate prototype circuits. However, in the last few years, programmable hardware such as field-programmable gate arrays (FPGAs) [70, 122] have been developed that allow the implementation of hardware prototypes. However, the speed of these prototypes is slower than the speed that will be obtained by a later implementation as an application-specific integrated circuit (ASIC). Therefore, simulation of the hardware-software system is often still necessary to check the correct interaction between the software and hardware parts. However, the problem with simulation is that it is quite slow when large designs are to be simulated, and what is even more intrinsic to the approach is that only errors can be found, while the *absence of errors* can never be shown.

For this reason, the application of formal methods in the design of software, hardware, and hardware-software codesign is more and more frequently discussed. There are some approaches that are independent of the kind of system to be verified. However, in most cases, the kind of system determines the kinds of properties to be verified, and this in turn makes one or another verification formalism more or less suited. For example, when a hardware controller is to be verified, we will usually not have to consider abstract data types. Instead, we are confronted with some sort of finite state machine that has to emit the desired control signals at the right point of time. For this reason, the kind of system is important for the choice of a suitable verification formalism.

Therefore, it is no wonder that a plethora of formal methods has been developed over time. Some of them consider the same problem and try to solve it in different ways, but there are also very different approaches that have almost nothing in common. In fact, only a few of the formal methods that are considered nowadays are discussed in this book, as it has a special focus on so-called *reactive systems* and *finite-state based verification procedures*.

For this reason, we will first consider in the next section different ways to apply formal methods to the design of complex systems. In particular, we distinguish between three main approaches: formal specification, verification, and formal synthesis or program derivation. While these approaches only explain where and how formal methods are applied in the design flow, no formal method has yet been determined. The particular choice of a formal method crucially depends, as already mentioned, on the kind of system, and also on the kind of specification, that is considered. Therefore, in Section 1.1.3, we give a classification of systems with their related properties. Having then seen the important classifications of formal methods, systems and properties, we then consider in Section 1.2 the history of some formal methods.

The content of Section 1.2 is however rather limited to the focus of this book and does not consider the genealogy of other formalisms. Nevertheless, it is very fruitful to see the relationship between the considered approaches.

1.1.2 Classification of Formal Methods

So far, we have discussed the usefulness of formal methods in the design of software, hardware or hardware-software codesigns. There are, however, very different formal methods and, furthermore, very different ways that these formal methods can be used in the design.

In this section, we discuss the main applications of formal methods in system design and can therefore give a first classification of formal methods. We can also explain what can be achieved by formal methods in system design and what they can not do. There are at least the following ways to use formal methods in system design:

- writing formal specifications
- proving properties about the specification
- deriving implementations from a given specification
- verifying specifications w.r.t. a given implementation

We will discuss these different approaches and, moreover, discuss the limitations of formal methods in the design process.

Writing a formal specification. Formal methods are used to reason about mathematical objects. However, hardware circuits are, e.g., not mathematical objects, but physical objects of the real world. Therefore, it is necessary to develop a mathematical model of the system and also to describe the properties that the system should have by means of some mathematical language. This should be clear, but it must be stressed for two reasons: First, the properties are normally given in an informal manner, i.e., by means of natural language. It is frequently the case that these requirements are inconsistent, and therefore, they can not be satisfied at all. Secondly, no matter what formal method is used, it can only argue about the formally given system description and the formally given properties. If these are not what is actually intended, the application of the formal methods will prove things that nobody is really interested in.

For the system's model, this problem is often overestimated. Any programming language with a formal semantics is a mathematical model that can in principle be used for the application of formal methods. At least, it can be converted into another mathematical model. Hence, any programmer and any hardware designer already deals with formal models even if 'formal methods' are not believed to be applied in the design flow.

Additionally, the environment of the system has often to be modeled to show that the system meets its specification. The environment often restricts the inputs that can occur by physical restrictions. For example, a

robot's arm can not reach every place and therefore, it is not necessary to consider all locations of it. But even if the environment model has to be taken into account, the problem is overestimated. In any engineering discipline mathematical models are used; and nobody who finds a problem in the construction of a crane would suggest that crane builders should abandon mathematics.

Proving properties of the specification. We have already pointed out that specifications are often given in an informal language and must therefore be first rephrased in a formal specification language. This is an error-prone task, and therefore, it is often desirable to prove some properties of a given specification to see that it actually means what one has in mind. In particular, it is important to show its satisfiability to assure that at least one solution exists. For example, the specification $a^{(0)} \wedge a^{(1)} \wedge \forall t. a^{(t+1)} = \neg a^{(t)}$, that should express that a is true for the first two points of time and then oscillates, is not satisfiable (a correct specification would be $a^{(0)} \wedge a^{(1)} \wedge \neg a^{(2)} \wedge \forall t. a^{(t+3)} = \neg a^{(t+2)}$).

It is often fruitful to prove the equivalence of different specifications to get a deeper understanding of the task that is to be implemented. Even if the specifications are not verified against a later implementation these steps often lead to a deeper understanding and a better structuring of the problem. Therefore, formal methods allow us to find errors in the specification phase. Hall writes in his article [231] about his experience with formal methods in software design in his company:

In an informal specification, it is hard to tell what is an error because it is not clear what is being said. When challenged, people try to defend their informal specification by reinterpreting it to meet the criticism. With a formal specification, we have found that errors are much more easily found – and, once they are found, everyone is more ready to agree that they are errors.

Deriving implementations from a specification. Once a specification has been set up and one has figured out that it is indeed what is desired, it would be helpful to have a design method that could automatically derive a system's implementation that fulfills the given requirements. This has actually been the idea of the fifth generation programming languages like PROLOG, where the specification and implementation phases become closely related to each other.

However, specifications are often given in a declarative manner and not in a constructive manner. This means that these specifications only describe *what* the system should do, but not *how* this function can be achieved. It is certainly not possible to derive correct programs from declarative specifications since these problems are intrinsically undecidable so that machines can never solve them. Therefore, the construction of appropriate implementations will always remain a creative task for human beings.

Nevertheless, the algorithm can be given in an abstract manner where many implementation details are ignored. It is then possible to use appropriate design tools to construct a more detailed system's implementation out of such an abstract system description. For example, the algorithm can be written in a high-level programming language and a compiler can then be used to translate the algorithm to some machine language so that it can be executed on a microprocessor. In hardware design, a high-level hardware description can be translated by so-called synthesis tools to a low level description that can be physically implemented. Similarly, software and hardware descriptions can be generated from abstract and realization independent descriptions for example those given in SDL [139, 524] or synchronous languages [42, 230]. Therefore, *formal software* [93] and *hardware synthesis* [57, 58, 297] have their applications here.

Verifying specifications w.r.t. a given implementation. As it may be possible to automatically derive detailed system descriptions by less detailed ones, or at least to use tools that assure the correctness of the manually applied design steps, there is no need to reprove properties at different levels of abstraction. The design steps that are used to refine the system's description must not affect the validity of the specification. However, it still remains to check whether the abstract implementation satisfies the originally given specifications. This process is normally called formal verification of the system and this is also the main topic of this book.

There are two main ways formal verification can be applied. On the one hand, one can describe both the system's model Φ_{imp} and the specification Φ_{spec} in a formal language and consider the resultant formulas with a special calculus that is suited for the chosen formal language. As both the specification and the system are given as formulas of the same logic, the remaining task is to prove properties of the language such as $\Phi_{\text{imp}} \rightarrow \Phi_{\text{spec}}$ or $\Phi_{\text{imp}} \leftrightarrow \Phi_{\text{spec}}$. Therefore, these methods are based on *automated theorem proving* for certain formal languages (logics). Usually, one wants to confirm that the formula $\Phi_{\text{imp}} \rightarrow \Phi_{\text{spec}}$ holds, i.e., to check that the specification holds for the implementation. The specification can therefore, and in general will, only describe a partial behavior of the system.

While the above mentioned approach is reduced to the theorem proving problem $\Phi_{\text{imp}} \rightarrow \Phi_{\text{spec}}$ of a certain logic, another approach has been developed since the 1980s which is called *model checking*. In model checking, the system's description is not given in the logic. Instead, it is given as an interpretation \mathcal{M}_{imp} of the considered logic. A model checking procedure has the task of evaluating the specification Φ_{spec} in the interpretation \mathcal{M}_{imp} .

It is often the case that the model checking problem of a logic is simpler than the related theorem proving or *satisfiability problem*, but both may also share the same complexity. It can however be easily seen that model

checking is somehow simpler than satisfiability checking on nondeterministic algorithms: Model checking must evaluate a given formula in a *given model* while satisfiability checking must check *whether an interpretation exists* such that the formula can be evaluated to true. In particular, nondeterministic machines can guess a suitable interpretation, if one exists, and can then use the corresponding model checking procedure to evaluate the formula in the model. Therefore, model checking is often less complex than the related theorem proving problem, at least when nondeterministic computation models are considered.

The difference is best seen by a simple example: Consider as specification logic the simple propositional logic. Interpretations of this logic are simply truth assignments to the variables that occur in the formulas, and model checking is simply performed by evaluating a propositional formula with such an assignment. It is clear that the latter can be done in linear time with respect to the length of the formula. Theorem proving for propositional logic is however equivalent to the satisfiability problem and therefore known to be NP-complete [124, 204].

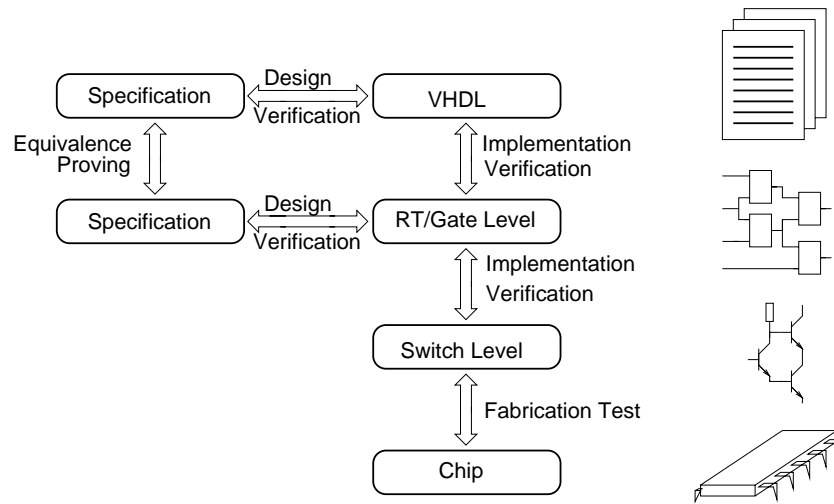


Fig. 1.1. Different ways to apply formal methods in the design

Different ways to apply formal methods for the hardware design are pictorially given in Fig. 1.1, but the restriction to hardware design is not important to the discussion at the moment. Specifications can be compared with each other, where equivalence between specifications can be checked and where one specification implies another one. *Implementation verification*, or *equivalence checking* compares different implementation descriptions with

each other. This approach is often called ‘vertical verification’ which is shown in Fig. 1.1. If one implementation is obtained by derivation with a formal method from another one, one speaks about *formal synthesis* or *program derivation*. Finally, formal specifications can be checked for a given implementation description, which is often called *property checking* or simply *(formal) verification*. In general, we may define verification to be the process of constructing a mathematical proof that shows that an implementation fulfills a given specification.

While all the mentioned ways of using formal methods in the design of complex systems are important in assuring the correctness of the system that is to be developed, we mainly consider the verification of properties in this book. Moreover, for some formalisms in this book, the verification problem, which is normally interpreted as a model checking problem can be solved by the same methods as the satisfiability problem. Therefore, we will also consider satisfiability problems for some formalisms.

One point to be discussed here is that it is often argued that the verification of a system guarantees that the system will be free of errors. This is normally not the case. What is shown in a formal verification is that the formal model of the system satisfies a formal specification. Therefore, we need to additionally assure that the specification actually describes the properties that are to be fulfilled and that the system’s model is actually what is implemented later on. We already addressed the problem in the paragraph on ‘writing a formal specification’: Programming languages, hardware description languages and other system level description languages are already formal models that can be used for the verification as well as for the refinement of the implementation. Hence, the problem of the system’s formal model being different from the later implementation is not a severe one. This not so simple with the specification: As it is impossible to guess what a specification engineer had in mind, when he or she wrote a specification, the only way to check if the specification is what is wanted is to prove the properties of the specification. A major point here is that the specification language should be as readable as possible so that it can be easily understood. Many complaints about formal methods are prejudiced by the fact that many are hard to understand. Nevertheless, even if a readable formalism is chosen, engineering practice has shown [231] that any formally given specification should additionally be explained in natural language to be better understood. This is also for reasons of redundancy.

