1

Components of a Theory

Software engineering is about specification, design, implementation, and maintenance of software artifacts. Like any engineering profession, it requires various kinds of knowledge and skills. One of the necessary ingredients is a theory that makes it possible

- to formulate, discuss, and reason about the properties that are essential in software artifacts,
- to specify and design software systems that have the desired properties, and
- to verify that software designs actually meet the given requirements.

The characteristic properties that such a theory should help to deal with are those of the *dynamic behaviors* that are generated by the execution of software.

Since it is software that has made it possible to create artifacts with involved dynamic behaviors, managing this kind of complexity has become an issue only with the proliferation of software. The same problem arises, however, in any engineering of complex systems. Therefore, a theory that helps in managing the logical complexity of dynamic behaviors is also applicable more generally. Due to the non-physical character of software, software engineering faces the problem, however, in a pure form, where physical theories play no essential role.

Dealing with the complexity of dynamic behaviors is what this book is all about. The main emphasis is on a theoretically justified specification and design method, which supports incremental derivation of operational specifications that have the desired properties. The specific focus of this book is on *reactive systems*, i.e., on systems that are in continual interaction with their environments. By the environment we understand in this context human users, physical environment, and other reactive systems that interact with the system under consideration.

Although the attribute 'reactive' emphasizes the most distinctive characteristic of the systems that we are interested in, there are other frequently used terms that characterize some more technical aspects of such systems:

- Reactive behavior is usually associated with *embedded systems*, in which software is an integral part of devices that have been designed for specific purposes. The hardware and the software of such dedicated systems have to be designed together, which is usually referred to as *codesign*.
- Real-time properties are often crucial in the requirements for reactive systems, in which case the term *real-time system* is used. Depending on whether real-time requirements are essential for correct behavior, or are formulated as statistical requirements that have to be satisfied on the average, real-time systems are called *hard* or *soft*.
- When a system consists of cooperating subsystems that are physically or even geographically distributed, it is called a *distributed system*. The subsystems of a distributed system are often reactive, and real-time requirements may also be associated with them.
- The purpose of a real-time system may be to *monitor* and *control* some physical phenomena in its environment. In addition to discrete states, the specification of such a system may need to refer to physical quantities that change as continuous functions of time. Such systems are referred to as *hybrid systems*.

The purpose of this introductory chapter is to outline the components that are needed in a theoretically justified approach to reactive systems. The structure of the chapter is as follows:

- In Sect. 1.1 we briefly analyze the relationship between theory and practice in software specification.
- Section 1.2 outlines some general requirements for a theory that would provide comprehensive support for the specification and design of reactive systems.
- Section 1.3 gives a brief outline of the rest of this book.

1.1 The Role of Theory

We start by inspecting the role of an underlying theory in the specification and design of software.

1.1.1 Theory and Practice

The history of software engineering is still very brief. In just a few decades, software has become a ubiquitous basic technology for the implementation of complex systems. The size and complexity of software artifacts have grown immensely at the same time, and the qualitative character of software and its applications have also changed rapidly.

These advances in the state of the art have been largely based on practical rather than theoretical understanding of software. Also, due to the speed of development, theoretical understanding of software has not had much time to

mature. As a result, the need of theory is often underestimated in the practice of software engineering.

Although the driving forces for theoretical and practical developments are different, useful theory of software cannot be developed in isolation from understanding what the essential practical problems in software and software engineering are. Otherwise, theoretical work would remain as useless formalization. On the other hand, the need for a solid underlying theory is also becoming more and more evident in software practice. We are already facing the situation where some software cannot – or should not – be developed without theoretically justified confidence in its behavior.

1.1.2 What Is Theory?

In a broad sense, anything that provides understanding at some level of generality can be called theory. The Oxford English Dictionary explains theory, among other things, as "systematic conception or statement of something; abstract knowledge, or the formulation of it."

For theoretical understanding of practical artifacts it is important that a theory supports thinking in terms of *abstractions*, which allows us to omit those aspects of the reality that can be managed trivially, and to concentrate on those aspects that at least potentially may cause problems. Different theories may support different kinds of abstractions, which is useful for taking multiple views of the same artifacts. For instance, focusing on logical behaviors abstracts away all physical characteristics, which are also important in all digital devices.

In particular, a theory of software (or any systems with behavioral complexity) should provide a rigorous basis for

- *specification*, i.e., formulation of the required behavioral properties,
- *design methods*, i.e., systematic methods for developing systems that meet the given requirements, and
- *verification*, i.e., reasoning on whether the required properties are, indeed, satisfied by the design.

