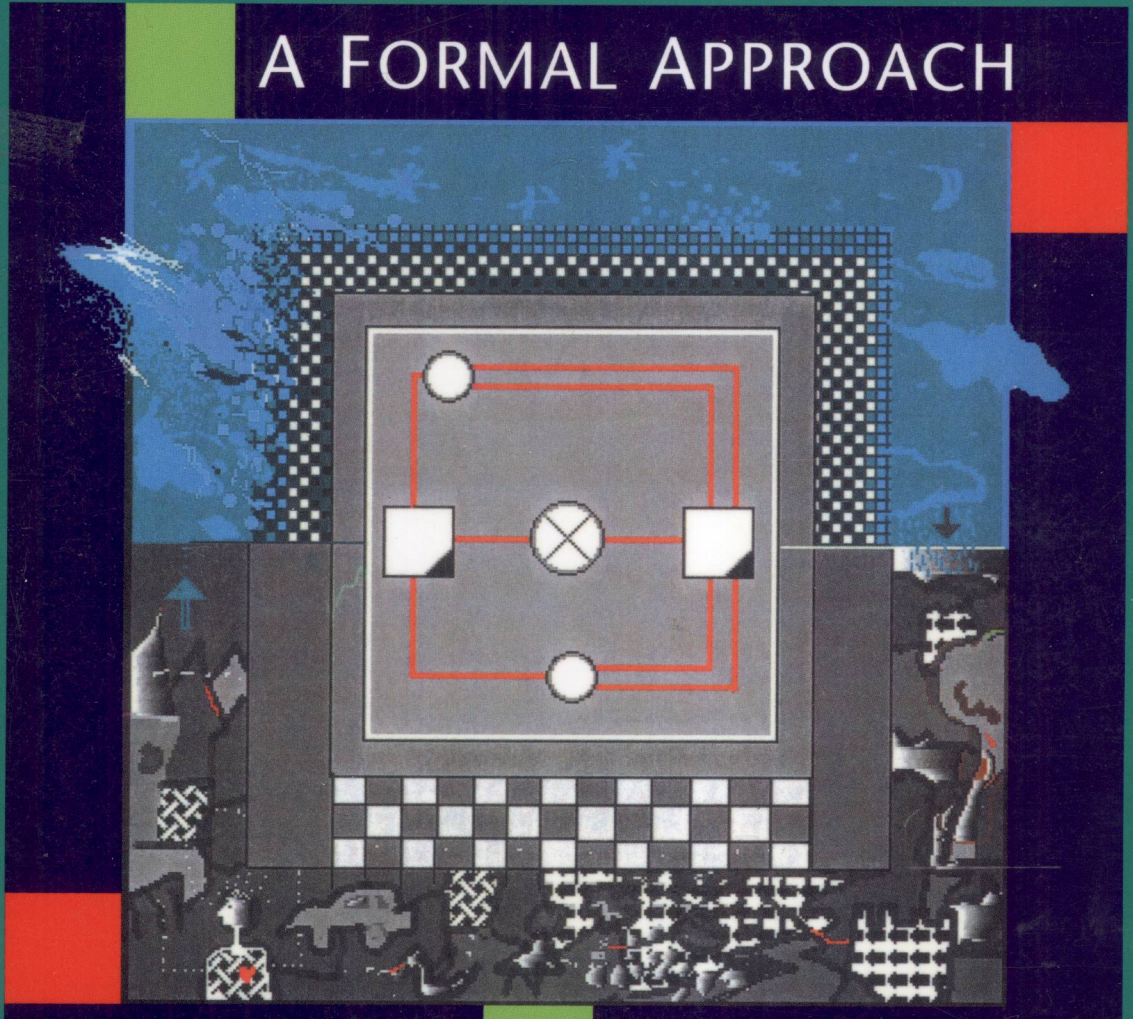
INFORMATION SYSTEMS ENGINEERING

A FORMAL APPROACH



K. M. VAN HEE

INFORMATION SYSTEMS ENGINEERING: A FORMAL APPROACH

K. M. van HEE

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore,
São Paulo, Delhi, Dubai, Tokyo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521110648

First published 1994
This digitally printed version (with corrections) 2009

A catalogue record for this publication is available from the British Library

ISBN 978-0-521-45514-5 Hardback
ISBN 978-0-521-11064-8 Paperback

The illustration for the cover was supplied by Annelies Schott

INFORMATION SYSTEMS ENGINEERING: A FORMAL APPROACH

Preface

The term *systems engineering* covers a wide area of activities, focused on the development of systems. One of these activities, called *systems modeling*, is the construction of *models* of systems. Models are made and used by systems engineers for various reasons. Sometimes a model is used to document the *functionality* of a system, i.e. the model describes the behavior of a system in an abstract way. For instance this is done in *requirements engineering*, in case it concerns a new system, or in *reverse engineering*, if an old one has to be renovated. In both cases the model is used as a *specification* of the system.