However, even if the specification and the system are correctly given and successfully verified, there are some problems that can nevertheless lead to a malfunctioning system. For example, it may be the case that the underlying physical components that finally execute the systems computations are damaged. This is a particular problem for the fabrication of hardware circuits: Layout defects that may occur during the fabrication process may also lead to malfunctioning chips. These defects can not be checked by formal verification, and are normally avoided by *testing*. This means that a chip is

tested under a given set of test pattern sequences that distinguish a correct from a faulty circuit. The test patterns are usually automatically generated. As for large circuits the number of test patterns may become prohibitively large, additional test circuitry like scan-paths and self-tests, may become necessary. Modern test procedures establish a high quality, so that normally less than five bad chips out of one million are delivered [282].

Moreover, correct systems can be used in a wrong manner which results in *errors by wrong usage*. This is a general problem as systems are usually designed in a modular manner and are therefore implemented with assumptions on their environment. Such a problem has occurred with the Ariane 5, since a software module had been reused from the Ariane 3 that assumed certain values on the acceleration that did not hold for the Ariane 5.

Wrong usage is, in particular, a problem for embedded systems as these systems are by definition integrated into an environment which corresponds with certain assumptions in the design phase. Hence, there will always be some assumptions on the interaction between the system and its environment which may be functional, or of other kinds. For example, temperature, voltages, physical size, and many more requirements may have to be considered. Hence, the correct usage of the systems must also be specified.

Furthermore, it may be the case that the system is correct and is used in a correct manner, but *aging of the system* may lead after some years to defects. These kinds of errors can not be avoided by verification approaches. The solution here is to use fault tolerance methods, e.g., to implement redundant systems that are able to run even if one or two system's components fail.

Therefore, to guarantee 'trustworthy' systems, formal verification, test, and fault tolerance methods must all be used. There are many examples, ranging from the verification of train stations, the Pentium bug, the Ariane 5, and many others, that show the necessity of formal methods in system design. Formal methods have already led to the discovery of several errors in systems that are already in use and in standards that have been signed-off as correct.

Having outlined the limitations of formal methods, it is important to say that their usage is still essential for avoiding design errors. Modern system design flows will benefit from formal methods in that design errors can be avoided or found quicker than by traditional test and simulation methods. According to a recent investigation [449], in 82% of the cases, design errors were responsible for the malfunctioning of systems. Moreover, it has been reported [231] that the usage of formal methods has led to a better structure of a system since the different tasks that the system should perform are better understood from the beginning of its design. It might be surprising, but it is often the case that designers have captured the problem that they should have solved only after they have made the implementation. An indication for this phenomenon is that re-implementations are often much better than the first ones.

So, we have outlined different ways on how formal methods can be applied in modern system design. We have already distinguished between verification, formal synthesis, and the theorem proving of properties to check the specifications. For a further discussion, we will now try to classify different kinds of systems and their related properties in the next section. We will then finally be able to describe the kind of systems and kind of formalisms that will be considered in this book.

1.1.3 Classification of Systems

In the previous section, we outlined several ways of applying formal methods in the design of complex systems. We now refine this view in considering different kinds of systems, their different properties, and thereby, a certain preselection of verification formalisms. Depending on their architecture, systems can be classified in several ways, for example:

- asynchronous/synchronous hardware
- analog/digital hardware
- mono- or multi-processor systems
- imperative/functional/logic-based/object-oriented software
- multi-threaded or sequential software
- conventional vs. real-time operating systems
- embedded systems vs. local systems vs. distributed systems

As specific architectures such as the ones given above are dedicated to specific tasks, it is clear that one formal method will be more or less suited than the other. For example, the behavior of analog hardware is essentially based on differential equations so that their formal verification is closely related with solving these equations. In contrast to this, digital hardware only considers Boolean valued data types that may be collected in so-called bitvectors to encode finite data types like 16-bit integers. Clearly, infinite data types and complex abstract data types play no particular role in hardware systems (at least at the register-transfer level), and are therefore solely considered in software design (or in high-level design phases of hardware designs). The problems that occur for the verification of digital hardware circuits are intrinsically based on finite-state machines and are therefore, in principle, solvable by computers. However, the number of states may become prohibitively large, so that the verification approaches must fight the state-explosion problem.

Digital hardware design becomes more and more ‘soft’, which means that the hardware design starts at the level of algorithmic descriptions. For example, special applications such as image processing algorithms like the JPEG and the MPEG algorithms are often implemented in special hardware circuits that are found in modern digital cameras. On the other hand, the performance of modern microprocessors is so good that often elaborate hardware

designs are not much faster than the highly optimized microprocessor structures where many person-years of development have been invested. Therefore, hardware implementations are only recommended when efficiency is crucial for the functioning of the system or if large numbers of circuits are expected to be sold. The reason for the latter issue is that the development of a hardware circuit is much more expensive than that for an equivalent software component, and these development costs need to be recovered by the sale of a large number of circuits.

However, as more and more complex algorithms are implemented in hardware, the requirements for hardware verification also changes. The need to consider complex multi-threaded systems whose threads heavily interact with each other becomes more and more challenging. Even if no complex data types need to be considered, the tasks that are performed by modern hardware systems are more and more complex, and the number of states in these systems has reached very large numbers. The implementation of complex control tasks will moreover require the consideration of high-level description languages that can deal with many interacting threads.

Hence, the difference between this high-level hardware design and the design of multi-threaded software diminishes, so that we need, in principle, not to distinguish between hardware-oriented multi-threaded software with low-level data types and high-level hardware descriptions. A more distinguishing criterion is the paradigm of the underlying programming language, i.e., whether it is imperative, functional, logic-based or object-oriented. Logic-based languages such as PROLOG, and functional languages such as ML, Haskell, and many more are well-suited w.r.t. verification. However these languages are rarely extended by multiple threads which limits their usefulness in system design. Moreover, these languages require, in general, more memory than imperative languages, at least when the user is responsible for storage management.

Hence, for the implementation of *embedded systems*, these languages have not really been considered, since memory requirements are often of an essential importance for these systems. This domain is completely determined by imperative languages like C. With the increasing complexity and therefore the increasing need to reuse components, it is probable that object-oriented languages as C++ and maybe Java will be used in future. Therefore, software and hardware designs will come closer in the future.

Another issue is added by embedded and distributed systems. As the components of these systems have to interact with each other or with the environment, it is important to guarantee that this interaction is done in a correct and efficient manner. For example, it must be guaranteed that messages are never lost and arrive within some time constraints. The interaction of different system components with each other gives us another major distinction. From the viewpoint of a component all other components belong to the environment, we simply consider a component together with its environ-

ment in the following. Depending on the type of interaction, we distinguish the following important classes of system:

Transformational Systems: After being started by the environment, these systems read, at the beginning of their computations, some input data and produce the desired output data. It is assumed that these systems should always terminate, because the outputs are only available after the termination. An example of a transformational system is a compiler that translates a program of a high-level programming language (which is the input data) to a machine language (which is the output data).

Interactive Systems differ from transformational systems in that it is not desired that they terminate unless the user explicitly instructs them to do so. Instead, they continuously run and interact with the environment (the user). These interactions are split into the *action* that is given by the environment and the *reaction* which is the answer of the system to the given action. It is important for the distinguishing from the next category that the environment of an interactive system has to wait until the system is ready for new inputs. This means that the frequency of interactions is completely determined by the performance of the system.

Reactive Systems differ from interactive systems in that the environment can freely determine the points in time when an interaction is desired. This means that a reactive system must be at least fast enough to react on a given environment action before the next action of the environment occurs. For this reason, reactive systems must satisfy some real-time constraints and therefore fall into the category of *real-time systems*. Two different kinds of real-time constraints are distinguished: these are hard and soft real-time constraints. The former are necessary for a correct functioning of the system, while the latter ones are wanted to increase the comfort. Because of their nature, reactive systems must often satisfy hard real-time constraints and are therefore usually implemented as concurrent systems. This means that these systems consist of a couple of concurrent threads that may be dynamically generated and aborted.

The implementation of reactive systems is not an easy task. The environment often gives clear real-time constraints by the physical construction of the system. For example, a valve can be closed within a certain amount of time, a car can be stopped within a certain amount of time, or an airbag can be opened within a certain interval of time.

As it is quite clear what real-time constraints are to be met, it is not so clear, how a reactive system should be implemented to meet these constraints. In general, to be as fast as possible, one might suggest a complete hardware solution. However, this is in most cases not necessary and would be far more expensive than it should be. Therefore, the systems are normally implemented partly in hardware and in software, so that a compromise between the costs and the efficiency has to be made.

Reconsidering the system classes that we have previously considered by examining the architecture, we first note that nearly all hardware circuits are reactive systems in the following sense: In the design of asynchronous hardware, it is required that all signals become stable within a certain amount of time so that oscillating signals are avoided. In the design of synchronous hardware circuits, the points of time where the interactions will occur are fixed by the system's clock which synchronizes all parts of the circuit. The 'real-time' constraint of any submodule of such a circuit is therefore that any signal must be propagated through the circuit and all signals have stabilized before the next clock cycle occurs. CAD tools for designing hardware circuits determine the critical path of a hardware circuit and can minimize its length so that the clock speed can be increased. Hence, hardware designs do always have to consider real-time constraints.

In contrast to that, the software design does not usually aim to construct reactive systems. However, there are new applications that make software obey given real-time constraints. For example, a distributed data base to manage flight reservations must be updated fast enough for all requests to be considered w.r.t. the actual state of the data base. Moreover, computers are used to control complex systems such as aircrafts or even nuclear plants, and must therefore be able to react on unforeseen situations before serious damages occur. For this reason it is usually important to use real-time operating systems that guarantee fixed time bounds for interrupting processes and executing interrupt routines. It is very important for the implementation of reactive systems that process management is done within guaranteed real-time constraints: Everybody who has ever erroneously deleted a directory on a Unix system that contains many files, and tried to abort the deletion process without success, will agree that Unix is certainly not a real-time operating system.

Special programming languages for the development of reactive systems have been suggested in the past decade, in particular the *synchronous languages* [230]. The main paradigm of these languages is the *perfect synchrony paradigm* that is most easily explained with the synchronous language Esterel [42, 43]. The perfect synchrony paradigm is achieved in Esterel as follows: By the Esterel semantics, almost all program statements do not require time for their execution. There is only one basic statement, namely the **pause** statement that does consume time, and this is actually all it does. Each time a **pause** statement is executed, one logical unit of time is consumed, which means that the control flow rests for one unit of time on that **pause** statement. Therefore, threads run synchronously to each other since they will always execute all statements between two **pause** statements in zero time and will then synchronize (by the semantics of Esterel) at the next **pause** statement.

Clearly, it is only an idealized view that the other Esterel statement can be executed without consuming time. However, the same idealization has been used for designing complex hardware circuits. There, the designer assumes that the signals propagate through a combinational circuit without

delays, although this is physically not possible. However, if the clock cycle is not too fast, this idealization is legal and liberates the designer from the burden to consider the circuits in a more detailed manner than necessary. The complexity of actual hardware designs proves that this idealization is reasonable. Berry, the main developer of the Esterel language, compares this idealization with the models that are used in mechanics [41]: Although we know that Newtonian mechanics is not precisely true and that it is only an approximation of the relativistic mechanics due to Einstein, most applications nowadays still use Newtonian models, simply because these are much more efficiently computed, and for most applications the results are precise enough.

After a (synchronous) program has been written, it can be translated to a finite state machine that controls the manipulation of the data variables. The states of this finite state machine correspond with the possible locations of the control flow in the program text. As the program text is finite, it follows that the control flow can always be modeled with finitely many states. However, if the values of the data variables are considered, the state set may become infinite when infinite data types come into play. In practice, however, there is no system that is able to really handle infinite data types, so that we usually have finite data types: Integers in most programming languages like C have a certain bitwidth and are therefore finite data types. For this reason, we can compile most programs to finite state machines, even if these have an astronomical number of states.

Synchronous languages have a nice property in that the concurrency of the programs can be handled at compile-time: due to the semantics all programs are deterministic, and therefore the interactions of the threads are known at compile-time. The languages moreover provide convenient statements for controlling the complex interactions of threads like different variants of abortion and suspension. Even if these programs are quite readable, the important properties that one is interested in the verification of reactive systems are nevertheless concerned with the correct temporal behavior. Particular problems are to prove the absence of deadlocks, absence of live-locks, mutual exclusion, and write clashes, and in general, that signals occur at the right time so that, for example, protocols are correctly implemented.

An important taxonomy of properties, as given below, is found in many books and papers, in particular in [351, 352]. It is based on both practical and theoretical observations. On the one hand, the properties below can often be found more or less directly in many specifications. On the other hand, many formalisms can be classified when they are either able to express these properties or not.

