CHAPTER 1
Introduction

1.1 Motivation

> Anyone who has never made a mistake has never tried anything new.
> Albert Einstein

We are living in a world where software and hardware systems control the temperature in our rooms, our railroad crossings and our nuclear plants, where computer systems are responsible for transferring our salaries, for monitoring kidney dialysis and for supplying our supermarkets with fresh products. The correct functioning of those systems is clearly of vital importance. However, the construction of software and hardware products carrying out these jobs is often an extremely complex task and various examples evidence the presence of serious errors in computer systems [Pet96]. The consequences of a malfunctioning system may vary from user frustration, for instance in the case of an operating system that crashes, to economic damage, as is illustrated by the hardware bug in the Intel Pentium 5 microprocessor, and life threatening situations, as is demonstrated by the software bug in the Therac–25 machine. The latter caused several mortal victims due to overdoses of X–radiation.

Formal Methods

> But there is another reason for the high repute of mathematics: it is mathematics that offers the exact natural sciences a certain measure of security which, without mathematics, they could not attain.
> Albert Einstein

The field of *formal methods* proposes a development methodology that should help to improve the quality of software and hardware and to prevent errors as above. In this field, one argues that the reliability of computer systems improves significantly when formal methods are given a more prominent place in the design and implementation of these systems. Formal methods refer to a wide variety of techniques, tools and languages for system analysis that have rigorous mathematical foundations. These methods can be utilized throughout the entire software life cycle, that is, from the initial investigation of the customer's wishes, the design, implementation, to the testing and maintenance. This thesis pays attention to the **formal specification** and **formal verification** as means to prevent system malfunctioning.

Good software engineering practice requires that, when building a computer system, one starts from a specification, which expresses the desired system behavior. Since misinterpretation is a common source of errors, it is important that the specification is unambiguous and precise. Many people working in formal methods believe that *formal specifications* are often suitable here, because these fulfill the requirement for precision to its highest form.

Formal specifications are specifications written in formal languages, that is, languages whose meaning has been pinned down mathematically. In industry, however, specifications are often written in natural language (usually English) or semi–formal languages (such as UML [BRJ99]).

Apart from avoiding misinterpretations, the use of formal specification languages has another advantage. When the system has been implemented, formal specifications – unlike specifications written in natural language – allow one to prove mathematically that the implementation indeed meets the properties expressed by the specification. This process is called *formal verification* and if a system has been verified formally, one knows that the system behaves correctly under all circumstances – provided, of course, that the specification accurately expresses the intended system behavior.

Since formal verification is rather time consuming, the current industrial practice is to investigate the system correctness via *testing*. As has already been recognized in the 1960s by E. Dijkstra, testing can only show the presence of errors, not their absence. This is because exhaustive testing is usually not feasible, as it requires a very large number of tests, often infinitely many, to be performed. Thus, even if testing did not reveal any errors, this does not mean that the system is correct, because there remain many cases that have not been tested. Therefore, formal verification is favorable when system correctness is crucial.

However, it is important to realize that verification can never give an absolute guarantee for correctness. In fact, verification is always relative to the correctness of the tools applied in the verification, the compilers used to compile the final program, the hardware the implementation runs on and – very importantly – the extent to which the specification really expresses what one has in mind. Therefore, other techniques to evaluate correctness remain necessary.

Reactive Systems

> You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat.
>
> Albert Einstein

A lot of effort has been spent and is being spent on the verification of *transformational programs*, [Dij69, Apt81, Hui01]. These are computer programs, such as programs to compute the sinus function or to bookkeep one's finances, that are given some input, then compute for some while and finally output their results. This thesis, in contrast, is concerned with verification techniques for **reactive systems**. Reactive systems are systems that continuously interact with the environment in which they are operating and in principle run perpetually. These systems react to stimuli from their environment and generate stimuli which, in return, influence the environment.

A simple example of a reactive system is an automatically controlled railroad crossing. The system controlling the gates has to check continuously whether a train is arriving and, if so, to close the bars. When the train has left the crossing, the bars have to be reopened (unless another train is approaching). The environment of the controller is formed by the train and the bars. Stimuli here are the signals telling that the train is arriving or leaving and telling that the bars should be raised or lowered. Other examples of reactive systems are telephone switching systems, radar systems, chemical batch plants, steam boilers, etc.

Many aspects play a role in the operation of reactive systems. Depending on the type of application, one or more of the following issues can be relevant. One can study the system's *functionality*, which concerns the qualitative behavior of the system; one can consider the system's *performance characteristics*, which concern its quantitative behavior and efficiency; one can focus on *security*, which is the extent to which the system is resistant to undesirable access by intruders; one may take into account the *availability*, which is concerned with the extent to which a system can fulfill the service requests it gets; one can treat the *fault tolerance* of the system, which is the way the system deals with failing components; one can study the *mobility* of the system, which is the possibility of a system to exchange communication links and to run remote applications, etc.

This thesis focuses in particular on **functional behavior** of reactive systems, although several of the techniques described are also relevant in performance analysis and fault tolerant systems.

Secondly, a system can be described at several levels of detail. Depending on the kind of application, one or more of the following features can be incorporated in the system model. *Data* that is manipulated, stored and communicated in the system, *probabilities* which describe the random choices and stochastic behavior in the system, *continuous* behavior that is exhibited by physical components in the system, *parameters* to represent a whole class of systems with the same structure, etc.

This thesis is about **probabilistic**, **timing** and **parametric** aspects in systems modeling and verification. More specifically, as we will explain in the next section, we focus on **discrete** rather than on continuous probabilistic choice, we use **continuous time** rather than discrete time and the only system parameters we consider are **timing parameters**.