In addition, effective use of a theory should also be helpful for

- *validation*, i.e., checking that the design (or its eventual implementation) meets the actual needs from which the requirements were derived.

1.1.3 Reality and Abstractions

Every theory has some inherent limitations. Since abstractions do not model all properties of reality, a theory can be used only for those properties for which it has been designed.

Since a theory deals with abstractions, it does not say anything about the reality itself. Therefore, theoretical results and proofs are relevant for reality

only as far as this obeys the basic assumptions that have been made in the theoretical models. The validity of such assumptions will always remain beyond the reach of mathematical proofs. In addition, no formal proof of software can show that the original informal requirements have been correctly formalized. This is why formal methods can never remove the need for validation.

Although these remarks about distinguishing between the reality and abstract models are a truism, there is considerable danger for confusion between the two, especially in connection with software.

A written program seems to belong to reality, as its execution on real computers gives rise to those real phenomena that we are interested in. We are not, however, interested in these physical phenomena as such, but on their interpretation as computations. Since widely different physical representations can be used for computations, a program is, in fact, an abstraction that is independent of the actual phenomena that arise in its real executions, and it can therefore be subjected to formal analysis and proofs.

The relevance of any proofs of programs always depends on some assumptions, like absence of ambiguities in the programming language that is used, correctness of all system software that it relies on (compilers, operating systems, database systems, communication protocols, etc.), fault-free operation of the hardware involved, and satisfaction of (often implicit) assumptions on ranges of numbers, amounts of data, frequencies of communication, etc. All these are non-trivial assumptions, as is shown by the multitude of bugs in commonly used system software, famous design errors in hardware, and the Y2K problem, for instance.

Worse still, abstractions of interactive behavior make assumptions on all partners of interaction. Therefore, reactive systems cannot be specified independently of modeling the environments in which the systems are intended to be used. Although assumptions on the environment can be formalized in more and more detail when increased confidence in the satisfaction of these assumptions is required, it is important to understand that the reality itself always escapes full formalization.

For all these reasons, theoretically well-justified formal methods and proof techniques can never replace testing and validation. Instead, the two kinds of approaches complement each other and should be used in tandem to increase our confidence in complex systems.

1.1.4 Role of Theory in Engineering

Natural sciences and the associated mathematical theories provide the basis for theoretical understanding in traditional engineering disciplines. In practical engineering work one may not need to go to these fundamentals all the time, but it is always possible to resort to them when necessary.

These fundamentals are considered essential in engineering education, and one would not think of teaching only practical engineering skills without associated theoretical understanding. This shows that the importance of a uni-

versally accepted theoretical framework is recognized, even though practical engineering standards and industrial practices may be more important in everyday engineering work.

In software engineering the situation is a bit different. Obviously, physics and calculus do not provide a useful basis for understanding software. Since no comparable and generally accepted theoretical basis has been agreed on, software-engineering education typically concentrates on programming skills, available tools, standard practices, organization and leadership of software projects, etc. Instead of a general theory of programs, discrete mathematics is taught, as well as specialized mathematical theories that are relevant in computer science, like those needed for understanding compilers, computability and complexity issues, efficiency of algorithms and data structures, etc. Although all of these are essential for the software-engineering profession, they are no replacement for an underlying theory of software.

1.1.5 Emerging Need for a Theory

The need for a general theory depends greatly on the size and complexity of the artifacts to be constructed, and on the severeness of the potential consequences of errors in them. Just plain logical thinking without any special theory is sufficient for understanding short and simple programs. In larger programs, simple programming errors can be avoided by using appropriate tools, and an error can often be understood and corrected without much theory, once somebody points out under which circumstances it occurs.

This has led to a situation where the significance of high-level programming languages and other tools, maintainable software architectures, teamwork skills, systematic testing methods, good management of the software process, etc., is recognized in software-intensive industry, but no stringent need is felt for competence in an underlying theory of software.

With growing requirements for the degree of confidence in software, the situation is, however, becoming unsatisfactory. This is the case, in particular, with software that is used to control life-critical systems, and also with embedded consumer products like automobiles, where human control more and more often takes place through software interfaces. Networking of systems has also greatly increased the need for trustworthiness even in the presence of deliberate misuse and malicious attacks.

Although understanding of theoretical foundations is only one aspect in producing high-quality software, its importance will necessarily increase when the field becomes more mature, and when consumers become more conscious about software quality.

Review Questions

Question 1.1.1 What are the main purposes for which a theory of software can be used?

QUESTION 1.1.2 What makes interactive behaviors more difficult to specify than non-interactive behaviors?

QUESTION 1.1.3 What makes software engineering fundamentally different from other engineering disciplines?