Safety properties state that for all computations of the system, and for all instances of time, some property will invariantly hold. For example, a safety property of a traffic light controller would be that at no point of time, the traffic lights of crossing streets will have a green light.

Liveness properties state that some desired state of the system can eventually be reached. There is no fixed bound of time given in which the state must be reached. For example, a liveness property could be that the initialization phase will definitely terminate and that the system will therefore become ready for computations.

Persistence properties are related to the stabilization of certain properties. In general, a persistence property describes that for all possible computations, there is a point of time when a certain property will always hold afterwards.

Fairness properties state that some property will infinitely often hold. The notions of fairness and liveness are often mixed up since, for example liveness properties in the theory of Petri nets means that the net is always alive, i.e., that there is always some progress. In our classification, this is however a fairness property. Manna and Pnueli call these properties *recurrence properties* [351, 352].

The notion of ‘fairness’ is derived from specifications, where these properties are used to state that no process is ignored infinitely often by an operating system that should schedule the processes on a processor. Several notions of fairness have been introduced such as strong fairness, unconditional fairness, and many others (see [175, 197] for a more detailed discussion of these fairness properties).

From a viewpoint of expressiveness, we will see in Section 4.6, that liveness, safety, and persistence properties can be reduced to equivalent fairness properties when appropriate observers are added. The above properties are used to define a hierarchy for ω -automata given in Section 4.6, which is then adapted to temporal logic in Section 5.4.3. We will moreover see that the first three classes can be translated to very simple fixpoint expressions, while the latter class requires a higher effort, since it requires mutually dependent nested fixpoints for its description. Hence, these properties can be used to structure many formalisms in a hierarchy. We will elaborate this issue in detail throughout this book.

Safety and liveness properties are of particular interest, even they can both be reduced to persistence and fairness properties when appropriate observers are used. One reason is simply that most specifications are safety properties, and another reason is that the verification of safety properties can be done by specialized verification procedures. For example, these properties can be verified by induction, so that no traversal of the state spaces is necessary. Therefore, safety properties can also be proved for infinite state spaces. For example, induction-less induction methods that are based on term rewriting can verify safety properties.

To summarize, we have now seen where and how formal methods can be applied in modern system design. Our main interest is the formal verification of already implemented systems. We have furthermore classified the systems in many ways, and the most relevant systems for this book are re-

active systems that can often be reduced to finite-state machines. Hence, we are interested in the formal verification of finite state systems. Moreover, we have already mentioned important classes of properties, without taking a specific formalism into account.

The kind of system clearly influences the properties to be specified, and therefore the choice of a suitable verification formalism. A plethora of formalisms has been developed, and still new formalisms are invented for different purposes. Therefore, we will consider in the next section some of the most popular ones and will, in particular, list how these formalisms have evolved over the time.

1.2 Genealogy of Formal Verification

In the previous section, we defined what classes of formal methods exist, and have seen the application of formal methods in specification, formal synthesis, and formal verification. Moreover, we have found reactive systems to be of particular interest for formal verification, since their design is an error-prone task due to the intensive use of multi-threading and the heterogeneous design which is often split into hardware and software. Moreover, reactive systems are well-suited to formal verification, since they can often be reduced to finite state systems so that the verification problem becomes decidable.

In this section, we consider approaches that are used for the formal verification of properties in a more detailed manner. In particular, we give some historical notes on the formalisms that are considered in detail throughout this book. Parts of the next section are based on [50], who themselves borrowed material from [140]. The other sections are a summary of other research papers of the corresponding formalisms.

1.2.1 Early Beginnings of Mathematical Logic

The idea that formal reasoning could be mechanized such that machines can generate mathematical proofs is an old dream that has its origins in the seventeenth century. *Déscartes* (1596-1650) developed an algebraic foundation of the ancient Euclidian geometry known from the Greek philosophers. Based on the introduction of coordinate systems, he was able to express all geometric problems by means of algebraic equations that are then solvable by purely algebraic means. *Déscartes* was aware of the fact that his ‘decision procedure’ could be mechanized in a similar way to the arithmetic computations have been mechanized by the calculators of *Schickard* (1592-1653) and *Pascal* (1623-1662). In his work, he wrote:

... it is possible to construct all the problems of ordinary geometry by doing no more than the little covered in the four figures that I have explained. This

is one thing which I believe the ancients did not notice, for otherwise they would not have put so much labor into writing so many books in which the very sequence of the propositions showed that they did not have a sure method of finding all ...

Leibniz (1646-1716) had an even more general vision: His aim was to do the same for all fields of mathematics and even more for any kind of human thinking. In a research project that he planned for the next three centuries (!), his aims were to develop a formal language called the '*lingua characteristica*', and to develop a corresponding calculus, called the '*calculus ratiocinator*'. The *lingua characteristica* would be powerful enough to express all kind of properties, and the *calculus ratiocinator* would provide laws that could be implemented – in modern words – in some sort of decision procedure, so that one could build a machine that would be able to derive any kind of truth.

However, the contribution of Leibniz to this project was not very promising, although his research in the differential calculus went in that direction. A first calculus in the sense of Leibniz was then developed by A. de Morgan and G. Boole (1815-1864) for propositional logic. Originally, Boolean algebra had been developed for the formalization of set theory. Boole himself viewed his work as a contribution to Leibniz's research programme. It is remarkable to mention that in 1869 S. Javins built a machine that was able to check Boolean expressions so that Leibniz's research programme had already been successful for propositional logic at that time.

The next step was then taken by Gottlob Frege (1848-1925). In his book [198] entitled '*Begriffsschrift*', he actually developed what we now call first order predicate logic. This logic is a convenient formalism that can express many interesting properties. In particular, it extends propositional logic by quantified formulas of the form $\exists x.\Phi$ and $\forall x.\Phi$ which express that the property Φ must hold for at least one or for all elements x . Atomic formulas are not only propositional constants, but may also depend on arguments, i.e., they may be of the form $p(\tau_1, \dots, \tau_n)$, where the τ_i 's are terms of the logic. These terms are recursively constructed of variables, constants and function applications which look like $f(\tau_1, \dots, \tau_n)$. Note however that $f(\tau_1, \dots, \tau_n)$ is interpreted as an element of the considered domain, while atomic formulas are interpreted to be either true or false. Additionally, equality \doteq is often added in that for two terms τ_1 and τ_2 , the expression $\tau_1 \doteq \tau_2$ is an atomic formula that expresses that the elements τ_1 and τ_2 are the same. The domain \mathcal{D} that is used for interpretation of the terms and formulas is thereby given by an interpretation of the logic. An interpretation must moreover provide a function \mathcal{I} that maps predicates p of arity n to a relation $\mathcal{I}(p) \subseteq \mathcal{D}^n$, and function symbols f of arity n to functions of type $\mathcal{D}^n \rightarrow \mathcal{D}$. In contrast to propositional logic, first order logic therefore allows reasoning about infinite domains.

Although Frege's work has not become very famous, his contribution is important when one considers the argumentations that were used at that

time. In particular, it was not clear at that time what the basic constructs and axioms of mathematics were. Therefore, many proofs were not sound since they are based on other things that were, in turn, based on the theorem that was to be proved. Frege's work mainly contributed to the formal language, i.e., *lingua characteristica*, although he made proofs by the 'modus ponens' which was his 'calculus ratiocinator'. Frege's work was moreover important because he was the first who distinguished between *syntax* and *semantics* of a formal language and therefore, his work is an early forerunner of the principles that are used nowadays in computer science.

G. Peano applied Frege's mathematical logic to other fields of science. He wrote:

... 'I think that the propositions of any science can be expressed by these signs alone, provided we add signs representing the objects of that science.'

Therefore, Peano's aim was to eliminate natural language for the precise formulation of properties in a precise formal manner. However, his reasoning about these properties was then done in natural language so that he ignored the initial work of Frege in terms of the 'calculus ratiocinator'.

Automated reasoning has however been controversially discussed from its beginning: For example, the famous mathematician H. Poincaré wrote:

Thus it will be readily understood that in order to present a theorem, it is no longer necessary or even useful to know what it means. We might replace geometry by the reasoning piano imagined by Stanley Javins; or if we prefer, we might imagine a machine where we could put in axioms at one end and take out theorems at the other, like that legendary machine in Chicago where pigs go in alive and come out transformed into hams and sausages. It is no more necessary for the mathematician than it is for these machines to know what he is doing.'

Nevertheless, it became clear that mathematics had to be put on a sound basis with a clearly defined set of axioms and inference rules. Whitehead and Russel have proved in their famous work, the 'Principia Mathematica', that such a formal foundation was possible for all known mathematics. At this time, the formalization of *higher order predicate logics*, or as it is sometimes called the *type theory*, started in order to circumvent antinomies like Russel's paradox (to construct the set of all sets which could not be correctly typed).

The classical mathematical logic was then quickly developed: Th. Skolem developed in [461] and [462] a systematic way to check the satisfiability of logical formulas. In [461] he developed his quantifier elimination method that became known as 'skolemization'. The essential idea is thereby that formulas of the form $\forall x. \exists y. \Phi(x, y)$ denote some dependency between y and x so that there must be a function f that maps any element x to an element y so that the relation $\Phi(x, y)$ holds. Hence, the formula $\forall x. \exists y. \Phi(x, y)$ is satisfiable if and only if the formula $\forall x. \Phi(x, f(x))$ is satisfiable (where f should not

already occur in Φ). The skolemization therefore eliminates positive occurrences of \exists quantifiers for the consideration of satisfiability. As a result, only universally quantified formulas need to be considered, which can be moreover brought into prenex normal form, so that only formulas of the form $\forall x_1 \dots \forall x_n. \Phi$ have to be considered, where Φ does not contain quantifiers at all.

In [462], Skolem constructed a ‘standard model’ for the universal fragment of first order predicate logic, namely the Herbrand universe, whose name is therefore not attributed to its inventor. The essential idea is that for any considerations of satisfiability, one can use the set of (variable-free) terms as domain \mathcal{D} , hence, interpreting terms by themselves, and predicates as relations between terms. Hence, Skolem proved that any satisfiable first order logic formula does also has a *countable* model, which means that uncountable domains such as the real numbers can not be characterized by first order logic.

In the same year, Hilbert and Ackermann [246] presented in their book ‘Grundzüge der theoretischen Logik’ an axiomatization of the first order predicate calculus and imposed two important questions: firstly, the completeness of this axiomatization, i.e., whether it is possible to derive any valid property with it, and secondly, its decidability, i.e., whether one can build a machine to deduce truth.

Both problems have been solved by Gödel: In 1930, he showed [211] that the calculus given by Hilbert and Ackermann was in fact complete, which was then a promising approach to automated reasoning. However, one year later, Gödel showed [212] that any formal system that is strong enough to express arithmetic can either be not complete, or it is not decidable whether a formula is an axiom of the system. In particular, he presented a construction of a formula in any such formal system that can neither be proved nor disproved. To this end, he encoded the notions of derivation in a calculus as first-order formulas and considered the formula that says ‘I am not derivable’. If this formula was valid, it could be used as a witness for a valid formula that can not be derived. If it was false, it must be derivable, but then the calculus is not correct since it derives the wrong formulas. Therefore any formalism that is powerful enough to formalize arithmetic with natural numbers is incomplete and therefore has ‘leaks’ (i.e., formulas that are neither true nor false). Some of the yet unsolved propositions of number theory such as Goldbach’s conjecture are supposed to be examples of such leaks (however the same had been thought about Fermat’s theorem until a proof was recently found).

The decidability problem for first order logic as raised by Hilbert and Ackermann was then independently solved by A. Turing and A. Church [99] in 1936 in that they showed the undecidability of the problem. While Turing reduced the problem to the termination problem of his Turing machines, Church found a similar reduction in the evaluation of λ -calculus expressions.

These negative results evidently destroyed the dream of implementing a ‘calculus ratiocinator’ as implied by Leibniz.

However, *Herbrand* already pointed out in his dissertation of 1930 that any valid sentence can be proved in finite time, i.e., we can define algorithms that are able to prove any valid formula of first-order predicate calculus. The essential property of first order logic that enables us to do this is the compactness property which states that a (possibly infinite) set of formulas is satisfiable iff each finite subset of it is satisfiable. In other words, a possibly infinite set of formulas is unsatisfiable iff there is a *finite subset* of it that is unsatisfiable. The objective of Herbrand’s proof procedure is therefore to replace quantified formulas $\forall x.\Phi$ by the set $\{[\Phi]_x^\tau \mid \mathcal{T}\}$ where \mathcal{T} is the set of all terms and $[\Phi]_x^\tau$ is obtained from Φ by replacing x by τ .