Formal Verification Techniques

> We can't solve problems by using the same kind of thinking we used when we created them.
>
> Albert Einstein

There are many verification techniques available to establish the correctness of a system. *Algebraic methods* show that a system meets its specification by expressing them both as terms in process algebra. Then both terms are equated via algebraic laws, very much like the calculations one makes in algebra or analysis. In a proof via *behavioral relations*, one compares the external behavior of two state transition systems, one representing implementation and the other the specification of a system. *Temporal logics* express the specification by a logical formula with temporal operators, which can be used to verify a state transition system. In *Hoare–style proofs* one reasons about the program text by using axioms and logical inference rules to derive statements about the program.

Since verification tends to be a rather complex task, computer support is indispensable for the verification of most real–life applications. *Manual verification* is only feasible for small systems and, moreover, verification carried out by hand is very error prone; in the words of Wolper [Wol98], "Manual verification is at least as likely to be wrong as the program itself." Tool support exists for all of the techniques mentioned above, by providing either fully automated or semi–automatic techniques for program analysis. *Fully automatic* verification methods are preferable, because these are fast and easy, but are only available for a limited class of systems. Especially *model checking*, which is based on temporal logic, has become popular. Also, several variants of (bi–)simulation relations can be checked automatically.

Semi–automatic verification requires user interaction with the computer tool, where the user provides his or her knowledge of the system and the tool checks whether or not the information provided indeed establishes the desired result. Theorem provers such as PVS and Isabelle allow the user to formalize and prove a mathematical theorem and have turned out to be very successful in proving a programs' correctness.

This dissertation considers **manual** and **fully automatic** techniques. More specifically, we use **model checking** and **proofs via (bi–)simulation**. Moreover, we use a combination of model checking and bisimulation: we use model checkers to construct a bisimulation proof.

This Thesis

Summarizing, this thesis presents several techniques based on model checking and (bi–)simulations to reason about the functional correctness of systems with discrete probabilistic choice, real–time and/or timing parameters. More specifically, we develop several theoretical results for probabilistic simulation and bisimulations, we present a model checking technique for parametric systems and we combine both techniques to analyze the IEEE 1394 Root Contention Protocol. This is an industrial leader election protocol in which timing, probabilistic and parametric aspects play a crucial role and which is therefore a suitable case study to investigate the feasibility of the verification techniques presented in this thesis.

The rest of this introduction is organized as follows. Section 1.2 introduces the system models used in this thesis to analyze reactive systems and explains how timing, parametric and probabilistic aspects can be incorporated in these models. In Section 1.3, we explain the concepts of model checking and (bi–)simulation proofs. Finally, Section 1.4 presents an overview of the results in this thesis and the contents of subsequent chapters.

1.2 Models for Timed, Parametric and Probabilistic Systems

> Everything should be made as simple as possible, but not simpler.
>
> The human mind has first to construct forms, independently, before we can find them in things.
>
> Albert Einstein

Verification is an activity to be carried out at the level of language, not on physical objects in reality. In this respect, verification differs from testing, because testing can be performed at physical systems directly. Since reactive systems intrinsically contain physical components, their verification has to be based on system models. A system model describes the system at an appropriate level of detail. As it is the case with a model of any object, one omits the details which are irrelevant for the aspects of interest, thus keeping the verification feasible. Note that a system model can be an intermediate product created during the design phase of the system.

The field of formal methods has put forward an abundance of formalisms to describe reactive systems. We mention just a few of them. *Process algebras* [Hoa85, Mil80] are an algebraic framework in which process equality is specified by algebraic laws. *Petri nets* [Rei85] are based on state transition systems whose dynamics are described by the passage of tokens. *State charts* [Har87], *Message sequence charts* and *UML* [BRJ99] are graphical languages with a rich syntax. UML subsumes the former two and its precise semantics is under construction. Almost all these formalisms have been extended with probabilistic, timing and/or parametric aspects.

The models used in this thesis are all based on **labeled transition systems**, also called **automata**. These have turned out to be an intuitive, yet powerful framework to describe and analyze reactive systems and they have been extended with probabilities, timing and parameters.

This section introduces the automaton model and its extensions with probability, time and parameters in an informal way. A formal treatment will appear in subsequent chapters. As a running example in this section, we treat a railroad crossing. This example is a benchmark problem in verification and has been defined in [HJL93]. Clearly, the models presented here are simplified in many ways, but are still rich enough to illustrate various aspects in modeling and analysis of reactive systems.

Labeled Transition Systems

> Most of the fundamental ideas of science are essentially simple, and may, as a rule, be expressed in a language comprehensible to everyone.
>
> Albert Einstein

An *automaton*, or *labeled transition system*, describes an application as a set of *states* and *transitions*. A state represents a snapshot of the system and the transitions represent state changes in the application.

Example 1.2.1 Figure 1.1 shows a simple automaton model of a controller in a railroad system. The task of the controller is to open and close the gates. The controller starts in the state $Idle$, which is denoted by the double circle in the picture. When it receives an $approaches?$ signal, indicating that the train is arriving, the controller moves to the state $Lower\_bars$. We also say that the controller *takes a transition* from $Idle$ to $Lower\_bars$ labeled by $approaches?$. In the state $Lower\_bars$, it sends out a $lower!$ signal to tell the gates to close and moves back to the state $Idle$. Similarly, when the controller receives a $leaves?$ signal, it takes the corresponding transition. This leads to the state $Raise\_bars$, in which the controller sends a $raise!$ to the gates and returns to the state $Idle$. Note that the receipt of a signal (usually called an *action*) is denoted by $a?$ and the sending of it by $a!$.

Furthermore, consider the (very much simplified) models of the train and the gate in Figure 1.2, which do not do much more than sending $approaches!$ and $leaves!$ signals and receiving $raise?$ and $lower?$ messages respectively. We only mention these to illustrate the following. When, for instance, the train sends an $approaches!$ signal, then it is received by the controller immediately. This means that the train and the controller take their $approaches-$transitions simultaneously, i.e. both together and at exactly the same time. In technical terms, we say that the train and the controller *synchronize* on the action $approaches$. They also synchronize on the action $leaves$. Similarly, the controller and the gate synchronize on the actions $lower$ and $raise$.

Timed Systems

> The only reason for time is so that everything doesn't happen at once.