1.2 Parts of a Comprehensive Theory

In computer science and software engineering, people are used to develop and deal with different kinds of abstractions. The problem is therefore not in the lack of useful abstractions, but in their incompatibilities, and in the lack of a comprehensive theory that would support them.

In this section we discuss different kinds of abstractions that should fit together in a balanced manner and without incompatibilities, in order that a theory would be useful for the practice of software specification.

1.2.1 Spectrum of Abstractions

Specification formalisms are often classified into *property-oriented* and *model-oriented* ones. The purpose of the former is to express required properties independently of how they can be implemented. In the latter, specifications are given as *operational models*, which can be understood as 'abstract implementations'.

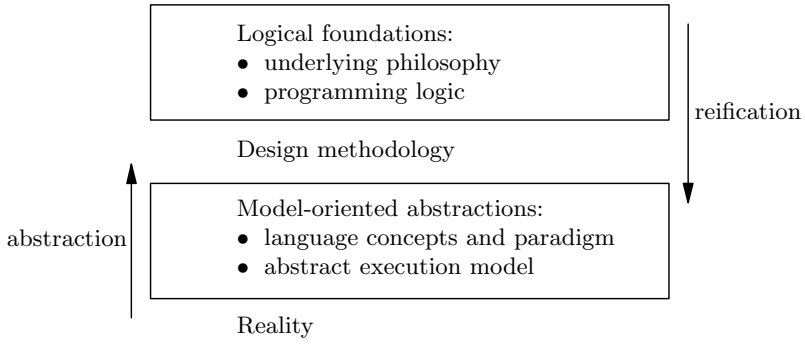

Fig. 1.1. Parts of a comprehensive theory

A practical theory cannot be limited to either category alone, since it has to support both kinds of views. Figure 1.1 outlines the spectrum of abstractions that a successful theory has to address. *Logical foundations* constitute the high end, which has to support property-oriented abstractions. *Design methodology* has been placed in the middle, between logical foundations and

model-oriented abstractions, since it deals with the design of operational models and requires language support for this, but also needs to be based on solid logical foundations.

1.2.2 Development of Abstractions

Two opposite directions can be seen in the process of developing abstract concepts for software, *abstraction* and *reification*, as shown in Fig. 1.1.

Abstraction

Historically, and in programming education, the dominating direction for developing understanding of software is to start with concrete programs and to design abstractions in the *bottom-up* direction. Real computer architectures and machine languages are then taken as the reality, for which useful higher-level abstractions are gradually developed in terms of language ideas and language-related programming concepts.

This approach has been very successful, and has led to effective machine-independent *languages* and *tools*, to reusable *design patterns*, and to informal *design methods* for managing the complexity associated with software. In particular, these abstractions help in the *architecture* of software, i.e., in structuring it into units that have well-defined properties and relationships.

Together with a specification of what a program is intended to accomplish, such concepts make it easier to check whether a final product meets the intentions. However, this abstraction process assigns no abstract meaning to programs as such. Therefore, these concepts are not directly suited for addressing semantically relevant relationships between programs, such as program equivalence.

On the other hand, being independent of the meaning of programs, these abstractions are relatively insensitive to what kinds of essential properties programs are interpreted to possess. As an extreme example of our freedom in this interpretation, consider being interested only in the side-effects on console lamps, for instance. In this connection it may be interesting to recall that the machine instructions of some early computers had side-effects on a loudspeaker, which allowed us to interpret the meaning of a program as the tune played by its execution.

Reification

The reification process starts with the fundamental question of what the essential properties of programs really are. Mathematical abstraction of the meaning of programs allows us to discuss rigorous reasoning on these properties, and can also give a solid basis for design methods.

One of the problems with this *top-down* direction is that it is possible to start with different kinds of underlying philosophies. It may, in fact, be

the case that no single philosophy will ultimately be good for all kinds of programs. The essential properties in mathematical subroutines, interactive systems, and real-time control software, for instance, seem to be very different, and specialized theories for them may therefore give better results than a single unified theory of programs.

Another problem with top-down development of abstractions is how to integrate the resulting rigorous methods smoothly into the software process. In particular, bottom-up development of abstractions has led to extremely complicated languages like C++, and gaps and incompatibilities are therefore unavoidable between mathematically manageable basic concepts and those that are currently advocated in practice.

1.2.3 Execution Models

Abstract *execution models*, or *abstract machines*, provide a possible first step in providing abstractions of software. In the specification of software, such models are important for operational specifications that can be executed, simulated, or animated. To allow effective reasoning, the models should, however, be simpler than real execution of programs on existing computer architectures.