However, if the formula is satisfiable, the procedure may run into an infinite loop that encounters an infinite model and will therefore never terminate. The problem is that as long as the algorithm does not terminate, we know nothing about the truth value of the formula. As we have no upper bound on how long the algorithm will run, there will be no result found unless the algorithm successfully terminates.

1.2.2 Automated Theorem Proving

The rapid development of the first computers started a new kind of research, namely *automated theorem proving*. It now became possible to build, in a much simpler way than ever before, a ‘calculus ratiocinator’, simply by programming a computer. The first program in that direction, a decision procedure due to Presburger for his arithmetic [415], was implemented in 1954 by *M. Davis* on a Johniac, a ‘Röhrencomputer’. A big success was achieved by the proof of the fact that the sum of two even numbers is again an even number. This was the first proof made by a machine.

Based on more powerful computers, better implementations followed: *H. Wang* developed in 1958 at IBM, and afterwards from 1959-1964 at Bell-Labs, an automated theorem prover that was able to prove 350 theorems of the ‘Principia Mathematica’ (quite simple laws of predicate calculus with equality). In 1960-1962, *Martin Davis* and *Hillary Putnam* presented a new proof procedure which was split into two parts [141]: firstly, a part that instantiates the quantified formulas in a systematic way (by terms of the Herbrand universe), and a second part for efficient evaluation of propositional logic. The latter part is still in use, but the former part was more important in the 1960s.

In 1960, *Prawitz* [414] recognized that the enumeration of all terms of the Herbrand universe was not reasonable, and therefore developed an algorithm for computing only the relevant instantiations by a process that we call *unification*. His idea was implemented by Davis, McIlroy and others. In 1965, *Robinson* [425] integrated Prawitz’s unification algorithm in a single deduction rule, namely the resolution principle. In the meantime, a lot of refine-

ments of resolution calculus such as hyper-resolution or theory-resolution [470] have been developed and implemented in efficient theorem provers like Otter, Setheo, or SPASS. Moreover, other representations like clausal graphs, the connection graphs of Kowalski [287], the matings of *P. Andrews* [16], or the connection method [48] of *Bibel* have been developed for the implementation of more efficient theorem provers.

The field of automated theorem proving is still a major topic of research. Many refinements have been added and many new calculi and specialized logics have been invented. For example, tableau calculi [195, 464] are a promising way to construct automated theorem provers for special kinds of logics, such as modal logics [98, 255, 292]. The relationship between tableau and the more frequently used resolution calculi is outlined in detail by *Fitting* in [195]. Eder considers in [150] the complexity of different calculi, and d'Agostino [134] points out some improvements for tableau calculi so that theorem provers based on them may reach the same efficiency as the resolution-based ones.

Nearly all automated theorem provers are designed for first order predicate logic. We have already mentioned that the compactness property of first order logic leads directly to a semi-decision procedure for the logic. However, as there are sound and complete calculi for first order logic, it follows by Gödel's result that first order logic can not express Peano's arithmetic. As any abstract data type such as lists, can be used to encode numbers, it moreover follows that first-order logic can not characterize such abstract data types (up to isomorphisms between the interpretations).

For this reason, extensions of the first-order logic have been investigated that extend the logic by means of induction principles. The most popular provers for first order logic (with induction) are ACL2 [277], Eves [132], LP [205], Nqthm [60], Reve [322], and RRL [276]. In general, these fall into the two classes of *explicit* and *implicit* induction provers. Explicit induction provers like [25, 60, 61, 92, 206] use induction rules as explicitly given proof rules, e.g., for tableau construction. These induction rules are based on a well-founded ordering of the terms and require that the induction step transfers results from smaller terms to larger ones.

Implicit induction calculi, also known as inductionless induction, are based on term rewriting and have been developed since the eighties [59, 199, 254, 267, 268, 385]. These calculi are based on the consideration that an equation γ is a consequence of a set of equations Γ if and only if the same set of (variable-free) formulas can be derived from both Γ and $\Gamma \cup \{\gamma\}$. The restriction to equation systems is not as severe, as it might look at a first glance: any proposition can be written as an equation of the form $\Phi = \text{true}$. Moreover, there are extensions [144, 275, 286] that consider conditional equation systems. The advantage of the inductionless induction methods is that they are able to automatically deduce all lemmas that are required for the proof, while in explicit induction provers, the user has to manually set up the ap-

appropriate lemmas. Note that the induction often fails for valid formulas and must then be applied to a stronger property.

Another, and even more powerful extension of first order logic is obtained by *higher order logics*. While first order logics only quantify over elements of the considered domain \mathcal{D} , higher order logics additionally quantify over sets of such elements, and functions of such elements. These additional extensions enhance the expressiveness of the logics so that all facts of mathematics can be expressed. For example, Peano's axioms for the natural numbers can be easily expressed in higher order logic, and hence, the natural numbers can be formalized with this formalism.

Clearly, this means by Gödel's result that there is no complete calculus for this logic. Therefore, it is hardly possible that automated proof procedures can be implemented for it (Andrews et. al. [18] have implemented an automated theorem prover for higher order logic, and Kerber showed in [279, 280] how simple properties of higher order logics can be proved by means of a first order theorem prover). Hence, theorem provers for higher order logic are usually proof assistants, where the user has to manually invoke proof steps that are then checked by the system. The system has therefore only the task of book-keeping subgoals, checking the applicability of the rules, and clearly to generate the subgoals by applying the rules.

In particular, theorem provers for higher order logic are interesting for the formal verification of systems. In fact, one of the main applications of these theorem provers is verification. The most popular higher order logic theorem provers that are used for this purpose are HOL [216], PVS [396], Coq [125], Veritas [233], Nuprl [123], and Isabelle [398]. These provers differ in the kind of higher order logic they use, e.g., PVS uses dependent types², while HOL is based on simple types. Nuprl is based on intuitionistic logic, while the others are based on classic logic. Moreover, the theorem provers differ in the comfort they provide. In particular, PVS and HOL provide a rich set of efficient decision procedures including Presburger arithmetic, propositional logic, and also proof procedures for first order logic.

Both induction provers, and theorem provers for higher order logics, in particular HOL and PVS, have been used to verify different kinds of systems. For example, in 1986 Camilleri, Gordon and Melham verified a n -bit broad CMOS adder and a sequential switch level circuit for the computation of $n!$ [94]. Herbert formalized delay times in combinational hardware circuits [245] in 1988, as well as a network chip in ECL logic. Gordon verified a n -bit sequential multiplier and many other circuits [215] around the mid-eighties. Kumar, Schneider, and Kropf presented, in 1991, a structured approach for the verification of register-transfer circuits in HOL [298, 440, 441]. HOL has also been used to show the correctness of microprocessors like TAMARACK-

² In simple type theory, the types are defined independently of the terms. In dependent type theory, the set of types and terms are mutually recursive. Therefore, it is in general not decidable whether a term is correctly typed.

1 [269], Viper [121, 133, 517], and DLX [480]. PVS has also been used for the same purpose, e.g., the processor AAMP5 has been verified by Miller and Srivas [368]. Nqthm has been used by Hunt [256] to verify the 16-bit microprocessor FM8501 whose complexity is comparable with a PDP-11. The work has been extended to the verification of the 32-bit processor FM8502 that has also a more powerful instruction set [257]. The verification of the FM8502 was part of a larger verification project where also a code generator, an assembler, and even a kernel of an operating system were verified [45].

1.2.3 Beginnings of Program Verification

In the so-far mentioned work, we mainly considered pure theorem proving for first and higher order predicate calculus that had been successfully applied to the verification of some systems, but that is still not specialized, neither in terms of the formal language nor in terms of the proof methods. As there was however an early interest in the verification of computer systems by mathematical proofs, specialized logics and proof procedures were invented in the late 1960s. Thus, a new field for automated reasoning, namely the *verification of computer programs and systems* was born.

The earliest work in this area probably stems from *Floyd* [196] and *Hoare* [248]. They proposed guarantee commitment style proof rules for computer programs: Given that Φ and Ψ were formulas of some predicate logic, and P is a program statement, then the ‘Hoare triple’ $\{\Phi\}P\{\Psi\}$ means that: if Φ holds when the program P is started, and P terminates, then the property Ψ holds. In this ‘Hoare triple’ the condition Φ is called the precondition and Ψ is called the postcondition. The most interesting rule of Hoare’s calculus is the one for the verification of loops. The rule is based on so-called invariants Φ_I and is as follows:

$$\frac{\Phi \rightarrow \Phi_I \quad \{\Phi_I \wedge B\}P\{\Phi_I\} \quad \Phi_I \wedge \neg B \rightarrow \Psi}{\{\Phi\}\mathbf{while} \ B \ \mathbf{do} \ P \ \mathbf{end}\{\Psi\}}$$

If a specification Ψ is to be shown for the loop **while** B **do** P **end**, where we can assume the precondition Φ , we must first search a suitable invariant Φ_I so that we can apply the above rule. The remaining problem is then to prove the propositions that are given above the line. In particular, the condition $\{\Phi_I \wedge B\}P\{\Phi_I\}$ amounts to saying that Φ_I is an invariant of P : if it holds before the execution of P , it will also remain true after the termination of P (it may however be false during the execution of P). The problem is that invariants are not always detected easily so that the conditions above the line hold.

For this reason, some investigations started to compute or approximate invariants. In particular, *Dijkstra* [145] suggested to compute weakest preconditions for a specification Ψ and a given program P . Informally, the weakest precondition $wp(P, \Psi)$ satisfies the Hoare triple $\{wp(P, \Psi)\}P\{\Psi\}$, and for any

other precondition Φ that satisfies $\{\Phi\}P\{\Psi\}$, we have $\Phi \rightarrow wp(P, \Psi)$. Weakest preconditions can be used to give a fixpoint characterization of invariants. We will consider these relationships in the next paragraph, and in more detail in Section 3.8.3. Moreover, we note that Owicki and Gries extended these proof methods for safety properties to deal with concurrency in [394, 395].

1.2.4 Dynamic Logics and Fixpoint Calculi

Dynamic logics, also called program logics, due to Pratt and Harel [235, 411] can be used as a formal foundation of both Hoare's calculus and Dijkstra's weakest preconditions. Both can be defined in dynamic logic, and the rules revealed by Hoare and Dijkstra can then be proved.

Dynamic logics are special cases of modal logics [98, 255, 292] which have been developed in philosophy. In modal logics, the truth values of the atomic propositions of the logic are no longer fixed by a particular interpretation as in the previously mentioned predicate logics. Instead, it depends on a current state, which is often called the 'current world'. Moreover, an interpretation of modal logic determines what the next state of a state could be. Models of modal logics are therefore Kripke structures $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \xi)$ where \mathcal{S} is the set of possible states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ determines the transition relation, and ξ is finally a function that maps any state $s \in \mathcal{S}$ to an interpretation of the atomic formulas. $(s, s') \in \mathcal{R}$ means that the state s' can be reached from the set s , i.e., it is a possible successor state of s . There may be more than one successor state, and sometimes a condition α may be given to select a particular next state. This roughly explains the semantic models for modal logics. Several modal logics are distinguishable in that one often gives additional requirements on the transition relation, for example, that it must be symmetric or transitive.

For the syntax of dynamic logics, there are two important additional operators for modal logics: $\langle \alpha \rangle \Psi$ holds in a state s if this state has a successor state s' that can be reached from s with the condition α such that the formula Ψ holds in s' . The dual operator $[\alpha] \Psi$, defined as $[\alpha] \Psi \equiv \neg \langle \alpha \rangle \neg \Psi$, states that all successor states s' that can be reached under the condition α must satisfy Ψ .

It is quite clear, how modal logics and program logics are related to each other: States of a Kripke structure \mathcal{K}_P can be used to model different stages of the computation of a program P . The transition relation \mathcal{R}_P can be derived with the semantics of the programming language used to implement the program P . Hence, by means of the programming language's semantics, we can transform every program into a corresponding Kripke structure (e.g. see [435, 436, 438] to derive a Kripke structure from Esterel programs). Therefore, modal logics are particularly well-suited for reasoning about the before-after behavior of programs, when we consider all the situations that could occur during the execution of a program.

In dynamic logics, the modal operators $\langle \alpha \rangle \Psi$ and $[\alpha] \Psi$ are extended in that the conditions α need not necessarily be atomic program statements (this distinguishes them from *Hennessey-Milner* logic [239]). Instead, they may be entire program statements P , and in this case, the formula $\langle P \rangle \Psi$ intuitively holds in a particular state s if the program statement P can be executed from this state so that after termination of P a state s' is reached where Ψ holds. Note that several transitions can be taken during the execution of P . In a similar way, the formula $[P] \Psi$ holds iff for all executions of the program P the property Ψ must hold after termination (P may be nondeterministic or its execution may depend on the inputs). If no terminating execution of P can be started in this state, the formula does trivially hold. Usually, dynamic logics consider some set of program constructs for determining the statements that are allowed in the formulas. A detailed definition is given in Section 3.8.3.