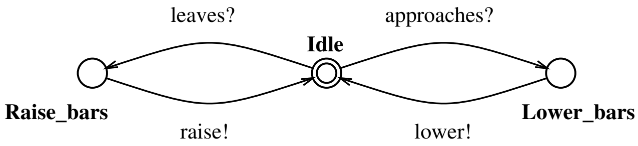

Figure 1.1: Simple automaton model of a railroad controller

Figure 1.2: Train and gate models

Albert Einstein

Timing aspects are crucial in many computer applications. In the railroad example for instance, assume that a train approaches the railroad crossing. Then the bars have to be closed when the train is at the crossing. This means that the closing signal has to be passed to the gate fast enough so that the time needed for the bar to close is smaller than the time needed for the train to reach the crossing. Obviously, small delays can have catastrophic effects.

The necessity of timing aspects has long been recognized and many formal methods have been extended with time. In this dissertation, we use **timed automata** [AD94] and **timed I/O automata** [MMT91], which are both timed versions of labeled transition systems.

With the use of time, the railroad crossing can be modeled more accurately. The following example presents a timed automaton description of the railroad system.

Example 1.2.2 Figure 1.3 depicts a timed automaton model of railroad controller. The states and actions are the same as before, but now *clocks* come into play. Initially, all clocks are zero. Via *state invariants*, clocks can limit the amount of time that a component can stay in a state and via *guards* on transitions, clocks can restrict the times at which a transition can be taken. Finally, clocks can be reset to 0.

Let us explain this by means of the controller in Figure 1.3. When the controller receives a $approaches?$ signal and moves to the state $Lower\_bars$, its clock z is reset. The controller may stay in the state $Lower\_bars$ as long as its invariant $z \leq 1$ holds, that is, for at most one time unit. Furthermore, the system may take the $lower!$ transition, leaving the state, if and only if its guard $z = 1$ is satisfied. This reflects the fact that it takes exactly one time unit between the arrival of the $approaches?$ signal and the sending of the $lower!$ signal.

The intuition behind the gate model (rightmost in Figure 1.4) is similar. When the gate receives a $lower?$ signal, a clock y is reset and the gate moves to the state $Lowering$. In this

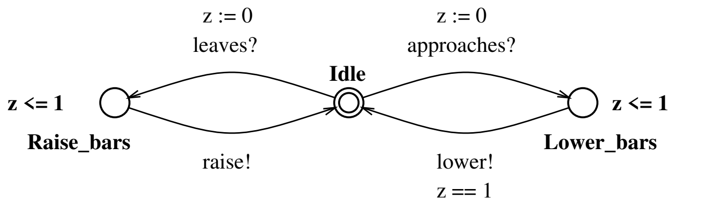

Figure 1.3: Timed automaton model of a railroad controller

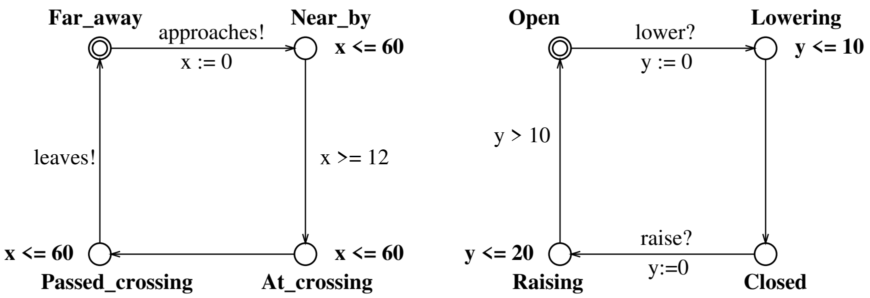

Figure 1.4: Timed automata modeling a train and a gate in a railroad system

state, it can stay for 10 time units at most, reflecting the fact that lowering the gates takes at most 10 time units. The transition going out of the state $Lowering$ is not labeled, which represents an internal step of the gate. Furthermore, in the state $Closed$, the gate waits for a $raise?$ signal to arrive and then it moves to the state $Raising$, where it can stay for strictly more 10 time units but less than 20.

The train is modeled with the same ideas. Note that the clock x is not reset on the transitions leading from $Near\_by$ to $At\_crossing$ and from $At\_crossing$ to $Passed\_crossing$. This means that the time to stay in $Near\_by$, $At\_crossing$ and $Passed\_crossing$ is less than 60 in total, which models the fact that it takes at most 60 time units between the arrival and the departure of the train.

A crucial property of the railroad system is that the bars have to be closed whenever the train is at the crossing. In terms of the model, we require that the gate is in the state $Closed$, whenever the train in the state $At\_crossing$. This property is definitely a part of the specification of any realistic railroad controller system. (Obviously, a complete specification takes many more aspects into account, such as the fact that the bars should eventually reopen when the train has passed.) Actually, one can prove automatically that the property mentioned above does hold for the timed railroad system (but not for the untimed system). Note that we have modeled only one train, so we cannot be sure whether the property also holds if more trains are around. The section on verification techniques below will explain how this can be done.

Parametric Systems

> The eternal mystery of the world is its comprehensibility.
>
> Albert Einstein

Parameters are used to describe a class of systems with the same structure but with different constants. Case studies in verification of a variety of applications show that almost all systems are parameterized in some sense. Parameters in the railroad crossing system are for instance the maximal time for the train to arrive at the crossing and the time for the controller to propagate a signal. Parameters in other systems can be the number of processes in the application, the communication delay, the clock speed, the network topology, etc.Usually, these systems work correctly for some values of the parameters and for others they fail. It is the aim of *parameter synthesis* to derive the exact conditions on the parameters that are needed for correct system operation. These conditions are usually called *parameter constraints* and knowing these constraints provides useful information for building correct systems. For the railroad crossing, the parameter constraints tell whether or not the railroad is still safe if a faster train is introduced.

This thesis restricts itself to timing parameters, as in the railroad example. More precisely, we consider **linear parametric timed automata**. These are timed automata where the guards and invariants may contain linear expressions over the parameters.