Turing machines, *finite automata*, and *Petri nets* are examples of abstract execution models that have been designed for different theoretical purposes. The notion of Turing machines has been used successfully as a basis for the theory of computability and for complexity theory, but it would be totally inapplicable as a basis for discussing the specification and design of software. Finite automata, on the other hand, are suitable for modeling special classes of software, like communication protocols, for instance, and Petri nets have been designed for the modeling of parallelism in concurrent systems.

The execution model to be used in this book is a simple *action-oriented* execution model, which will be explained in Chap. 2.

To distinguish between executions of real systems and those in a model, the former will be called *computations* in the following, whereas those in abstract machines will be referred to as *executions*.

Formal execution models allow rigorous reasoning on executions in abstract machines, but they do not assign abstract mathematical meanings to the models. Therefore, they may be useful in the modeling of software, but as such they are insufficient to support a comprehensive theory. In particular, without additional information they cannot be used to address the fundamental question of whether two different systems are equivalent or not, or whether one of them models a correct implementation of the other.

1.2.4 Language Concepts

Programming concepts and *paradigms* provide the conceptual basis for the constructs and facilities in programming languages. Therefore, the history of

high-level languages and the associated paradigms reflects the development of our conceptual understanding of programs.

Programming language mechanisms are, however, often treated as if they would be the primary objects of concern, instead of the underlying concepts that they should be designed to reflect. For instance, extending programming languages to deal with concurrency has often been discussed as if this would primarily be a question of language constructs and their implementation mechanisms, not a question of how concurrency affects the fundamental properties of programs.

In each case, language concepts are an important step beyond execution models. In particular, high-level programming languages have declarative features, which allow much freedom in algorithmic implementation. Programming language concepts therefore also provide useful abstractions for operational specifications, although the latter may have features that prevent automatic compilation into executable programs, in general.

One of the idealistic goals in developing high-level languages has been that they would by themselves be sufficient for discussing the properties of programs, so that no higher-level abstractions would be needed on top of them. For large and complex programs this has, however, turned out to be unrealistic.

One reason for this is that, although high-level languages aim at simplicity, they are not at all simple. In fact, when all the ingredients for a modern high-level language are put together, their combined effects are bound to lead to complexities and ambiguities that are difficult to foresee and to manage. For this reason, some theoreticians have often argued for much simpler programming languages, but the practice has not followed their advice – and often with good reason. In each case, much of the complexity of current programming languages has to be abstracted away in order to obtain a practical theory.

A good example of current trends in programming languages is object orientation, which has proved to be a very successful paradigm. However, if the facilities in object-oriented programming languages are adopted in specifications as such, without abstracting away some of the associated complications, specifications are not any easier to reason about than programs.

1.2.5 Underlying Philosophy

Each theory has an *underlying philosophy*, which determines informally the semantic properties that one wishes to express and reason about.

One of the fundamental distinctions to be made at this level is whether the purpose of executions in a model is thought of as transforming input to output, or as continued interactions between a system and its environment. Depending on this distinction, the semantics of an approach is either *transformational* or *reactive*.

In this connection it should be noticed that a program itself is always just a program, not a transformational, reactive, real-time, or some other kind of a program. Such distinctions are not in the programs themselves, but in the theories that are useful for reasoning on their properties. This is analogous to having just one physical reality, but different kinds of physical theories, like classical Newtonian physics, theory of relativity, and quantum mechanics. Different theories of software make it possible to concentrate on different kinds of properties, and the properties that are crucial depend on the intended use of a program in its intended application environment.

Transformational Philosophy

With *transformational* semantics, a program can be visualized as a black box, which, for any input x, determines a corresponding output $f(x)$, as illustrated in Fig. 1.2. Instead of a purely functional correspondence between input x and output $f(x)$, a specification may also allow several alternatives for $f(x)$, and for some x it may allow the program to give no output at all.

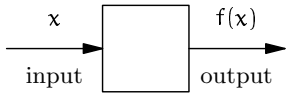

Fig. 1.2. A transformational system as a black box

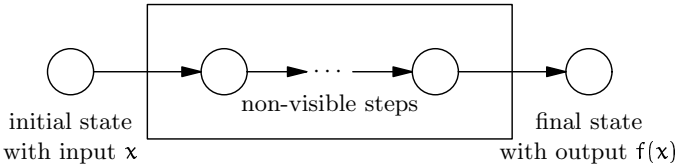

Fig. 1.3. Illustration of a transformational black-box execution

In an operational model of a transformational system, computations are modeled to proceed in *discrete steps*, starting from an *initial state*, where input x has already been read in. Intermediate steps take place in a black box and are therefore not visible. If the execution terminates, output $f(x)$ is eventually available in the *final state* as shown in Fig. 1.3. Nonterminating and aborted computations give no value for $f(x)$.