Using dynamic logics, Hoare triples $\{\Phi\}P\{\Psi\}$ can be defined as an abbreviation of the formula $\Phi \rightarrow [P]\Psi$. Therefore, Hoare's rules can also be given as formulas in dynamic logic and therefore the correctness of the rules can be proved. For example, the above mentioned rule for loops is translated to the following valid dynamic logic formula:

$$\left(\begin{array}{l} \Phi \wedge \\ (\Phi \rightarrow \Phi_I) \wedge \\ (\Phi_I \wedge B \rightarrow [P]\Phi_I) \wedge \\ (\Phi_I \wedge \neg B \rightarrow \Psi) \end{array} \right) \rightarrow [\mathbf{while } B \mathbf{ do } P \mathbf{ end}] \Psi$$

Also the correctness of the rules for the computation of weakest preconditions can be proved. In particular, it turns out that the weakest preconditions of loops can be defined as greatest fixpoints:

$$wp(\mathbf{while } B \mathbf{ do } S \mathbf{ end}, \Psi) := \nu x. (\neg B \wedge \Psi) \vee (B \wedge wp(S, x))$$

Thus, the expression $\nu x. \Phi$ denotes the greatest fixpoint of Φ . To explain what this means in our setting, recall that the models are Kripke structures and that the formulas are interpreted on states of the Kripke structure. Therefore, we can compute for any formula Φ and any Kripke structure \mathcal{K} the set of states $\llbracket \Phi \rrbracket_{\mathcal{K}}$ of \mathcal{K} where Φ holds. If we change the structure \mathcal{K} to \mathcal{K}_x^S such that the variable x holds in the states $S \subseteq \mathcal{S}$, we obtain a so-called *state transformer* by defining $f(S) := \llbracket \Phi \rrbracket_{\mathcal{K}_x^S}$. Note that f is a function of type $f : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$. $S \subseteq \mathcal{S}$ a fixpoint of this state transformer iff the condition $S = f(S)$ holds. As the set of all subsets $2^{\mathcal{S}}$ is ordered by the set inclusion, we can furthermore talk about least and greatest fixpoints, and denote these as $\mu x. \Phi$ and $\nu x. \Phi$, respectively.

These observations showed the usefulness of fixpoint expressions and stimulated the construction of fixpoint logics. In 1975, *Kfoury and Park* proved that properties like termination and totality of programs can not be expressed in first order logics [283] (see Section 6.2.2). For this reason, *Park* [397], *Hitchcock and Park* [247], and *de Bakker and de Roever* [142] introduced a least fixpoint operator to remedy these deficiencies. *Emerson* and *Clarke* showed in

1980 [160] how fixpoints can be used to formulate correctness properties of parallel programs. The resulting formal systems were powerful enough to express properties like termination, liveness, and freedom from deadlocks or starvation. Based on these forerunners, Pratt [413] and Kozen [288, 289] have developed the propositional μ -calculus. This formalism is discussed in detail in Chapter 3 of this book and we will see that it is a very powerful basis for all other formalisms in this book. In particular, we will see in Section 3.8.3 that dynamic logics can be easily defined in terms of the μ -calculus.

In the early 1980s, it was not immediately clear, whether the propositional μ -calculus was decidable. It should be noted that the compactness property that is the reason for the semi-decidability of first order logic does neither hold for propositional dynamic logic nor for the propositional μ -calculus. In 1983-1984, Kozen [288, 289], Vardi and Wolper [499] developed exponential time decision procedures for fragments of Kozen's μ -calculus. In 1983, Kozen and Parikh [290] and independently, Niwiński showed that the satisfiability problem of μ -calculus formulas can be reduced to the second order theory of n successors (SnS). By Rabin's results [418] (Rabin has proved the decidability of SnS), a first decision procedure was obtained for the propositional μ -calculus. However, like Streett and Emerson's decision procedure of 1984 [477], its runtime complexity was nonelementary, which means that the runtime can not be bound by a finite number of exponential nestings. Better decision procedures were found afterwards: Streett and Emerson found, in 1989 [478], a triple exponential time decision procedure for the propositional μ -calculus. Based on further improvements for (1) determinizing ω -automata in exponential time by Safra's construction (cf. Definition 4.29), and (2) checking the emptiness of ω -automata on trees in exponential time [165, 168, 304, 409] this was reduced to a single exponential time decision procedure (cf. Appendix B.3). The decidability and complexity of the extension to past time modalities was finally shown by Vardi [430, 496] in 1998.

Beneath the test for satisfiability, the evaluation of a given μ -calculus formula in a given Kripke structure is of interest, i.e., the *model checking problem of the μ -calculus*. In particular, the verification problem can be reduced to such model checking problems. In general, one distinguishes between local and global approaches. Global approaches aim at computing the set of states where the formula holds, while local approaches only try to check whether the formula holds in a particular state or not. In general, local model checking approaches may be more efficient than global ones, since the validity in a given state can possibly be answered by considering only a part of the entire state set. In the worst case however, the same complexity is reached as in global approaches. Global approaches, on the other hand, have the advantage that they can make use of a breadth-first search which can be very efficiently implemented by means of symbolic traversals with binary decision diagrams [44, 88]. Therefore, global approaches can cope with very large structures, i.e., structures with up to 10^{100} states.

A first model checking procedure, given by Emerson and Lei in 1986 [174], runs in time $O\left(|\mathcal{K}|(|\mathcal{S}||\Phi|)^{\text{ad}(\Phi)+1}\right)$, where $|\mathcal{S}|$ is the number of states, $|\mathcal{K}|$ is the size of the Kripke structure (maximum of transitions and states), and $\text{ad}(\Phi)$ is the number of alternating nestings of fixpoint operators (see page 116). In 1991, Cleaveland and Steffen [118, 119] presented a sophisticated model checking procedure for the alternation free μ -calculus (this is the set of μ -calculus formulas with $\text{ad}(\Phi) \leq 1$) that runs in time $O(|\mathcal{K}||\Phi|)$ and therefore improved the previous result. Therefore, the alternation-free μ -calculus is a promising language for the automated verification since it has very efficient model checking procedures. Cleaveland, Klein, and Steffen extended their model checking procedure afterwards to formulas of arbitrary alternation depth and obtained a runtime of order $O\left(\left(\frac{1}{d}|\mathcal{S}||\Phi|\right)^{d-1}|\mathcal{K}||\Phi|\right)$ [116] with $d := \text{ad}(\Phi)$.

In 1993, Emerson, Jutla and Sistla [169] presented some more fragments of the μ -calculus that can be checked in time $O(|\mathcal{K}||\Phi|^2)$, i.e., linear in terms of the system's size and quadratic in terms of the length of the formula. As the system's size is usually the limiting factor, this is still an acceptable result. In 1994, Long, Browne, Clarke, Jha, and Marrero showed that the previous results on μ -calculus model checking can be improved to $O\left((|\mathcal{S}||\Phi|)^{\frac{d}{2}}|\mathcal{K}||\Phi|\right)$ [333]. We will give proofs for these results in Chapter 3.

Local model checking procedures (see Appendix B) for the μ -calculus have been given by Stirling and Walker [471, 472] (1989-1991), Cleaveland [115] (1989), and Bradfield and Stirling [67] (1992). Bradfield [63] also discusses model checking procedures for infinite state spaces.

The μ -calculus is of particular interest for the complexity theory, since the model checking problem of the μ -calculus is known to be in NP and also in the complement coNP [168–170], as shown by Emerson, Jutla and Sistla in 1993. This means that there is a nondeterministic procedure that can check μ -calculus formulas in polynomial time. There are very few problems that belong to $\text{NP} \cap \text{coNP}$ for which no polynomial deterministic procedure is known. The model checking problem for the μ -calculus is one of them. The result was even refined to membership in $\text{UP} \cap \text{coUP}$ by Jurdziński in 1998 [270].

Another question related to the alternation depth was independently solved by Bradfield [64, 65] and Lenzi [321] in 1996. They proved that the alternation hierarchy of the μ -calculus is strict, i.e., for every number n , there are μ -calculus formulas of alternation depth n which can not be expressed by μ -calculus formulas of lower alternation depth. All currently known model checking procedures are, however, exponential in the alternation depth.

The research on efficient decision procedures for the μ -calculus is still not complete. In particular, it is not clear whether a polynomial model checking procedure exists or not, although the existence of a polynomial algorithm is unlikely. Moreover, efficient decision procedures for the μ -calculus are im-

portant, since we will see throughout the book that most formalisms can be translated to the μ -calculus.

1.2.5 Temporal Logics

μ -calculus formulas are quite hard to understand, in particular when interdependent fixpoints are nested. For this reason, there is a need for more readable specification languages. Special modal logics have been developed with the view that the change from one state to another requires one unit of time by Prior in 1957 [416, 417]. Therefore, the resulting logics are called *temporal logics*. Beneath the simple next-time operator, which we discussed in the previous section, temporal logics provide further temporal operators for describing complex temporal properties. The most important temporal operator is the \underline{U} operator that has the following meaning: $[\varphi \underline{U} \psi]$ holds if there is a point of time in the future where ψ holds and up to this point the condition φ holds. Other temporal operators are G and F with the following meaning: $G\varphi$ states that φ must always hold, and $F\varphi$ states that φ must hold at least once. The next time operator is usually written as X , so that $X\varphi$ means that φ holds at the next point of time. Note here that all mentioned temporal operators do refer to a particular computation of a system, i.e., to a *path through the Kripke structure*.

From the viewpoint of temporal operators, Kamp [274] used, in 1968, a combined form of the \underline{U} and the X operator, which we denote as \underline{XU} . The operator is defined as $[\varphi \underline{XU} \psi] := X[\varphi \underline{U} \psi]$, i.e., it behaves like \underline{U} except that the present point of time is ignored. Moreover, Kamp considered the corresponding past-time operator: $[\varphi \overleftarrow{XU} \psi]$ holds iff there was a point of time in the past, where ψ held and since then φ holds up to this point. Kamp proved that \underline{XU} is able to express all previously mentioned future temporal operators and is strictly more expressive than the temporal operators G and F (see page 289). Kamp also proved that his temporal logic is as expressive as the first order theory of linear order (see Section 6.4). The same was proved [203] in 1980 for the future time fragment of the logic by Gabbay, Pnueli, Shelah and Stavi so that past time operators have been viewed as unnecessary since then. However, in 1985, Lichtenstein, Pnueli, and Zuck [326] reintroduced past operators for reasons of clear and uniform specifications. Laroussinie and Schnoebelen, also favored past operators for sake of simple and succinct specifications [315, 316, 355, 443] (for example: ‘when an accident happens, then a mistake is made before hand’). Further operators, such as the X operator, the precede operator, and the weak U operator were introduced by Manna and Pnueli in 1979 and 1982 [343, 345, 346]. We will discuss these operators in Chapter 5.

We have already remarked that temporal operators consider a particular computation of a system, i.e., a path through a Kripke structure. In 1980, Emerson and Clarke [160], and in 1981, Ben-Ari, Manna and Pnueli [36] explicitly used, for the first time, the path quantifiers E and A to quantify over

computation paths of a particular state. Hence, $E\Phi$ holds in a state of a Kripke structure iff at least one infinite computation path starts in that state that satisfies Φ . Analogously $A\Phi$ holds in a state iff all infinite computation paths that start in that state satisfy Φ . If a temporal logic provides these path quantifiers, one usually says that the temporal logic is a branching time temporal logic, otherwise it is called a linear time temporal logic. This is motivated by the fact that the use of a path quantifier allows branching to another path when the formula is evaluated along a certain path. Hence, these formulas can state properties about the ‘branching behavior’ of a system, which is not possible for linear time temporal logics. The linear time temporal logic with the temporal operators X and \underline{U} is usually denoted as LTL (we will call it LTL_p). By Kamp’s result, the temporal operators G and F , and many more can be defined, since this set of operators is already expressively complete w.r.t. the monadic first order theory of linear orders $MFO_{<}$. We prove this in Section 6.4.

The branching time temporal logic given by Ben-Ari, Manna and Pnueli in 1981 was called UB and is discussed in Chapter 5. In the same year, Emerson and Clarke [160, 161] pointed out that some forms of temporal logic formulas can be directly interpreted as fixpoint formulas of μ -calculus and therefore introduced the temporal logic CTL as a macro language of a fragment of the alternation-free μ -calculus. CTL consists of macro operators that are formed by coupling path quantifiers and temporal operators in pairs. This however makes it hard to express complex temporal properties for one computation path since each temporal operator must be preceded by a path quantifier that has the freedom to choose a new path for its evaluation.