A model can also be used as a *blueprint* of a system that has to be constructed. In this case the description is less abstract than a specification in the sense that it should be easy to map the building blocks of the model onto existing or realizable components.

A third reason to make a model is to *analyze* a system, for instance *performance* or *reliability*. If the system already exists one could in principle observe its behavior. However, it is often not feasible to experiment with a real system, due to the costs or the risks of experimentation. If the system under consideration is new, i.e. it exists only on paper and in the minds of some persons, a model is the only way to study its behavior. Instead of experimenting with the real system the systems engineer may perform experiments with a model.

In many cases it is possible to analyze the models in a formal way, instead of analysis based on experiments. This has the advantage that we are able to prove theorems for the behavior of a system, while experiments allow us only statistical assertions to make and may serve as counter-examples for hypotheses.

The models we consider here are *abstract* or *mathematical* rather than physical. Often we transform these models into a form appropriate for interpretation by a computer. In that case the computer can *simulate* the behavior of the system and experimentation is then called *computer simulation*.

Models are expressed with *formalisms*. A formalism consists of a mathematical *framework* and a *language*. A model is an *instance* of a framework. A model is described in the language, which might be textual or graphical. A description of a model in the language is called a *script*. So the *semantics* of a script is a model.

The formalisms we use apply to all systems that have a countable set of *states* and that make *state transitions* at discrete moments. We call these systems: *transition systems*. A very important type of system is called *information system*, in which the state is formed by a set of *information objects*. The scope of this book is however wider than computer science, and includes systems studied in industrial engineering and electrical engineering as well.

Most books on the development of information systems are focused on the techniques used in practice today. The formalisms they use to make models of systems are very weak. Often *data flow diagrams* and *entity-relationship diagrams* are the only "formalisms" offered. On the other hand these books pay a lot of attention to interview techniques, cost-benefit analysis and planning techniques. In this book we do not consider these topics at all, not because they are unimportant for the systems engineer but because there are already enough books on them.

In this book we concentrate on modeling of systems and therefore we need some formalisms to describe the models. Many formalisms consider only *aspects* of transition systems; for instance *data models* to define *state spaces* of systems, *process models* to define the *interaction* between different components of systems and so-called *specification languages* to define *local state transitions* or *operations*. (Note that data models and process models are in fact formalisms for *defining* models rather than models themselves.) It turns out to be very difficult to integrate these different *views* of a system. Therefore we provide a small set of formalisms here and we show how they may be combined to obtain an *integrated model* of a system. We do not provide a detailed survey of all existing formalisms here, but select a few and extend them a little in order to be able to *integrate* them.

The frameworks we will use are:

- transition systems,
- a binary data model with complex objects,
- timed and colored Petri nets.

The languages we use are:

- a specification language very close to Z,
- graphical languages for defining data models and Petri nets.

The *transition systems* (in fact unlabeled transition systems) provide the highest

level of semantics: every model denotes a transition system. A transition system has an *event set* and a function, called *transition law*, that assigns to a (finite) sequence of events a set of possible subsequent events. An event is a pair consisting of a *state* and a *time point*, which denotes the time the system moved to the state.

The *data model* is used to define state spaces of systems. The data model is a binary version of the entity-relationship model with two extensions: there are more facilities for expressing *constraints* and there is a notion of *aggregation* of entities and relationships into so-called *complex objects*. In fact a state is a set of complex objects.

The *transition law* of a system is defined by (timed and colored) Petri nets. In a Petri net we have *places* in which complex objects may reside and *processors* that may *consume* and *produce* complex objects from specific places. (In the Petri net literature processors are called *transitions* and the complex objects are called *tokens*.) Each processor has an *input-output relation*, which determines its consumption/production behavior. The time component of the formalism enables us to model *real-time* aspects of systems.

The *specification language* is a language based on *typed set theory* with a *constructive subset*, that is, a *typed functional language*, which is very close to Z. In fact it is a subset of Z, because we have restricted ourselves to a *countable universe of values* that can be represented by *finite sets*. Functions map values to values. Functions are in general infinite sets, so they are not considered to be values themselves, which implies that we do not allow function-valued functions. These restrictions do not impede the modeling of systems in practice, and make the theoretical treatment more easy; we present a construction for the language, including static type checking and evaluation of expressions. The constructive subset of the specification language is important because we advocate a style of modeling that is adopted from VDM, in which the systems engineer first gives a *descriptive specification* of an entity and afterwards a *construction* in a constructive subset of the language. So, we distinguish descriptive and constructive specifications. (Often it is possible to give a construction that is easy to understand at once, so that a descriptive specification is no longer useful.) Our subset of Z has very few primitive functions, which means that most functions used in specifications, are themselves constructed in the language. Of course the language is extendible in the sense that new primitive types and functions can be added. Our syntax differs from Z on minor points, mainly where we use λ-expressions. In Z the λ symbol is used for function definitions. However, in practice that is not always a convenient notation. (Most functional languages use more familiar ways for expressing function constructions.)