Reactive Philosophies

With *reactive* semantics, a system is assumed to be in continual interaction with its environment. Described as a black box, the input–output relationship

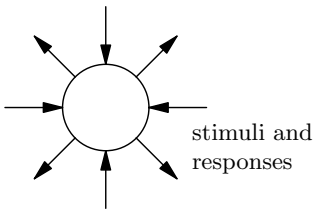

Fig. 1.4. A reactive system as a black box

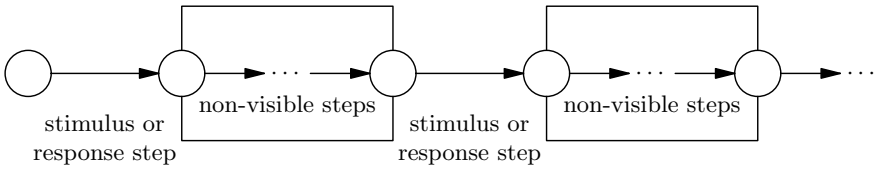

Fig. 1.5. Illustration of a reactive black-box execution

of transformational systems then generalizes to *stimuli* and *responses* with more complex causal–temporal relationships (see Fig. 1.4).

In an operational model of a reactive system, computations are normally understood to be nonterminating. In *interleaving models*, stimulus steps, response steps, and non-visible internal steps are all interleaved as illustrated in Fig. 1.5, i.e., all steps are assumed to be taken in some sequential order independently of the 'execution agents' by which they are performed. In contrast, *true concurrency* means that the concurrency of execution steps is also modeled. Instead of total ordering, this leads only to a *partial order* between the steps in an execution.

Under the so-called *synchrony hypothesis*, which essentially states that the system is 'infinitely fast' in comparison to its environment, no new stimuli can appear while a response is being computed. This leads to simpler models, since it is then possible to consider a reactive execution as a sequence of transformational executions. The initial state of each execution is then, however, affected by the history of previous executions.

In principle, execution steps can be understood either as *state changes* or as *events* with identification labels. Depending on this choice, an approach is called either *state-based* or *event-based*.

In general, there are several possible continuations for a given prefix of a reactive execution. Therefore, another basic question in reactive philosophies is whether one wishes to express properties of individual *sequences* of executions or of the whole *trees* that contain all possible continuations of an initial prefix. Depending on this choice, an approach is called *linear-time* or *branching-time*. The most obvious limitation of linear-time approaches is that *stochastic properties* of possible executions cannot be formulated as properties of individual linear executions.

Since different approaches to the modeling of reactive systems differ in their basic philosophies, they cannot be directly mapped into each other. This has caused some misunderstanding between their proponents. Such misunderstanding is often a sign of confusion between reality and a theory. Once one learns to think in terms of a given theory, one starts to consider only those aspects of the reality to be important that this theory is able to describe, and tends to ignore those aspects that have been abstracted away. Since different theories abstract away different kinds of properties, it may then be that only one's own theory can express what one considers important.

In choosing between different alternatives for a reactive philosophy, decisions are mostly based on intuition and subjective preferences. Ultimately, to understand the consequences of such selections, one should compare all aspects of fully developed theories, including their support for languages, tools, and design methods. An important point that needs to be understood in this context is that increasing the expressiveness of a formalism also adds to its logical complexity, and the main enemy of intellectual management is unnecessary complexity.

The choices on which the theory of this book is based are the following:

- The approach is '*truly reactive*' in the sense that it is not based on the synchrony hypothesis.
- The approach is *state-based*, although one can also see some event-oriented flavor in it.
- Reasoning in the approach is based on the *interleaving* model, which is simpler than true concurrency, but can still also be used for the modeling of distributed concurrency.
- The approach is *linear-time*, which means that specifications determine properties that must be satisfied by all execution sequences.

1.2.6 Programming Logic

Programming logic is a formal system for expressing properties of programs and to reason about them. In connection with state-based reactive philosophies, *temporal logics* are the primary vehicles for this. The choice in this book is a variant of linear-time temporal logic called *temporal logic of actions (TLA)*, which will be discussed in Chaps. 3 and 4.

In principle, an expression in a programming logic is a specification for a software system in the sense that it expresses the logical meaning of such a system.[1] In other words, expressions in the logic constitute the *semantic domain* for the systems. Two systems are equivalent if they have the same meaning in this logic, and a system is a correct implementation of a given specification if its meaning logically implies the specification.