The following decade saw an extensive debate [163, 172, 310, 324, 497] on whether the branching time logic CTL or the linear time temporal logic LTL is more suited for the specification and verification of finite state systems. In general, LTL specifications tend to be more readable than CTL specifications, since LTL directly allows the formalization of properties with more than one event, as temporal operators may be nested arbitrarily. For example, the property ‘ a has to hold until b holds the second time’ can be expressed in LTL as $[a \underline{U} (a \wedge b \wedge X[a \underline{U} b])]$. In 1985, Lichtenstein and Pnueli presented [324] a model checking procedure for temporal logic that runs in time $O(|K| 2^{c|\Phi|})$ for some constant $c > 0$. They argued that specifications Φ are rather short in comparison to the size of the model and that therefore their decision procedure is of practical use. In 1986, Clarke, Emerson, and Sistla gave a model checking procedure for CTL that runs in time $O(|K| |\Phi|)$ [103] which was a strong argument for the use of CTL.

However, this good theoretical result need not necessarily have led to verification tools that were able to verify large systems. In fact, first implementations were only able to handle systems with a thousand states. The breakthrough of the CTL model checking procedures was achieved with the development of efficient data structures for symbolically traversing the structure. These *symbolic model checking procedures* are based on manipulating Boolean

functions that are stored as binary decision diagrams (BDDs) [74] (cf. Appendix A). The application of these data structures in the model checking procedures is as follows: The states of the Kripke structures are encoded by a couple of state variables q_1, \dots, q_n , and hence, the transition relation can be described as a Boolean formula in q_1, \dots, q_n and corresponding variables q'_1, \dots, q'_n for the next state. This description of the structure can then be efficiently stored as a BDD, as can the sets of states of the structure. As the evaluation of the temporal logic formula simply consists of some fixpoint iterations over such sets of states, the entire model checking procedure can be implemented with BDDs. Note that a structure with n state variables may have 2^n reachable states. Hence, small propositional formulas can encode large structures. In particular, note that if all the 2^n states were connected with each other, the transition relation would be simply represented by the formula 1 (denoting truth). The same holds for sets of states.

Symbolic model checking was introduced by Burch, Clarke, McMillan, Dill, Hwang in 1990 [87–89], and independently in the same year by Berthet, Coudert, and Madre [44]. The use of symbolic model checking had finally led to the breakthrough in the verification of finite state systems and allowed the checking of systems with more than 10^{20} states. For this reason, symbolic CTL model checking still plays a dominant role, and verification tools such as SMV [360] and VIS [69] have successfully verified lots of systems, including the alternating bit protocol [103], traffic light controllers [71], DMA controllers [360], the Gigamax cache coherence protocol [360], and the Futurebus cache coherence protocol [106].

A major drawback of CTL is however its limited expressiveness: In particular, CTL can not express *fairness*. Therefore, extensions of CTL have been defined which augment the temporal operators of CTL by additional fairness constraints. In [103], it was shown that model checking procedures of polynomial runtime can be obtained for the extension of fairness constraints, and the procedures would still require polynomial runtime in terms of the size of the structure.

Nevertheless, the expressiveness, and even worse, the readability of CTL is not satisfactory. Therefore, the search continued for more powerful temporal logics that could be efficiently checked by symbolic model checking. Emerson and Halpern therefore introduced the temporal logic CTL* in [163] as a superset of CTL and LTL, in that path quantifiers *may* be applied to every subformula. Emerson and Lei [172, 176] then showed that any model checking procedure for LTL can be transformed to a model checking procedure for CTL* with roughly the same complexity. Therefore, there is no additional cost when path quantifiers are added to LTL. Expressivenesses and complexities of several temporal logics were investigated by Emerson and Halpern [162] in 1985, and, in particular for CTL* by Emerson and Jutla in [165]. It turned out that both LTL and CTL* model checking problems are PSPACE-complete. Moreover, neither LTL nor CTL* can be directly checked by symbolic model

checking (however, as we will discuss below, both can be translated to the μ -calculus which can be checked by symbolic model checking).

Recently, some new results have been added to the discussion. In 1994, Bernholtz and Grumberg presented a temporal logic CTL^2 that allows two nested temporal operators after a path quantifier. They showed that CTL^2 can still be checked in polynomial time. In 1997, Schneider presented the temporal logic LeftCTL^* that allows arbitrary deep nesting of temporal operators on one argument of the binary temporal operators without necessarily using path quantifiers. LeftCTL^* can be translated to the alternation-free μ -calculus, and in particular, to CTL. The size of the obtained CTL formula for a given LeftCTL^* formula Φ is thereby bound by $O(|\Phi| 2^{2|\Phi|})$, so that a LeftCTL^* model checking procedure with runtime $O(|\mathcal{K}| |\Phi| 2^{2|\Phi|})$ is obtained. Note that the blow-up shows that there are LeftCTL^* formulas that are exponentially more succinct than corresponding CTL formulas, and are therefore more readable.

In fact, from a result of Wilke [513], it follows that at least an exponential lower bound is required for this translation. Adler and Immerman [4] improved this result in 2001 to a lower bound $|\varphi|!$. In fact, the translation of LeftCTL^* to exponentially sized ‘CTL’ formulas given in Section 5.3.5 translates the formulas to CTL formulas where common subterms are shared. For this reason only a blow-up $O(|\Phi| 2^{2|\Phi|})$ is sufficient. Recently, Johannsen and Lange [266] proved that satisfiability checking of CTL^+ is 2EXPTIME-complete. As satisfiability checking of CTL is only EXPTIME-complete (as well as the sat-problem of the μ -calculus), it follows that every translation from CTL^+ to CTL (or to the μ -calculus) must necessarily have an exponential blow-up.

LeftCTL^* specifications tend to be as readable as LTL specifications (for the same reasons), but can be automatically translated to CTL, so that one benefits from the already existing efficient model checking tools for CTL. As an example, consider the LeftCTL^* formula $E[Fa \wedge FGb]$ that states that there is a path where a holds at least once and after some time b always holds. This formula is translated to the following equivalent, but less readable CTL formula:

$$EF[a \wedge EFEGb] \vee EF[b \underline{\cup} (a \wedge EGb)]$$

We will consider the translation from LeftCTL^* to CTL in detail in Section 5.3.3 and 5.3.5. The expressiveness of LeftCTL^* is the same as the expressiveness of the well-known logic CTL, which suffices for most applications. Nevertheless, we will also consider some extensions of LeftCTL^* . In particular, we will define a logic AFCTL^* in Section 5.4.5. AFCTL^* is strictly stronger than LeftCTL^* and CTL, but can still be translated to the alternation-free μ -calculus.

The translations of AFCTL^* and LeftCTL^* to the μ -calculus are very different. While we will translate LeftCTL^* directly to the alternation-free μ -calculus (Section 5.3.5), our translation of AFCTL^* is indirect via a transla-

tion to ω -automata (see Chapter 4 and Section 5.4). For this reason, the logic LeftCTL^* is still interesting on its own. The construction of the logic AFCTL^* is related to a result of Kupferman and Vardi [303] who characterized the intersection between the alternation-free μ -calculus and LTL. We will outline the relationship in Section 5.4.5. Therefore, the definition of the temporal logics LeftCTL^* and AFCTL^* is motivated by the well-known hierarchies of ω -automata and μ -calculus. It is thus of crucial importance that a good compromise between the expressiveness and the efficiency of the logics is found. In general, the stronger the expressiveness of a logic, the higher the complexity of its verification procedures.

In general, the expressiveness of temporal logics is well understood (cf. Theorem 5.76 on page 382). Kamp has already proved in 1968 that temporal logic is expressively complete w.r.t. the monadic first order theories of linear order $\text{MFO}_{<}$. In 1971, McNaughton and Papert [364] characterized the corresponding subset of ω -automata whose expressiveness matches that of temporal logics and also introduced star-free regular expressions for the characterization of this class. As temporal logics are less expressive than ω -automata, there are properties that can not be expressed by temporal logics, but with ω -automata. As an example, Wolper [519] mentioned the property that a variable a should hold for *at least* all even points of time. As these properties can, however, be expressed as μ -calculus formulas and also as ω -automata, Wolper suggested to extend temporal logic by operators based on regular languages [523]. Wolper also proposed a deductive proof system for his logic, which was shown to be complete, after some corrections, by Banieqbal and Barringer in 1986 [28]. The branching time analog was considered by Vardi and Wolper [500] in 1984. Wolper's extension had also some relationship to quantified temporal logic which was considered in [460, 519], and its complexity has been analyzed by Sistla, Vardi and Wolper in 1987 [459].

We conclude this section with some further historical remarks and a listing of temporal logics that are not considered in this book. First applications of temporal logics for the specification of computer systems were given in 1977 by Kröger [294] and Pnueli [407], who used temporal logics for the specification of sequential and concurrent program properties, respectively. Other authors that used temporal logics around the early 1980s as specification language were Hailpern [228], Owicki [229], and Lamport [311], and Manna and Pnueli [344, 346, 347]. Bochmann [53] was probably the first who specified the behavior of hardware circuits by temporal logics. Using temporal logic, Malachi and Owicki specified 'self-timed systems' [340] in 1981, and Manna and Wolper described the synchronization of communication processes by temporal logics in 1984 [353]. There are many more of these early applications that are not listed.

New directions of research mainly consider the state-explosion problem, which roughly means that the number of states that must be considered for verification grows exponentially with the size of the system. Promising results has been obtained by (data) abstraction (see Appendix C.4) by Clarke,

Grumberg, and Long [108, 109, 332] and also by Graf, Loiseaux, Sifakis, Bouajjani, and Bensalem [222, 331]. Other methods of solving the state explosion problem have been considered in the previous decade. Among them are reduction by exploiting symmetry of the structures [113, 178, 180, 181, 260] (see Appendix C.5), partial order reductions [400] that abstract away from different interleavings of asynchronous threads, on-the-fly methods [208], as well as combinations of these approaches [164].

So far, we have only discussed temporal logics whose semantics are based on discrete Kripke structures. However, there are also temporal logics that have different semantics. We will not consider these logics in this book, but list at least some of them to conclude this section. For example, there are real-time temporal logics which are either based on a continuous time model [9, 30, 90, 91] or on a discrete time model where transitions may require more than one unit of time [10, 11, 95, 238, 243, 316, 329, 330, 392, 427]. Furthermore, there are other variants such as first order temporal logics [1, 345], partial order temporal logics [406], and interval temporal logics [377, 450].

The semantics of the temporal interval logic ITL [232] maps sequences of states to variables, so that the truth value of a variable does not simply depend on a state of the Kripke structure, but on a sequence of states. The important operator of ITL is the chop operator ‘;’ that has the following meaning: $\varphi; \psi$ holds on a sequence of states s_0, \dots, s_n iff this sequence can be split into two parts s_0, \dots, s_i and s_i, \dots, s_n such that φ holds on s_0, \dots, s_i and ψ holds on s_i, \dots, s_n . Using this chop operator, many other interval based temporal operators can be described. However, the major drawback of ITL is that the satisfiability is no longer decidable [232]. Moszkowski [378, 379] used ITL to establish executable specifications and in this way verified entire arithmetic-logic units of microprocessors at the transistor level. Leiser [319] extended Moszkowski’s model by setup and hold times and used temporal PROLOG to also verify an arithmetic-logic unit at transistor level.

1.2.6 Decidable Theories and ω -Automata

A lot of decision procedures for special logics have been developed independently of the research in the verification of computer systems and the development of automated theorem proving procedures for first and higher order predicate logics. Based on the disappointing result of Gödel’s incompleteness theorem in 1931 and the undecidability theorem of first order logic of Church and Turing in 1936/1937 [99, 492], a considerable amount of work has been put on the investigation of decidable mathematical theories.