The specification language is used to define the types of the complex objects that *flow* through the Petri net and the input-output relations of the processors in the Petri net. An important language construct is a *schema.* (A schema denotes a subset of a labeled Cartesian product.) Schemas are defined using types, functions and predicates. The complex objects and the input-output relations are defined by schemas. The language also has schema operators to define complex schemas using already defined simpler schemas. Another difference between our specification language and Z is that we allow *partial schemas*, i.e. schemas in which not all *variables* or *attributes* must have a value. This is particularly important for the specification of processors that do not consume from all their input places and that do not produce output for all their output places.

Constructive specifications are useful because they are *executable*, which is important when a systems engineer needs to *validate* a model by means of simulation experiments. For information systems a computer model is a *prototype* that can be tested by potential users of the modeled system.

Graphical languages are used to define the structure of the Petri nets and parts of the data model. In these languages descriptions are in fact *graphs*, in which the vertices represent object classes, processors or places and in which the edges represent relationships between object classes or connections between processors and places. The graphical language for the data modeling is close to the usual ones. In the Petri net language we have introduced *hierarchy*, which makes it possible to *decompose* models in a way similar to *data flow diagrams*. This is a useful feature because data flow diagrams are very often used in practice and systems engineers who have experience with data flow diagrams, can use them in our formalism in almost the same way.

Our choice of formalisms is rather arbitrary. A combination of a process algebra, an algebraic specification language and another data model could have worked also. However, we have the experience that so-called *model-based* formalisms, such as we have chosen, are more comfortable with practitioners than *property-based* formalisms, such as the algebraic ones.

The author is leader of a research group at the Eindhoven University of Technology that has developed a software tool called ExSpect. This tool incorporates most of the formalisms introduced in this book and is commercial available by Bakkenist Management Consultants, Amsterdam. The tool has been operational since 1989. It is used by several industrial companies for a variety of applications, such as the modeling and analysis of logistical systems and the prototyping of distributed software systems. (It is also used by software houses in the ESPRIT project EP-5342 called PROOFS.) These experiences have motivated my decision to write this book. The chosen combination of formalisms is useful. It is

not necessary to use ExSpect to apply the theory of this book in practice. There are other tools on the market that support our approach as well. The use of a tool is recommended for large systems or if simulation or prototyping is needed. Each tool has its own peculiarities and requires some learning time. Note that our approach is not dependent on any tool however, there are tools that support it.

The book is divided into five parts. The first, called Concepts, introduces most of the concepts a systems engineer needs for modeling systems. The treatment is as informal as possible in order to give the reader an intuitive understanding of these concepts, which are illustrated with realistic examples.

In the second part, called Frameworks, the formalization of these concepts is given as well as the more theoretical details of the frameworks used. This part is interesting for experts; the frameworks are complex and it may take some time to understand them fully. If we expect systems engineers to make formal specifications then they have to understand the formalisms they use! However it is possible to understand many details of later parts of the book without knowledge of this part.

In the third part, called Modeling methods, we give a collection of methods for constructing models. These methods, all illustrated with practical examples, are *actor modeling*, concerning the modeling of (extended) Petri nets and *object modeling*, which is also called *data modeling*. We conclude this part with a chapter on *object oriented modeling*, which is a mixture of actor and object modeling. The modeling methods cover ones for constructing models after reality as well as others for transforming models from one formalism into another. The modeling methods are treated mostly in an informal way, using examples, in order to facilitate the readability.

Actor models and object models are "glued together" with the specification of types for complex objects and processor relations. Here we need the specification language that is introduced in part I and treated in detail in part V.

In part IV, called Analysis methods, we consider several methods for analyzing a model. They cover *invariant methods* and *occurrence graph methods* of Petri nets, methods for verifying *time constraints* and *simulation methods* for validating models by experimentation.

In part V, called Specification language, we define the Z-like specification language. The construction of the language is interesting in itself. Because function construction plays an important role in specifications, this topic is treated extensively. Therefore this part of the book can also be considered as an introduction to *functional programming*.

Each part concludes with an annotated bibliography and a set of exercises. Answers to the exercises are available from the author.

The book ends with three appendices on mathematical notions, the syntax of the specification language and a toolkit of useful functions.