Example 1.2.3 Figure 1.5 depicts a parameterized version of the railroad crossing. We have replaced all timing constants by parameters. (Note that this need not be the case; for several applications it can be useful to have both parameters and constants.) Thus, the parameter $\mathrm{max\_lt}$ stands for the maximal time needed to lower the bars. The parameter $\mathrm{min\_rt}$ for the minimal time to raise them and $\mathrm{max\_rt}$ is the maximum time needed for doing so. The parameters $\mathrm{min\_dt}$ and $\mathrm{max\_dt}$ stand for the minimal and maximal time that the train remains at the crossing and st is the time it takes for a signal to be propagated by the controller.

Now, we are interested in the question asking for which values of the parameters the railroad is safe, i.e. for which values the gate is in the state *Closed*, whenever the train is at the crossing. One can show that this property holds for all parameter values such that

$$\mathrm{st} + \mathrm{max\_lt} < \mathrm{min\_dt}.$$

This is not unexpected, because in order for the gates to close on time, the signal has to be passed to the gate *and* the gate has to be closed *before* the train arrives, even if the train is very fast and the signal and gate very slow. Therefore, the maximal time for propagating the signal and closing the gates has to be smaller than the minimal time needed for the train to arrive at the crossing, which is exactly expressed by the formula above.

Probabilistic Systems

> '"God dobbelt niet!" zei hij.'
> 'Zei Einstein dat?'
> 'Het schijnt.'
> 'Geloofde Einstein dan in god?'
> 'Einstein geloofde in ieder geval niet in het toeval.'
>
> Connie Palmen, *De wetten.*

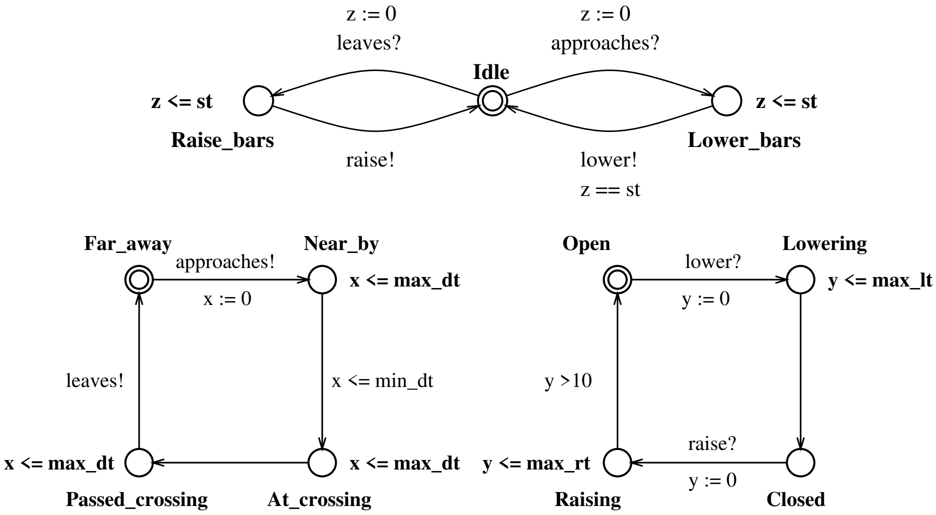

Figure 1.5: Parametric timed automata modeling a railroad system

Probabilities are as old as civilization. Egyptian excavations have brought up several perfectly shaped dice. In the Mahabharata, there is even a demi–god of dice and also the Bible mentions the use of lots to make decisions several times. It might therefore be surprising that it was not until the 16th century that a mathematical probability theory was developed [Hac75]. Pioneering work was done by Leibniz, Huygens and Pascal. Since that time, probability theory turned out to be a powerful means to model and analyze reality and rapidly entered in all fields of science. Today, one finds probabilistic models in physics, sociology, economics, epidemiology, etc.

Also in computer science, there are various applications of randomization[1]. We mention three of these. Probabilities can for be used to model unreliable or unpredictable behavior exhibited by a system. Secondly, stochastic mechanisms play a dominant role in performance evaluation of computer systems. and finally, randomized algorithms make clever use of coin flips or other probability mechanisms to obtain efficient computer programs. Below, we illustrate these applications of probabilities by means of the railroad example.

Note that the first two uses of randomization differ from the latter. When using probabilities to model an unreliable components or the performance of a component, they describe existing phenomena in reality. In randomized algorithms instead, they are deliberately introduced for the purpose of a more efficient solution.

Unreliable behavior

Stochastic mechanisms can be used to model unreliable behavior of a system or its environment. Some components of the system may inevitably fail or be overloaded at some points in time. The challenge is then to design a system with maximal reliability from those components. Usually, failures of system components are governed by a probabilistic mechanism and

[1] In this chapter, we use the words probabilistic, randomized and stochastic as synonyms.

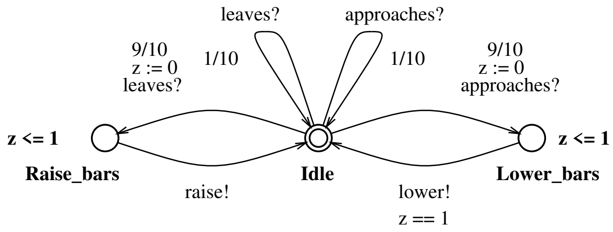

Figure 1.6: Probabilistic model of the controller

probabilistic analysis allows one to derive the probability of a failure of the whole system.

Example 1.2.4 In the railroad example, it is realistic to assume that catching a signal from a moving train is subject to faults. (Note that sending a signal from the controller to the gates, which are part of the same system, is much more reliable.) The system in Figure 1.6 models a controller that does not always receive the $approaches!$ signal sent out by the train correctly: with probability $\frac{1}{10}$ the controller misreceives the signal. This causes the controller not to move to the state $Raise\_bars$, but to remain in the state $Idle$ instead. With probability $\frac{9}{10}$, the signal is received correctly, in which case the system operates as before. As a consequence, the railroad system is not safe anymore and the probability that an error occurs is quite large. Actually, the probability that a signal is lost eventually equals one.