[1]To be more precise, the logical expressions that will be used in this book describe not only software, but any reactive systems together with their intended environments.

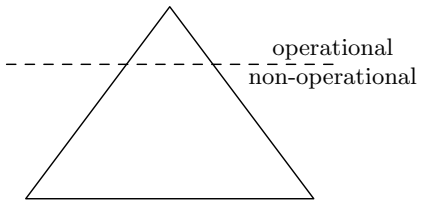

Fig. 1.6. The 'iceberg' of specifications

No matter how the logical meaning of systems is defined, if the semantic domain is restricted to correspond to operational systems only, it does not possess mathematical properties that would be easy to manage. However, by allowing the domain to contain also non-operational 'meanings' that have no operational interpretation, mathematically more elegant structures can be achieved. This is illustrated in Fig. 1.6 as an 'iceberg', where the tip corresponds to those specifications that have an operational interpretation. The purpose of a specification process is to lead to specifications with their meanings within this tip, but the full range of the 'iceberg' needs to be available during this process.[2]

1.2.7 Design Methodology

As such, *design methods* do not provide additional abstractions to a theory. However, as shown in Fig. 1.1 (p. 8), they have a central position in a practical theory, providing a link between logical foundations and model-oriented abstractions.

Traditionally, design methods have been developed with a view on model-oriented abstractions that can be supported by programming languages and various kinds of tools for manipulating software modules. This means that the emphasis has been on *structural* aspects of software, and on issues of *syntactic* compatibility.[3] In particular, attention has been paid to structural modularity, encapsulation of design decisions within modules, and to the design of module interfaces. The role of design methods has then largely been in providing guidelines for proper module design, in graphical illustrations for the structural aspects of the design, and in systematic documentation of design decisions.

[2]The same phenomenon is well known from extending the set of natural numbers to real and complex numbers. In electrical engineering, for instance, one frequently needs complex numbers in calculations, even when the final results are known to be real.

[3]In syntactic aspects we also include 'static semantics', which makes it possible to check that variables, objects, modules, etc., are utilized in accordance with their definitions.

The main problem with such design methods is that no *meaning* is associated with the system under design, except in terms of some scenarios of the intended 'dynamic properties'. Since such methods support rigorous inspection of only syntactic and structural properties, until the dynamic properties have been determined by an essentially implementation-oriented description, the slogan 'correctness by design' remains an empty phrase with them.

This problem can be overcome if design methods have a solid foundation in programming logic. Such a foundation makes it possible to reason about the semantic properties of a design already before any implementation-oriented descriptions are available.

Design methodology is, in fact, a crucial part in integrating the various aspects of a practical theory. It has to be rooted in the logical foundations of the approach, but it also affects the concepts of modularity that need to be supported by the design language. In fact, design methods that are supported by a theory reveal whether the abstractions at the different levels fit together in a reasonable manner. It should not be expected, as it is often done, that one could build a useful approach by taking a heterogeneous set of abstractions and associated tools, and constructing artificial bridges between them to overcome gaps and incompatibilities.

In other words, a good theory is not for constructing tools by which theoretically justified quality could be added to an arbitrary software process. A theory is a basis for thinking and understanding, and it necessarily also affects the software process, in particular the methods and ways of thinking that are used in early stages of specification and design. This is perhaps the biggest obstacle in adopting a theory in practice.

Review Questions

Question 1.2.1 Why is an execution model insufficient as such as a basis for a theory of programs?

Question 1.2.2 Why is any classification of programs (into transformational, reactive, and real-time programs, for instance) actually a classification of theories, not of programs?

Question 1.2.3 What is the difference between transformational and reactive philosophies?

Question 1.2.4 What is meant by an interleaved execution model?

Question 1.2.5 What is meant by the synchrony hypothesis?

Question 1.2.6 What is the difference between state-based and event-based approaches?

QUESTION 1.2.7 What is the difference between linear-time and branching-time approaches?

QUESTION 1.2.8 Why is it reasonable that the semantic domain also contains other 'meanings' than those that have operational interpretations?

QUESTION 1.2.9 What is the role of design methodology in a theory of programs?

1.3 The Structure of the Book

A comprehensive theory for reactive systems will be described in this book. The core chapters of the book have been grouped into three parts. These three parts are preceded by Part I, Prologue (this chapter), and followed by Part V, Epilogue (Chap. 11), which take a more general look at the characteristics of the approach.