The first investigations to find decidable fragments were to study quantifier prefix classes (see Section 6.2.3). This means that first order formulas of the form $\Theta_1 x_1 \dots \Theta_n x_n. \varphi$ with $\Theta_i \in \{\forall, \exists\}$ and quantifier-free φ are considered with restricted patterns of quantifiers Θ_i . For example, Bernays and Schönfinkel [39], and Ramsey [422] showed that the sat-problem of the formulas where only one change from \exists to \forall quantifiers is allowed, i.e., the frag-

ment $\text{FO}(\exists^*\forall^*)$, is decidable. Other fragments with decidable sat-problem have been found by Ackermann [3] ($\text{FO}(\exists^*\forall\exists^*)$) and Gödel, Kalmár, and Schütte [213, 273, 448] ($\text{FO}(\exists^*\forall^2\exists^*)$). These are the maximal decidable classes, since there are formulas with quantifier prefixes $\forall^3\exists$ (Surányi [479], 1959) and $\forall\exists\forall$ (Kahr, Wang, and Moore [272], 1962) that can describe undecidable problems. Hence, the prefix classes $\text{FO}(\alpha)$ were completely classified in 1962 for first order logic without equality. Only the result for $\text{FO}(\exists^*\forall^2\exists^*)$ with equality was missing, that has been answered by Goldfarb [214] in 1984 to the negative. Besides the quantifier prefix classes, also restrictions to a finite number of variables have been considered. It is remarkable that first order logic with only two variables, but arbitrary quantification, is decidable [218, 219, 375], which has been essentially shown by Mortimer in 1975. As modal logic can be embedded in predicate logic with only two variables, Vardi posed the question [495] whether this could be a reason for the robust decidability of modal logic. Section 6.2.3 gives more details on these classes, and for further reading, we refer to the original literature, and excellent books on this topic like [56, 146, 323].

Many other decidable theories have been considered, in particular fragments of arithmetic such as Presburger arithmetic [415] or Skolem arithmetic, and interesting decision procedures have been found for them [83, 84, 336, 446, 447, 521, 522]. Furthermore, the monadic second order theory of one successor S1S [80, 455, 486], which is equivalent to the monadic second order theory of linear orders $\text{MSO}_<$ is another decidable theory that subsumes Presburger arithmetic (see page 433).

We have to explain, what a ‘theory’ in this sense is. We have already explained that the domain \mathcal{D} that is used for interpreting the variables of predicate logic is chosen arbitrarily for the interpretation. For special theories, one accepts certain axioms which are formulas that are assumed to be valid. Hence, the set of domains is thereby restricted to those domains that satisfy these axioms. The theory that is induced by these axioms is then the set of formulas that can be derived from this set³.

For example, to only consider domains with more than n elements, we could establish the axiom $\exists x_1 \dots \exists x_n \cdot \bigwedge_{i=1}^n \bigwedge_{j=1, j \neq i}^n .x_i \neq x_j$. As another example, one could set up the following axioms to specify that \leq is a total order:

$$\begin{aligned} \text{Reflexivity: } & \forall x. x \leq x \\ \text{Antisymmetry: } & \forall x. \forall y. x \leq y \wedge y \leq x \rightarrow x \doteq y \\ \text{Transitivity: } & \forall x. \forall y. \forall z. x \leq y \wedge y \leq z \rightarrow x \leq z \\ \text{Totality: } & \forall x. \forall y. x \leq y \vee y \leq x \end{aligned}$$

³ As first order logic has complete proof procedures, this is the same as the set of formulas that are logical consequences of the axioms. For higher order logic, which is not complete, we must distinguish between theories that are *derived* by a special calculus and others that are obtained by *logical consequence*.

All first order formulas that can be derived from the above set of axioms form the first order theory of total orders. However, for the monadic first and second order theories of *linear order* $\text{MFO}_{<}$ and $\text{MSO}_{<}$, the domain is further restricted such that the elements are lined up in a chain. Hence, the domain is either a finite string, or isomorphic to the natural or the integer numbers. To axiomatize this, one has to add the Peano axioms (see page 405), which requires second order logic.

These theories are important for the formalisms we will consider in this book. However, at the beginning of the research, $\text{MSO}_{<}$ has not been discussed. Instead, an equivalent variant of the logic has been considered, which is called the ‘monadic second order logic of one successor’ S1S. Note, however, that although the second order logics $\text{MSO}_{<}$ and S1S are equally expressive, this is not the case for their first order fragments. Indeed, $\text{MFO}_{<}$ is strictly more expressive than the first order fragment of S1S.

In 1960, J.R. Büchi succeeded in proving the decidability of S1S by reducing S1S to ω -automata. These ω -automata differ from classical finite state automata as introduced by Kleene⁴ in 1956 [285] as follows: Kleene’s finite state automata are used to decide whether a finite word belongs to a regular language or not. This is done by an automaton by reading the word letter by letter, and possibly changing the internal state each time a letter of the word is read. A word is accepted, iff the automaton is in one of its designated final states after reading the entire word.

In contrast to that, ω -automata accept or reject *infinite* words, and therefore their acceptance condition must be defined differently. A natural condition imposed by Büchi was that a word should be accepted if there is a run over this word that reaches at least one of the designated states infinitely often. As this acceptance condition has been introduced by Büchi, these automata are nowadays called Büchi automata. A lot of other variants of ω -automata were defined after Büchi’s pioneering work including Muller automata [363, 380] (1963), Rabin automata [363, 418] (1969), Streett automata [476] (1982), and Parity automata [376] (1984). The other kinds of ω -automata have been introduced mainly for one reason: while for finite automata on finite words, it is possible to construct for any nondeterministic automaton an equivalent deterministic one, this does not hold for Büchi automata. However, for Muller, Rabin, Streett, and Parity automata this can be achieved. The complexity for determinizing these ω -automata is higher than for automata on finite words, although still exponential: While for the latter 2^n states are sufficient, the optimal construction of a deterministic Rabin automaton from a Büchi automaton requires $2^{O(n \log(n))}$ states (where n is the number of states of the given automaton). We will see that the automaton classes form a strict hierarchy that has moreover a strong relationship to

⁴ Kleene’s paper was actually a mathematical reworking of the ideas of W. McCulloch and W. Pitts [359], who, in 1943, had presented a logical model for the behavior of nerve systems that is in principle a finite automaton.

the Borel hierarchy known in topology. Details on the different kinds of ω -automata, the automaton hierarchy, and various translation procedures between ω -automata are given in detail in Chapter 4. We will also outline the relationship between algebraic structures and automata, and will characterize the important class of noncounting automata by algebraic properties.

The translation of S1S or $\text{MSO}_{<}$ formulas of length n to equivalent Büchi automata is nonelementary. This means that the procedure runs in a time that can not be bound by a finite nesting of exponentials and has therefore been considered to be useless for a long time. However, recently the procedure has been implemented and been successfully applied to the verification of generic hardware circuits [31, 240, 265, 278, 442]. A possible future direction of research is therefore to develop decision procedures for timing diagrams which can be understood as graphical representations of S1S or $\text{MSO}_{<}$ formulas. We consider translations between S1S and $\text{MSO}_{<}$, and also to equivalent ω -automata in Chapter 6.

ω -automata are strongly related to temporal logic, in that they also establish a temporal relationship on the input sequences they accept. Therefore, translations from temporal logics to ω -automata have been considered, mainly for the construction of efficient decision procedures for temporal logics. We have already mentioned that Wolper [523] suggested the use of ω -automata as special temporal operators to extend the expressiveness of temporal logics. We will consider such a branching time temporal logic \mathcal{L}_ω in detail in Chapter 4. A basic translation from linear time temporal logic to \mathcal{L}_ω is then considered in Section 5.4. It is remarkable that the translation can be performed in linear time w.r.t. to the length of the given temporal logic formula, when a symbolic description of the automaton is derived (that can be directly used for symbolic model checking). We will study this translation procedure in detail, and will show some novel improvements. These improvements will then lead to the definition of temporal logic classes TL_κ in correspondence with the automaton hierarchy: We define for any automaton class κ , a logic TL_κ that can be translated to the automata in the class κ . We will also prove the completeness of the logics TL_κ , which follows from related results of Manna and Pnueli [350–352].

Moreover, we will see in Chapter 6 that both S1S and $\text{MSO}_{<}$ can be easily translated to quantified temporal logic which can in turn be converted to ω -automata with a nonelementary translation procedure. Finally, we will prove in Chapter 6 Kamp's result that temporal logic is equal expressive as $\text{MFO}_{<}$, the first order fragment of $\text{MSO}_{<}$. In Section 5.5.1, we will show how noncounting automata can be translated to temporal logic, the converse is simple. Hence, robust characterizations of all of these logics exists.

Therefore, ω -automata have served as a basic formalism whose decidability and the corresponding complexity is well-known. In order to prove the decidability of a theory such as $\text{MSO}_{<}$, one simply has to give a translation to equivalent ω -automata. To determine the complexity, one simply has to add

the complexity of the translation procedure to the complexity of the remaining decision procedure for the obtained ω -automaton.

We have also seen that the μ -calculus is a ‘basic machinery’ that can be used as a foundation for other formalisms. It is therefore natural to ask how ω -automata and μ -calculus are related to each other. However, this relationship has been considered relatively late. Thomas showed in 1988 [485] that the temporal logic that is obtained by using ω -automata as temporal operators is strictly less expressive than the second order theory of two successors (S2S). In contrast to S1S, where only one successor is available, the monadic second order logic of two successors, S2S has two successors. For this reason, its domains are not lined up, but form trees that may be finite or infinite. In particular, domains for S2S are binary trees, and for S_nS trees of branching degree n . As Niwiński [389] has shown that S2S has the same expressiveness as the propositional μ -calculus, it follows that the μ -calculus is strictly more expressive than this temporal logic.

The same result follows by translating \mathcal{L}_ω to the μ -calculus (see Section 4.8). Such a translation procedure has been given by Dam in 1994 [135] where a given Büchi automaton with n states is translated to an equivalent μ -calculus formula of size $2^{O(n)}$ and an alternation depth ≤ 2 . By Bradfield’s and Lenzi’s recent result (that the alternation hierarchy is strict), it thus follows that the μ -calculus is strictly more expressive than \mathcal{L}_ω . We will present in Section 4.8 a translation procedure that constructs for given ω -automata equivalent vectorized μ -calculus formulas of size $O(n)$ with an alternation depth ≤ 2 . Using this translation, the decision procedures obtained from the μ -calculus match the lower bounds of other decision procedures that directly work on ω -automata. Hence, the μ -calculus is indeed a very expressive language that can serve as a basic machinery for various problems.

If one restricts the consideration to single paths or linear time structures, where each state has a unique successor state, it is interesting to note that \mathcal{L}_ω and the μ -calculus become equal expressive. This shows that the additional expressiveness of the μ -calculus is due to statements on the branching behavior of the structures: while any formula of \mathcal{L}_ω can only have a finite number of nested E and A quantifiers, there are only finitely many choices of taking a new path along a considered path. In contrast to that, the fixpoint iterations for μ -calculus formulas can look arbitrarily deep in the branching of the structure. We will explain this in more detail in Section 4.8.

The situation changes when ω -automata on infinite trees are considered. Pioneering work of Rabin in 1969 [418] proved that the second order theory of two successors (S2S) is decidable by reducing it to equivalent ω -automata on infinite trees. Hence, this is a branching time analogon to Büchi’s result of the equivalence of S1S and ω -automata on words. However, not all results that hold for ω -automata on infinite words have corresponding results for the tree automata. In particular, it can be shown that Rabin tree automata are equal expressive as the μ -calculus (see Appendix B.3), but strictly more expressive than Büchi tree automata.

Due to lack of space, we will not consider ω -automata on infinite trees in this book (apart from Appendix B.3), although their theory gives further insights into the formalisms considered here. The interested reader is referred to Thomas' survey [486, 489] that contains many more references and the new book of Grädel, Thomas, and Wilke [221]. Also, we will not consider *alternating* ω -automata [82, 96, 374] which are a generalization of nondeterministic automata. However, the symbolic descriptions that we use to embed automata in temporal logic are somehow related to alternating automata.

1.2.7 Summary

As can be seen, a plethora of formalisms for the verification of programs, and, in particular, for the verification of concurrent programs has been proposed. Up to now, their relationship is almost clear and for many different formalisms we already know if translations between them exist and how to translate them efficiently. In this book, the most important classical formalisms, namely μ -calculus, ω -automata, temporal logics, and predicate logic are considered and their relationship is outlined in detail. A special emphasis is thereby the existence of algorithms either for translation between these formalisms or for the implementation of decision procedures for the formalisms.

In particular, two basic machineries can be selected for the verification of most properties: we can reduce the property to an ω -automaton problem, or we could use the μ -calculus as a basic formalism. We have already discussed that \mathcal{L}_ω can be translated to the μ -calculus, so the decision procedure for the μ -calculus can be used to solve these problems with essentially the same complexity.

However, this complexity is determined in a theoretical manner. If the procedures are implemented, they behave rather differently. Therefore, we consider both solutions in the following: Given the ω -automaton \mathfrak{A} , a Kripke structure \mathcal{K} and a state s of the Kripke structure, suppose we have to decide whether $E\mathfrak{A}$ holds in s or not, i.e., we have to solve the model checking problem $(\mathcal{K}, s) \models E\mathfrak{A}$. Then, as a first solution, we translate $E\mathfrak{A}$ to an equivalent μ -calculus formula $\Phi_{E\mathfrak{A}}$, and check the equivalent problem $(\mathcal{K}, s) \models \Phi_{E\mathfrak{A}}$ by means of the model checking procedures for the μ -calculus. The second solution is to interpret the transition system of \mathfrak{A} as a Kripke structure $\text{Struct}(\mathfrak{A})$. Hence, the problem can be reduced to check whether the product structure $\mathcal{K} \times \text{Struct}(\mathfrak{A})$ contains a path that satisfies the acceptance condition of \mathfrak{A} ⁵.