Several solutions exist to deal with this situation – leading to a more complex system design. For instance, the train may repeatedly send $approaches!$ signals. Since the probability that none of the signals arrives correctly is quite small (10^{-n}, if n signals are sent), the system becomes much safer.

Another possible solution would be to have the controller acknowledge the receipt of an $approaches?$ signal. If, after a number of trials, none of the signals has been acknowledged, the train could switch to an emergency scenario, for instance, slowing down.

Performance analysis

Performance analysis aims at evaluating a system in a quantitative sense. Several performance measures can be estimated, for instance the number of telephone calls a telephone system can handle on average, the mean time it takes between two subsequent system failures (e.g. rejected calls due to system overload) etc.Performance analysis is a field in its own right and this thesis focuses more on qualitative rather than quantitative aspects. It is, however, an important topic for present and future research to integrate both aspects in system analysis.

Example 1.2.5 Figure 1.7 depicts a railroad model that could be used in performance evaluation. In each state, the system either waits until a signal is received or it waits for some amount of time t, where t is drawn according to an *exponential probability distribution*. This means that the probability to stay in state s for more than t time units is $e^{-\lambda_s \cdot t}$. Here, λ_s is a parameter of the exponential distribution and, for each state, its value is given by Figure 1.7. The exponential distribution has nice mathematical properties, one of them being that the mean time a process stays in state s equals $\frac{1}{\lambda_s}$. For instance, the parameter of the

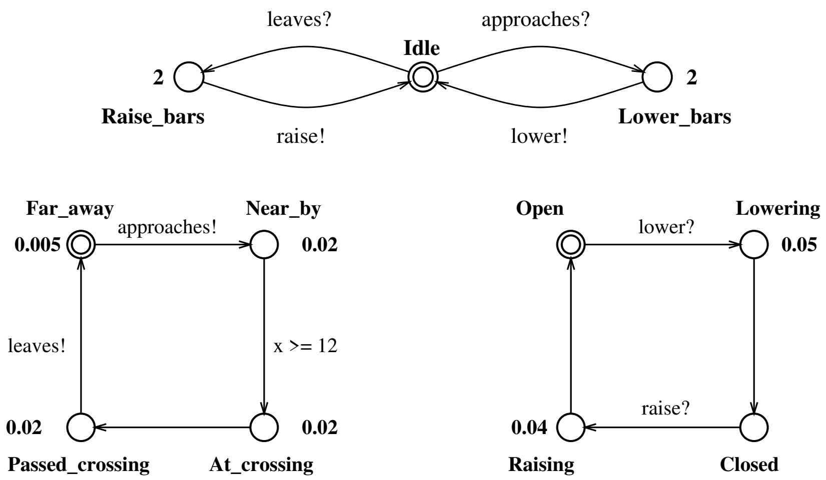

Figure 1.7: Automata modeling railroad system with exponential distributions

state Far_away equals 0.005, which means that, on average, the train stays there for 200 time units.

Note that, in the new model, the train, gate and controller need not meet the timing requirements imposed by the guards and invariants of Figures 1.3 and 1.4 anymore. An important issue, which can be tackled with techniques for performance analysis, is to estimate the probability to violate the safety property mentioned in Example 1.2.2.

Since the behavior of the model in Figure 1.7 is described completely probabilistically, we can compute the exact probabilities and expected times for many events. Apart from the probability to violate the safety property, we can for instance derive the mean time that the gate is in one of the states $Raising$, $Closed$ or $Lowering$, in which no cars can pass the crossing. This performance measure is relevant for the question whether or not traffic jams are caused because of the gate being closed too often.

Randomized algorithms

Thirdly, computer programs (or abstract versions of these, called *algorithms*) may contain random choices. Such programs may roll a dice and make a choice depending on the outcome of the cast. Such programs are called *randomized algorithms*. In a similar way, probabilities can be used in *communication protocols*. The latter are agreements on how several interconnected devices exchange information, for instance, how e–mails are transferred in the Internet. In this thesis, we consider a small part of the IEEE 1394 serial bus protocol. This protocol flips a coin to determine whether it is going to wait a long or a short time before sending a message.

Randomized algorithms are more powerful than non–randomized ones. They perform better in the following three aspects [GBS94]:

1. *Efficiency*, both in time and in space: the probabilistic hashing algorithm described in [GBS94] uses less memory than any known deterministic hashing algorithm. Hashing algorithms are important to store data efficiently.

2. *Symmetry*: it has been shown that there does not exists a solution to the *dining philosophers problem* in which all processes execute the same program, unless randomization is used [LR81]. The problem is relevant in cases of *resource contention,* that is, in order to divide a single resource (printer, processor) among several users (printer jobs, terminals).

3. *Computational power*: Lynch *et al.* [FLP85] have shown that the so–called *Byzantine agreement problem* cannot be solved by any non–randomized computer program. This problem requires several communicating processors to agree on the same value. Some processes may fail, in which case they do not have to agree. It follows from the results in [FLP85] that, no matter which non–randomized program the processors in the network run or how they communicate, agreement on the same value cannot be guaranteed. On the other hand, [GBS94] present a randomized algorithm solving the Byzantine agreement problem.

Probabilistic systems also have their drawbacks: randomized algorithms are harder to analyze than non–randomized ones and one's intuition often fails. This especially holds for probabilistic distributed applications, where a system contains several algorithms running in parallel. Even specialists in this field have proposed solutions that later turned out to be wrong, as has been pointed out for instance in [Seg95b]. This again emphasizes the need for rigorous formal methods in the design and analysis of probabilistic applications.

Some remarks

The probabilistic choice in the applications above can be either discrete or continuous. This thesis restricts itself to **discrete probabilistic choice.** The reason for this is that, among the applications above, we are particularly interested in randomized algorithms and communication protocols. These algorithms and protocols often contain discrete probabilistic choice. Moreover, the benefits of probabilities here are clear and so is the need for formal methods.