Although mathematical concepts and results are effectively utilized in the book, formal theorems and proofs are avoided in the presentation. At the end of most sections, some of the key points are reiterated in the form of simple review questions given to the reader, and exercises of varying difficulty are also given. Each chapter ends with some notes on related history and literature.

No single running example was found suitable to illustrate the different topics discussed in the book. Instead, example specifications of varying size and complexity are given throughout the book. Although these are only 'academic exercises', the reader is encouraged to study them also in those chapters that he/she will not otherwise read carefully. Although the examples are mostly placed after the text that they are supposed to illustrate, it may often be wise to study them in parallel with the text.

1.3.1 Part II: Fundamentals

Part II of the book addresses the fundamental ideas of the approach and consists of three chapters.

Since model-oriented abstractions are the most natural starting point for software engineers,

- Chapter 2 introduces the *execution model* of *action systems*, which provides the basis for operational interpretation of specifications in this theory.

This execution model is *action-oriented* in the sense that execution consists of an interleaved sequence of *actions*, which are considered to be *atomic* units of execution. To fulfill its role in the theory, the execution model is very simple, and has no built-in support for any program structures. For instance, unlike commonly used execution models, it has no inherent bias towards sequential control threads. As such it provides, however, a suitable basis for operational

specification of reactive systems, and of any systems in which concurrency and distributed execution is essential. Although the execution model could also be used as a basis for a programming language, this is not a relevant question in the context of this book.

To give a solid basis for the theory,

- Chapter 3 is devoted to the *logical foundations* of the approach.

As the logical basis we take temporal logic of actions (TLA), which is a variant of linear-time temporal logics. TLA is used here to express and reason about properties of 'closed systems', where the environment of a reactive system is also included. An important point in fitting the different parts of the theory together is that the action-oriented execution model provides a natural operational interpretation for TLA expressions in a certain canonical form.

To help readers who do not have strong background in logic, but who would like to understand what it means to carry out formal proofs in detail,

- Chapter 4 gives an introduction to *formal reasoning* in TLA.

The deduction rules that are discussed in this chapter are not essential for understanding how the theory can be used in practice. In particular, the reader is warned of the fact that even 'obvious' properties may lead to long and complicated proofs, when carried out in detail, and that less formal proofs in English may be perfectly adequate in practice.

1.3.2 Part III: Building a Practical Theory

Part III of the book addresses questions on building a practical theory on the fundamentals described in Part II, and consists of four chapters.

Since practical use of the theory requires a specification and design language,

- Chapter 5 is devoted to *language aspects*, by which notions like types, finite-state structures, object-oriented classes, relations between objects, and multi-object actions can be built on top of the primitive execution model and can be rigorously reasoned about.

The language ideas presented in this chapter will be used in the rest of the book. The main purpose of the chapter is not, however, to give a detailed language, but to present the main problems in designing a language as part of the theory.

Of special importance in this chapter is how the facilities of object-oriented programming languages can be abstracted to a level that is appropriate for specifications. In particular, single-object 'methods' and communication protocols between objects are abstracted into *multi-object actions*, which allows reasoning on *collective behaviors* even in early stages of specification and design.

As presented in Fig. 1.1 (p. 8), design methods have a central role in our theory. To discuss them,

- Chapter 6 introduces the basic mechanisms to support *design methods*.

The main principle here is that specifications are constructed in incremental layers, and that the design can therefore proceed incrementally, with rigorous support for the preservation of behavioral properties in each step. The modularity of the design language has been designed to support such a layered structure, where a specification layer need not correspond to a natural module in an implementation, but may, instead, correspond to a concern that cuts across them in an *aspect-oriented* manner.

The methodology gives a theoretically solid foundation for a specification and design process that can start at a high level of abstraction and proceed by stepwise refinements towards an implementable form. It also supports the preservation of certain crucial properties (safety properties) in each refinement step, without a need to resort to explicit proofs.

To allow object-oriented specification in the full meaning of the word,

- Chapter 7 extends the discussion of language aspects to *aggregate objects* and to object-oriented *inheritance*.

This is done in such a manner that objects of a subclass always satisfy all properties specified for the base class. This also holds when multiple inheritance is used.

Modeling reactive systems as closed systems raises issues of *partitioning* a closed system into independently implementable components. To deal with such matters,

- Chapter 8 analyzes how *interfaces* can be defined in closed-system specifications, and under which conditions components in a closed-system specification can be refined independently.

A special characteristic of the design method is that interfaces between components can first be given at a high level of abstraction; a form that models their implementation can then be achieved by refinements.

Although partitioning of closed systems has both theoretical and practical interest, no language support is provided for it, and the reader may wish to skip this chapter during the first reading of the book.