Both solutions have advantages and disadvantages: while the structure $\text{Struct}(\mathfrak{A})$ can be symbolically represented, this is not the case for the formula $\Phi_{E\mathfrak{A}}$, whose representation may therefore be exponentially larger than that of $\text{Struct}(\mathfrak{A})$. On the other hand, the product structure $\mathcal{K} \times \text{Struct}(\mathfrak{A})$

⁵ This is not quite correct, but we omit details here. Consider the discussion in Section 4.8

that must be considered in the ω -automaton based approach may suffer from a state explosion, so that it may not be representable. However, this need not necessarily be the case. In fact, many combinations of states of \mathcal{K} and $\text{Struct}(\mathfrak{A})$ are inconsistent and therefore do not occur in the product structure $\mathcal{K} \times \text{Struct}(\mathfrak{A})$. Hence, this structure may even be smaller than \mathcal{K} . This effect is best seen, when the automaton \mathfrak{A} is deterministic and does only accept a single input sequence. In this case, the structure $\mathcal{K} \times \text{Struct}(\mathfrak{A})$ will also consist of only one path.

This effect is exploited in on-the-fly approaches that aim at constructing the product structure $\mathcal{K} \times \text{Struct}(\mathfrak{A})$ in such a way that inconsistencies between \mathcal{K} and $\text{Struct}(\mathfrak{A})$ are detected as soon as possible. Hence, the ω -automaton based approach may be much more efficient than the one obtained by a translation to the μ -calculus. However, in some cases, there may be not many inconsistencies, and therefore the structure $\mathcal{K} \times \text{Struct}(\mathfrak{A})$ may really suffer from the state explosion. In these cases, the solution via the translation to the μ -calculus will probably be more efficient. However, this crucially depends on \mathfrak{A} and \mathcal{K} , and can hardly be estimated before the computations are actually performed.

Apart from the choice of a basic machinery, it is important to have different ways to specify a particular property. Some properties can be better expressed by ω -automata, others better by temporal logics, and others even directly in $\text{MSO}_{<}$. For reasons of redundancy, it is also desirable that a property is given in different formalisms, and additionally in natural language to explain the essential idea of the specification (preciseness is not necessary, and not even desired for the natural language description for that purpose).

The *readability of specifications* is a major issue. Considering the formalisms in this light, the μ -calculus may only be seen as a basic machinery since its formulas really tend to be unreadable. Therefore, specifications are rarely given directly in the μ -calculus. Another issue that is related with the complexity of the decision procedures and the expressiveness of a logic is how *succinct* the formalisms are. For example, it has already been noted that the translation from $\text{MSO}_{<}$ to \mathcal{L}_ω suffers from a nonelementary blow-up. This means that there are formulas in $\text{MSO}_{<}$ that can express something that can also be expressed in \mathcal{L}_ω , but with an enormous blow-up of the formula's size.

This observation holds for many formalisms. For example, the temporal logic LeftCTL^* is equally expressive as the temporal logic CTL. However, the translation from LeftCTL^* to CTL involves an exponential blow-up, so that the model checking procedure for LeftCTL^* runs in time $O(|\mathcal{K}| |\Phi| 2^{2|\Phi|})$, while the one for CTL runs in time $O(|\mathcal{K}| |\Phi|)$. But is this an advantage or a disadvantage of CTL? It can be shown that there are formulas in LeftCTL^* that can only be expressed by CTL formulas whose size can not be bound by a polynomial [4, 266, 513]. To illustrate this, simply take a NP-complete problem and describe it with a polynomially sized LeftCTL^* formula (see Section 5.2.2). As the model checking procedure for CTL runs in polynomial time, the existence of polynomially sized CTL formulas for that LeftCTL^* for-

mula would imply $P = NP$, which probably does not hold (this is to date an unsolved problem). The necessity of the exponential blow-up for the translation from LeftCTL^* to CTL follows from results given in [4, 266, 513].

Hence, the complexity of the decision procedures for the formalisms should be taken with some care as an argument for or against a formalism. In fact, the complexity of a given verification problem is more inherent to the problem itself than to the formalism in which we describe it. Therefore, the readability should be of more interest. However, readability is a subjective issue and can not be quantified in a reasonable manner, unless by the length of the formulas. *Therefore, there are good reasons to consider all the mentioned formalisms, and to use whichever one best suits the problem.* Succinctness can also be viewed as the reciprocal of the complexity of the decision procedure. Hence, we may say that LeftCTL^* is exponentially more succinct than CTL , or that $\text{MSO}_{<}$ is nonelementarily more succinct than ω -automata or temporal logic.

1.3 Outline of the Book

There is still no complete survey of all formal methods and their relationships, so we have to refer instead to a couple of surveys and books such as [158, 223, 486]. Clarke and Wing give an interesting survey on some formal methods and examples in [114]. Verification tools that are particularly well-suited for hardware verification are presented Kropf in [295] and also in the new book [38]. Furthermore, a good introductory textbook to model checking is given by Clarke, Grumberg, and Peled in [112].

The contents of this book is a self-contained and detailed treatment of the basic ‘finite-state’ formalisms, namely the μ -calculus, ω -automata, temporal logics, and (monadic) predicate logics. We describe translation procedures for any of these formalisms whenever they exist, and compare the expressiveness and complexities of these logics. We will see that the μ -calculus is the most expressive logic of the ones mentioned, and that ω -automata and the second order theory of linear order $\text{MSO}_{<}$ are equally expressive. Moreover, temporal logics are strictly less expressive than ω -automata, and equally expressive than the monadic first order theory of linear order $\text{MFO}_{<}$. It is also well-known which subset of the ω -automata corresponds with the temporal logic formulas, namely the noncounting ω -automata, which furthermore correspond with star-free ω -regular expressions (cf. page 382). We will refine these relationships by considering many sublogics, such as μ -calculus formulas of a certain alternation depth, ω -automata with special acceptance conditions, and many temporal logics. We will consider in detail how these fragments are related to each other. In particular, we will consider the already mentioned temporal logics LeftCTL^* and AFCTL^* that can be translated to the alternation-free μ -calculus. Moreover, we consider a hierarchy of temporal logics TL_κ that corresponds with the automaton hierarchy (i.e., the Borel hierarchy).

For a comparison of these formalisms, we will combine them to obtain a unified specification language $\mathcal{L}_{\text{spec}}$ that is essentially the union of μ -calculus, ω -automata, and temporal logics (formulas of different formalisms may however be nested). The semantics of $\mathcal{L}_{\text{spec}}$ is defined over finite Kripke structures that are general models for reactive finite state systems.

Beyond the translation of the various formalisms into each other, and proving the expressiveness results of these formalisms, we will also consider verification procedures for these logics. For this reason, we will use both the μ -calculus and ω -automata as basic mechanisms. In particular, we also show how to translate ω -automata to μ -calculus so that μ -calculus can be used as our single basic decision procedure. However, as we have already discussed, the decision procedures for ω -automata have some advantages for practical usage. Therefore, both decision procedures, i.e., those for μ -calculus and those for ω -automata, have their own merits.

It has to be emphasized that the book is almost self-contained. Most of the theorems are given with detailed proofs so that the reader will find explanations in any detail that he or she wants. However, those who are only interested in the facts and do not want to find out (at least in a first reading) *why* these facts hold, may skip the proofs.

The outline of the book is as follows: in the next chapter, we define the unified specification logic $\mathcal{L}_{\text{spec}}$, its models, and its semantics. We will also discuss some normal forms that will be used throughout the book. We will also present the syntax and semantics of the vectorized μ -calculus. This chapter is not meant for understanding the formalisms that are presented there in terms of their syntax and semantics. It simply lists the basic definitions of syntax and semantics of the different formalisms that are considered in more detail in the following chapters of the book. However, it presents the theory of Kripke structures, simulation and bisimulation relations, as well as products and quotients of Kripke structures that are fundamental constructions used in verification.

Chapter 3 will then define the μ -calculus both in its basic form as well as in its vectorized form. The vectorized form can be exponentially more succinct since it allows us to share common subformulas in equation systems without representing them several times. Starting from the theory of lattices and the fundamental theorem of Tarski and Knaster, we will develop efficient model checking procedures for the μ -calculus that are to date the best known algorithms to solve this problem. We will give detailed proofs and will determine the runtime complexity of the discussed model checking procedures. Appendix B discusses moreover local model checking procedures and decision procedures for the satisfiability of μ -calculus formulas. To this end, we need ω -tree automata, which is a formalism that is not used elsewhere in the book. At the end of Chapter 3, we consider reductions by (bi)simulation relations and the relationship to dynamic logic and infinite games.

Chapter 4 considers ω -automata on words. In order to incorporate ω -automata in our logic $\mathcal{L}_{\text{spec}}$, we use symbolic descriptions of ω -automata.

In these descriptions, the states are encoded by a finite number of Boolean state variables. The initial states and the transition relation can then be given by an almost propositional formula. In particular, we will define different kinds of ω -automata, compute Boolean combinations of them and convert different acceptance conditions into each other. Moreover, we consider their expressiveness and therefore reveal the Borel hierarchy. It is straightforward to define, in Section 4.8, a branching time temporal logic \mathcal{L}_ω whose temporal operators are ω -automata. We will present transformations between different kinds of ω -automata and also procedures to check if such a reduction is possible. In Section 4.7, we explain the relationship between finite state automata and monoids. In particular, we consider aperiodic monoids and noncounting automata. Furthermore, we consider the determinization of ω -automata which will allow us to flatten nested expressions in \mathcal{L}_ω (however with a nonelementary blow-up). Finally, we will discuss model checking procedures for our automaton based logic \mathcal{L}_ω in Section 4.8, and translations to the μ -calculus.

Temporal logics are considered in Chapter 5. We will first consider well-known temporal logics, including CTL, LTL, CTL*, and some other variants like CTL² and the logic LeftCTL*. We will show basic translation principles in Section 5.3.3 and Section 5.3.4 that allow us to translate LeftCTL* to CTL. We note that these principles are not restricted to LeftCTL*, and can be applied to general CTL* formulas to reason about their equivalence, or to translate temporal logic to the μ -calculus. Translations of CTL and LeftCTL* to the μ -calculus are given in Section 5.3, and an optimized version is given in Section 5.3.5 by directly translating to the vectorized μ -calculus. In Section 5.4, we will consider the translation of temporal logics to ω -automata. We start with a simple translation procedure and consider improvements of it, which then lead to the definition of the temporal logic hierarchy (Section 5.4.3). This hierarchy was investigated by Manna and Pnueli in [350–352]. However, they only considered a restricted normal form of the logics that we will discuss in Section 5.4.3. A novel contribution is the definition of the logics TL_κ [437] and the observation that the future time fragments of these logics are expressively complete w.r.t. to the corresponding full logic TL_κ . This enables us to define, in Section 5.4.5, on page 373, the branching time logic AFCTL* that corresponds with the set of CTL* formulas that can be translated to the alternation-free μ -calculus. We will moreover present a translation from AFCTL* to \mathcal{L}_ω that runs in linear time (note however the symbolic representation of \mathcal{L}_ω). Section 5.5.1 considers then the relationship between noncounting automata and temporal logic, and Section 5.5.2 shows the completeness of the TL_κ logics, because they have the separation property (Section 5.5.3). In Section 5.6, we briefly consider the complexity of some temporal logics.

Chapter 6 presents the relationship between ω -automata, temporal logics, and (monadic) predicate logics. After listing basic results on first and second order predicate logic, we list known results about decidable fragments of

first order logic. In Section 6.3, we prove Büchi's result, i.e., the equal expressiveness of $S1S$, $MSO_{<}$ and ω -automata, and in Section 6.4, we prove Kamp's result, i.e., the equal expressiveness of $MFO_{<}$ and temporal logic. The equal expressiveness between noncounting ω -automata and temporal logic is presented in Section 5.5.1. Hence, the book covers all relevant translation procedures between different formalism used for the specification and verification of reactive systems. Finally, we will give some conclusions in Chapter 7.



<http://www.springer.com/978-3-540-00296-3>

Verification of Reactive Systems

Formal Methods and Algorithms

Schneider, K.

2004, XIV, 602 p., Hardcover

ISBN: 978-3-540-00296-3