Besides probabilistic choice, also nondeterministic choice plays an important role in the verification of reactive systems, no matter whether or not probabilities are around. Hence, we need a model that combines probabilistic choice with nondeterminism. Such models have been developed recently and we consider the probabilistic automaton model from [Seg95b]. The differences and similarities between both types of choices are quite subtle and will be explained elaborately in Chapter 2.

On the other hand, performance analysis mostly studies models that combine discrete and continuous probabilistic choice, but which do not contain nondeterministic choice. Therefore, this thesis hardly addresses performance issues. As said before, it is an important topic for future research to combine techniques from functional analysis and performance analysis.

1.3 Verification Techniques for Reactive Systems

This section briefly explains the ideas behind the two verification techniques studied in this thesis. We explain model checking for timed and parametric systems and (Bi–)simulation relations for timed and probabilistic systems.

Model Checking

When using model checking to verify a system, one describes the system as a (labeled) transition system and expresses its correctness as a logical formula. Then one checks that the transition system satisfies the formula. What made model checking popular is the development of powerful and easy–to–use software tools, called *model checkers*, to perform this check completely automatically. Moreover, if the property does not hold, a model checker comes up with a counter example showing how it is violated. In this way, model checkers support debugging the model.

Model checking tools and techniques exist for a wide variety of specification logics and system models, including timed, probabilistic and parametric transition systems.

Below we show how model checking can be used to analyze the timed and parametric models of the railroad system.

Example 1.3.1 We have already seen that a key correctness property for the railroad crossing is that, if the train is at the crossing, then the gates at the crossing should be closed. This is expressed by the following logical formula Φ.

$$\Phi = \forall\Box(\mathrm{Train}.\mathit{At\_crossing} \implies \mathrm{Gate}.\mathit{Closed}).$$

Here, $\forall\Box$ means that the formula holds in every state of the system and $\mathrm{Train}.\mathit{At\_crossing}$ indicates that the train automaton (see Figure 1.4) is in the state $\mathit{At\_crossing}$. Likewise, $\mathrm{Gate}.\mathit{Closed}$ refers to the state Closed in the automaton Gate (Figure 1.3).

The models presented in Figure 1.4 and 1.3, as well as the property Φ are in the input format of the model checker Uppaal and we used this tool to check the property. Uppaal returned that this property holds for the system and therefore the railroad described by this model is safe. However, if we change the invariant $y \leq 10$ of the location $\mathit{Lowering}$ into $y \leq 11$, then Uppaal will tell us that the property Φ does not hold anymore and it will provide a sequence of steps leading to a situation where the train is in the state $\mathit{At\_crossing}$ and the gate in a state unequal to Closed.

Example 1.3.2 We can also verify parametric systems by model checking. Henceforth, consider the parametric railroad system in Figure 1.5. We are interested in knowing for which values of the parameters the railroad is safe and for which is it not. In other words, we wish to derive the exact conditions on the parameters for which the property Φ holds. If we feed the parametric system and the property Φ to the parametric extension to Uppaal, then the tool will generate the constraint $\mathrm{st} + \mathrm{max\_lt} < \mathrm{min\_dt}$ as the exact condition needed for Φ to be satisfied.

In this thesis, we use the model checker Uppaal to establish the correctness of the IEEE 1394 Root Contention Protocol. Uppaal is a model checker for timed automata with a convenient graphical user interface. Moreover, this thesis presents a model checking technique to synthesize timing parameters for linear parametric timed automata, as we did in the second example above. This technique has been implemented on top of Uppaal. Model checkers for probabilistic systems are relatively new and are not used here.

Simulation and bisimulation

In the previous section, we expressed (a part of) the specification of the railroad system by the formula Φ. One can also describe a specification by an automaton. In order to prove that a

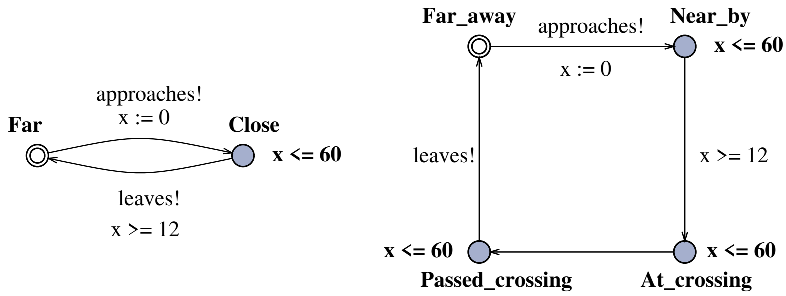

Figure 1.8: Bisimilar train models

system meets its specification, one shows that its external behavior is included in the external behavior of the specification automaton.

Different types of LTSs give rise to different notions of external behavior. Informally, the *external behavior* of an LTS, also called the *visible behavior*, is given by its sequences of external actions. The external behavior of a timed LTS also considers time passage as an external action and, in probabilistic LTSs, the probability on a sequence of actions is taken into account. Finally, the external behavior of parametric automata associates a set of action sequences to each parameter value. Note that internal actions and states do not belong to the external behavior of an LTS.

Proving behavior inclusion, that is showing that the external behavior of one automaton is included in the external behavior of another one, is a rather complex task. Therefore, simulation and bisimulation relations can be useful. These are relations that compare the stepwise behavior of two systems. When two systems are shown to be bisimilar, there is behavior inclusion. The idea behind bisimilar states is that each step one of them can take, can be mimicked by the others, as is explained below.

Example 1.3.3 Consider the leftmost automaton in Figure 1.8. We claim that it has exactly the same stepwise behavior as the train model leftmost in Figure 1.4, which is presented again in Figure 1.8 (rightmost), in order to make the comparison easier. The idea is that the state Far_away is equivalent to the state Far, i.e. they exhibit the same behavior, and so are the states $Near\_by$, $At\_crossing$, $Passed\_crossing$, and $Close$. This can be seen as follows; note that equivalent states are given the same color.