1.3.3 Part IV: Distributed and Real-Time Systems

Part IV of the book provides excursions to two more specific areas, which may not interest all readers.

Historically, the development of this theory started with an attempt to model *distributed systems* at a high level of abstraction, and the notion of multi-object actions was originally proposed for this purpose. Addressing the specific problems of distribution,

- Chapter 9 analyzes the applicability of the interleaved execution model to distributed systems, and how action systems can be implemented in a distributed fashion.

In particular, it is shown in this chapter that the simple interleaved execution model does, indeed, also provide a suitable basis for the modeling of distributed systems, in which real concurrency is involved.

Since real time is essential for many reactive systems,

- Chapter 10 describes how the theory can be applied to model and reason about *real-time properties*.

The treatment in this chapter also covers *hybrid systems*, in which *continuous state functions* of the environment are also relevant.

Bibliographic Notes

Model-oriented abstractions have a long history in computing, and are primarily reflected in the evolution of programming languages.

The idea of rigorous reasoning on programs can be found even in some early papers by Goldstine and von Neumann [68] and by Turing [190]. Serious interest in this topic did not, however, arise before Floyd's seminal paper [57]. At the same time, similar ideas were presented independently in a less widely known paper by Naur [163].

The next step towards a theory of programs was Hoare's work on associating axioms and logical deduction rules directly with the definition of a programming language [83]. *Hoare logic* involved triples of the form $\{\mathsf{P}\}\mathsf{S}\{\mathsf{Q}\}$ with the following transformational meaning: if the execution of a program statement S starts in a state where *precondition* P holds, then its execution terminates in a state where *postcondition* Q holds.

Based on Hoare's work, Dijkstra observed [43, 44] that each program statement S can be given an abstract meaning as a *predicate transformer* Φ_{S}, which for any postcondition Q gives the *weakest precondition* P for which $\{\mathsf{P}\}\mathsf{S}\{\mathsf{Q}\}$ holds, i.e., $\Phi_{\mathsf{S}}(\mathsf{Q}) = \mathsf{P}$. This was a crucial step in making mathematical manipulation of programs possible with the transformational basic philosophy. The associated *refinement calculus* gave a solid foundation for rigorous programming methods, as developed further by Gries [71], Hehner [79], Morgan [161], and Back and von Wright [21], for instance.

As for the more established specification languages and formal methods that are essentially based on transformational semantics, the reader is referred to *VDM* [96], *Z* [184], and *B* [9]. For a more detailed history of formal reasoning on programs the reader is referred to an extensive survey by Jones [97].

In dealing with reactive systems, the *synchrony hypothesis* provides the basis for Harel's *statecharts* [76] and the associated tools called *STATEMATE* [77], as well as for a family of synchronous programming languages that includes *Esterel*, *Lustre*, and *Signal* [73].

As for theories with 'truly reactive' semantics, *event-based* approaches were pioneered by Hoare's *communicating sequential processes (CSP)* [86] and Milner's *calculus of communicating systems (CCS)* [159, 160]. *LOTOS* [26] is a

specification language that combines such a *process-algebraic* approach with an *algebraic specification* of data structures.

Event ordering in distributed systems, which is essential for *interleaving* models, was first discussed by Lamport [136]. An early example of such execution models is the one given by Lynch and Fischer [149]. *Partial-order* and interleaving semantics for CSP-like languages have been compared by Reisig [176].

Temporal logic was introduced to *state-based* reasoning on reactive systems by Pnueli [170, 171]. More recent textbooks by Manna and Pnueli [152, 153] give a comprehensive treatment of this. The term 'reactive system' was coined by Harel and Pnueli [78]. The inherently greater complexity of reactive systems, when compared to transformational systems, has been discussed by Wegner [193], for instance.

Temporal logic of actions (TLA) was developed by Lamport [138, 141]. One of its goals was to achieve the situation that an implementation relationship between specifications corresponds to logical implication. TLA^+ [144, 145] is a language for constructing TLA specifications. The approach in this book is based on experiences with an experimental specification language *DisCo* [93, 124, 49], in which TLA has been used as the logical basis.

While most research has concentrated on limited aspects of a theory, Chandy's and Misra's *UNITY* [36] was an important milestone in developing a comprehensive theory for distributed systems. Its essential components are an execution model and an associated language (the UNITY language), a temporal logic tuned to deal with this language (UNITY logic), and modularity constructs that support certain design methods.

Although the theory presented in this book has been developed independently of UNITY, and its goals are somewhat different, there is much similarity between the two approaches. The most important differences will be discussed at the end of those chapters where the different parts of the theory are addressed.