The book is intended for advanced undergraduate and graduate students in computer science, electrical engineering and, industrial engineering as well as for professional systems engineers. The author is convinced that tommorow's systems engineers need an education in formal methods to model systems than is offered in most university courses to day. The only way to cope with the complexity of large systems is to make precise and concise models (of parts) of the systems under consideration. This book offers the foundations for such education.

Prerequisites of the book are: a good understanding of (naive) set theory and predicate calculus. Some experience in functional programming and some knowledge of Petri nets and data bases is useful but not necessary. For a more practical course the instructor could summarize part II and skip part V. Since there are three views presented in the book (the data base view, the Petri net view and the formal specifications view) instructors may decide to emphasize one of these perspectives in a course.

The whole book can be taught in about sixty hours. A course that covers only parts I, II and III can be taught in about forty hours. Exercises with a tool like ExSpect are very useful.

The book will also be of interest for researchers in the areas of formal specifications and systems modeling, in particular the methods part contains many challenges for future research.

Acknowledgements

The main part of the book is written while I was on a sabbatical leave at the University of Waterloo, Ontario. I thank John Ophel and Farhad Mavaddat from the Computer Science Department of this University and the students of course CS757 (1992) for many useful comments. Furthermore I wish to express my gratitude to Jan Paredaens (University of Antwerp) for several helpful comments.

The book is the result of the research and software development of the ExSpect group at the Eindhoven University of Technology. I thank them all for their contributions, but in particular Wil van der Aalst, Lou Somers, Peter Verkoulen and Marc Voorhoeve.

The research of the ExSpect group was supported by the ESPRIT program (EP-5342) and the TASTE project of the Netherlands organization for applied scientific research, TNO. Therefore I am grateful to the European Commission

and TNO as well as the Eindhoven University of Technology for granting me a sabbatical leave.

Last but not least I thank Jane Pullin and Jeroen Schuyt for a large part of the text editing, and Susan Parkinson for excellent copy editing.

Kees M. van Hee,
Eindhoven University of Technology,
e-mail: wsinhee@win.tue.nl
1993

Contents

Part I

System concepts

1
Introduction

Engineering is the scientific discipline focussed on the creation of new artifacts designed to be of some use to our society. Such artifacts range from buildings to software and from screws to airplanes. Different types of artifacts require different engineering approaches; therefore there exist many engineering disciplines. However, in all these disciplines the *development* of a new artifact is divided into *stages*. Three stages can always be recognized in some form, although the terminology may differ between disciplines or even between schools within the same discipline. These stages are as follows.

Analysis, which involves:

- *evaluation* of the existing artifact (if any) that is to be replaced or improved and of the *environment* in which the artifact should fulfill its tasks,
- *specification* of the *requirements* the artifact has to fulfill for its environment.

Design, which involves the creation of two models:

- the *functional model*, also called the *specification*, which describes the behavior of the artifact in an abstract way;
- the *construction model*, also called the *blueprint*, which models the artifact in terms of existing components and materials from which the artifact is to be constructed.

The design stage also involves *verification* that the functional model satisfies the requirements specified in the analysis and that the construction model has the functionality described in the functional model.

Realization, which involves:

- *construction* of the artifact (or a prototype) according to the construction model,

– *testing* the constructed artifact with respect to the functional model.

These three stages concern the development of an artifact; however the *life cycle* of an artifact involves further stages. If it is a *mass product*, there are two additional stages.

Production, in which copies of the realized artifact are produced (note that often the production process itself has to be designed first).

Distribution, in which the copies are brought to the environment where they are wanted, often via market mechanisms.

These stages do not occur if the artifact is a unique product. Whether or not the latter is the case, the following stages belong to the life cycle.

Introduction of (a copy of) the artifact in the environment where it will be used.

Maintenance of the artifact to keep it working or to adapt it to new requirements.

In the present book we only consider the first two stages of the development process. We focus our attention on a specific type of artifact, called a *discrete dynamic system*. Such a system consists of active components or *actors* that *consume* and *produce* passive components or *tokens*. Many complex artifacts in our world can be considered as discrete dynamic systems. Three subtypes will be studied in more detail:

– business systems (such as a factory or restaurant);
– information systems (whether automated or not);
– automated systems (systems that are controlled by an automated information system).

The first subtype is studied by *industrial engineers*, the third by *software engineers* and *electrical engineers*, whereas the second is a battlefield for all three disciplines. We hope that our approach suits these disciplines; we call their union *systems engineering*. The types of discrete dynamic systems on which we focus our attention are described in the next chapter.

During the analysis and design stages of the development process for discrete dynamic systems the systems engineer is working with *models* of these systems. A model is in fact another system, but one that is easier to analyze or observe than the original system. But the model has so much similarity to the original system (in certain aspects at least), that conclusions drawn from the model are assumed to be valid for the original system as well.