- In the states Far and Far_away, one can remain as long as desired.
- In the states $Near\_by$, $At\_crossing$, $Passed\_crossing$ and $Close$, one can stay any time less or equal than 60 time units.

This shows that the equivalent states have the same timing behavior. Now, we argue that they have the same stepwise behavior too. The idea is that if two states are equivalent and one can take a step from one state, then this step can also be taken by the other state (possibly via one or more internal transitions) and both steps lead again to equivalent states. This is exactly the idea of a weak bisimulation relation.

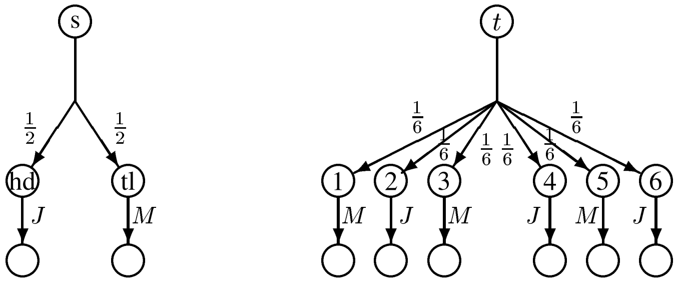

Figure 1.9: One coin flip is bisimilar with one dice roll

- In the state Far, one can move with an $approaches!$ transition to the state $Close$, which is a grey state, and so can one move in the state Far_away with an $approaches!$ transition to the state $Near\_by$, which is also a grey state. Note that both transitions reset the clock x. Therefore, an $approaches!$ transition moves from equivalent states to equivalent states.
- In each of the grey states, one can move with a $leaves!$ transition to a white state, provided that $x \geq 12$. In the states $Near\_by$ and $At\_crossing$, one has to take some internal transitions (respectively two and one) before taking the $leaves!$ transition. This does not matter, because these are internal transitions, which are not visible.

Therefore, the two train models in Figure 1.8 are, what is called, *weakly bisimilar*.

Since bisimulation relations are compositional, we obtain an equivalent railroad system if we replace the rightmost train model in Figure 1.8 by the leftmost model.

Apart from bisimulations, simulation relations play an important role in this thesis. Simulation relations can be considered "unidirected" variants of bisimulations. In a bisimulation relation, all states have to exhibit the same behavior. In contrast, if a state s is related to t by a *simulation* relation, then all steps of s can also be performed by t, but not necessarily the other way around. Thus, t may exhibit more behavior than s.

Probabilistic simulation and bisimulation

> Do not worry about your difficulties in mathematics. I can assure you mine are still greater.
>
> Albert Einstein

Simulation and bisimulations are also useful for proving probabilistic systems correct. We illustrate their use by two examples.

Example 1.3.4 Suppose one wishes to flip a fair coin to make a decision, but only a dice is available. For instance, one wants to decide who is paying for the beers, if head comes up then John will and otherwise Mary will). What one can then of course do is the following. Someone rolls the dice and John pays if an even number comes up and otherwise Mary does. This situation is shown in Figure 1.9, where the actions J and M indicate who is paying and where the names of the bottom states, being irrelevant, have been left out.

We claim that both processes are probabilistically bisimilar: the states hd, 2, 4, and 6 are equivalent: they clearly exhibit the same stepwise behavior and so do tl, 1, 3, and 5. But

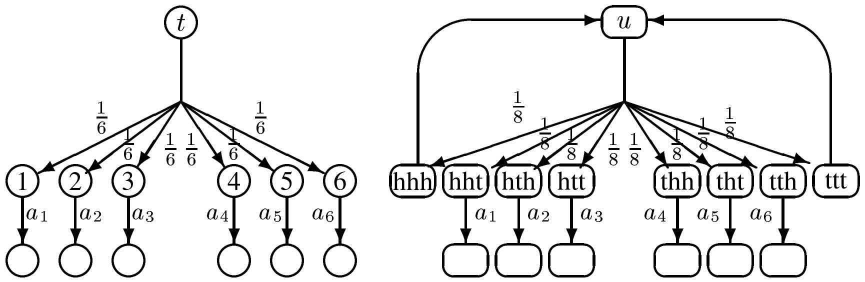

Figure 1.10: Many coin flips can be used for one dice roll

then also the states s and t are also bisimilar, because in both states, the probability to reach one of the equivalent states hd, 2, 4, 6 is equal to $\frac{1}{2}$ and so is the probability to reach one of the states tl, 1, 3, and 5. Thus, the basic idea in probabilistic bisimulation is that one does not consider the probability of a transition to reach an individual state, but the accumulated probability to reach one out of a set of equivalent states.

Example 1.3.5 Now consider the converse problem: one wants to decide on events $a_1, \ldots a_6$ (say who out of a group of 6 people will pay) with a fair dice, but only fair coins are available. Then one can toss 3 coins c_1, c_2, c_3, instead of a dice. If the outcome is either one of hht, hth, htt, thh, tht or tth then the corresponding person will pay. Here h stands for heads, t for tails and the outcomes of the three coins are listed in order. If either 3 heads or 3 tails come up, the coins are rethrown. This coin throwing process is repeated until the outcome is different from hhh or ttt. This situation is depicted in Figure 1.3.5. With elementary probability theory one shows that the probability that the coins will eventually be different equals 1. This means that the probability that one goes on coin flipping for ever is 0. (This probability is exactly 0, not "very close to 0!") Moreover, this implies that, eventually, each of actions a_i is taken with probability $\frac{1}{6}$. Thus, both processes in Figure 1.3.5 have exactly the same stepwise behavior. In other words, they are probabilistically bisimilar. In this type of bisimulation, one is allowed to combine arbitrary many internal transitions.

This thesis considers so–called **step refinements** and **hyperstep refinements**. These are the simplest kind of simulation relations for probabilistic systems that allow one to abstract from internal steps. These are relations that have the form of a function and require every step to be mimicked by a single step, except for internal transitions, which can be mimicked by remaining in the same state. Secondly, we develop several notions of *normed (bi–)simulation*. These are relations that also allow a limited abstraction from internal transitions and we prove that one can check efficiently (polynomial time and space) whether or not two states are normed bisimilar.

1.4 Overview and Results

The search for truth is more precious than its possession.
Albert Einstein

This thesis is divided into 7 chapters. Chapter 2 provides the basic model that we use throughout this thesis and Chapter 3 repeats some basic probability theory. The main results are stated in Chapters 4 – 7. These chapters can be read independently from each other. Since they originate from a collection of papers, the notations and concepts in these chapters are sometimes slightly different from the ones in Chapter 2. In some cases, definitions are repeated.

Main Results

The main results of this thesis can be summarized as follows.

- We present a testing scenario that justifies the definition of external behavior of (closed) PAs proposed in the literature. (Chapter 4)
- We develop a type of probabilistic (bi–)simulation that abstracts from internal computation in some sense and that can be computed efficiently. (Chapter 5)
- We present a model checking technique capable of synthesizing parameter constraints for the correctness of timed systems. (Chapter 6)
- We prove the correctness of the IEEE 1394 Root Contention Protocol using techniques from previous chapters and analyze the probabilistic, timing and parametric behavior of this protocol. (Chapter 7)

Summary

Chapter 2 recalls the probabilistic automaton (PA) model from the literature. This model has been developed for the modeling and verification of systems with discrete probabilities. We use this model throughout the thesis to formulate our verification techniques for probabilistic systems. Moreover, we carry out the verification of the Root Contention Protocol in this framework.

Chapter 2 explains the basic theoretical concepts and ideas behind this model. In particular, we show how PAs can be used to model systems with probabilities and how the basic concepts standard LTSs can be extended to PAs. We elaborate on the rather subtile combination of nondeterministic and probabilistic choice in the PA model. Moreover, we discuss the behavior of a PA, which is defined as the set of its trace distributions. Furthermore, we treat the notion of parallel composition, an extension of the PA model with time and two simulation relations for PAs. Finally, we give an overview of other models for the analysis of probabilistic systems.

As explained in Chapter 2, the implementation relation that has been proposed for (closed) PAs is trace distribution inclusion. This means that we say that one PA is a correct implementation of another if each of its trace distributions is also a trace distribution of the other PA. Chapter 4 justifies, for closed PAs, this implementation relation via a testing scenario. This comprehends a notion of observability for these automata based on relative frequencies. In particular, we define how probabilities can be "observed," for instance, how one can, distinguish between a fair and an unfair coin.

We analyze the outcomes of a number of independent runs of the PA with statistic methods. More precisely, we use the classical theory of hypothesis testing to determine whether a given list of outcomes is likely to be produced by that PA. If so, we consider this list to be an observation of the PA. The main result in this chapter states that the observations of one PA are included in (are equal to) the observations of another PA if and only if its trace distributions are included (equal to) those of the other PA.

Chapter 5 is devoted to simulation and bisimulation relations for PAs. First, we recall several (bi–)simulations known from the literature. Then we introduce a new type of relations, called *delay (bi–)simulation*, for which we consider four variants. These relations abstract from internal transitions only in a limited way, but what we gain is efficient algorithms to compute the (bi–)similar states automatically. In technically terms, delay (bi–)simulation can be decided in polynomial time and space. Furthermore, we give an alternative characterization of the delay (bi–)simulations in terms of norm functions.

In Chapter 6, we present a model checker that is able to synthesize parameters for linear timed parametric automata (LPTAs). These are timed automata where the clock guards and state invariants may contain linear equations over the timing parameters. The tool is an extension of the existing model checker Uppaal and compares favorably with similar tools. Moreover, we present an extensive elaboration of the theory behind our implementation. In particular, we present a correctness proof of the model checking algorithm we implemented.

Furthermore, we identify of a subclass of parametric timed automata (called L/U automata), for which the parametric model checking problem is much easier.

Probabilities, timing and parameters come together in the verification of the IEEE1394 Root Contention protocol. This protocol has become a popular case study in formal methods and we start Chapter 7 with an overview of several approaches to its verification. Then we present a protocol model that includes real–time, probabilistic and parameterized aspects of the protocol. The verification of this model is carried out manually, using the step and hyper step refinements. Since the communication in this model was discovered to be too abstract, we also present an enhanced protocol model. For this model, the real–time and parametric properties have been investigated, because in this respect the enhanced model differs from the first model. The verification has been carried out mechanically, using the model checker Uppaal. First, the parameter analysis has been done "experimentally" by checking a large number of instances and later, the model has been fully explored by the parametric extension of Uppaal.

History of the Chapters

This thesis is based on a collection of papers. Below, we mention their origins.

Chapter 2 This chapter explains the basic theory of probabilistic automata, which has been developed by Segala [Seg95b]. It contains few new technical results, except for the step refinement and hyperstep refinement relations. These appeared in [SV99b].

Chapter 3 This is a short chapter introducing a few concepts and notations from probability theory.

Chapter 4 The testing scenario presented in this chapter have been developed in cooperation with Frits Vaandrager and will appear as [SV02b].

Chapter 5 The work on (bi–)simulations for probabilistic systems reported in this chapter is joint work with Christel Baier. The notion of delay bisimulation version has been reported on in [BS00].

Chapter 6 The parametric extension to the model checker Uppaal is joint work with Thomas Hune, Judi Romijn and Frits Vaandrager. A shorter version has appeared in [HRSV01] and the full version will be published as [HRSV02].

Chapter 7 The case study described in this chapter has been reported on in several papers. The overview of the approaches to the verification of RCP is presented in [Sto01]. A discrete time verification, which was a starting point to further verification, appeared in [Sto99a]. The first verification described in this chapter was carried out together with Frits Vaandrager and was published in [SV99b]. The second verification described in this chapter was done in cooperation with David Simons and appeared as [SS01]. The third verification appeared as a part of [HRSV01].