Formal Methods for Software-based Systems Engineering

Dines Bjørner, Professor Emeritus

Technical University of Denmark

Forum Academique AFIS'2007

Programme du pré-Forum Academique
Rencontre "Université – Industrie"

Abstract

- We analyse the two composite terms of the title of this talk:
  - ⋆ Formal Software Development Methods **(FM)**
  - ⋆ Systems Engineering **(SE)**
    (**SE** does here not stand for Software Engineering)
- Then we look at their composition:
  - ⋆ What does it mean to do
    Formal Methods Software-based Systems Engineering ?
    (FMS^{2}E)
  - ⋆ Why would one want to do **FMS^{2}E** ?
  - ⋆ Who, how and where should engineers for **FMS^{2}E** be educated ?
  - ⋆ Can we today do **FMS^{2}E** ?
- Finally we conclude.

Formal Software Development Methods (FM)
What Is a Method ?

- A method
  - ⋆ is a set of **principles**
  - ⋆ for **selecting** and **applying**
  - ⋆ a number of **techniques** and **tools**
  - ⋆ in order to construct an artifact.
- Some methods are better than other methods:
  - ⋆ lead more effectively to the final product,
  - ⋆ and/or lead to trustworthy, believable products.
- Formal (software development) methods are claimed — and have, in many cases, led to
  - ⋆ shorter, lower cost production times,
  - ⋆ of products that are safe and reliable, correct and usable.

What Is a Formal Software Development Method ?

- A formal software development method is one which
  - ⋆ offers **techniques** and **tools**
    - ◇ for the **specification** of software requirements and abstract designs and concrete code
    - ◇ and for the **proof of correctness** of formally specified designs with respect to formally specified requirements;
  - ⋆ **tools** (like specification languages) that have
    - ◇ a **formal syntax,**
    - ◇ a **formal semantics,**
    - ◇ a **formal proof system** and
    - ◇ software to support specification construction and proofs.
  - ⋆ **techniques** like
    - ◇ specification **refinement** and **proof** techniques.

What Is a Formal Syntax ?

• What Is a Syntax ? •

- A **syntax** is a set of **rules** for how to form sentences from **ground terms** (characters, keywords, literals, mathematical and other symbols).
  - ⋆ A syntax defines what a syntactically correct sentence is; thus we can use syntax
    - ◇ **generatively**, to generate sentences, and
    - ◇ **analytically**, to analyse sequences of ground terms.

• What Is a Formal Syntax ? •

- A formal syntax is a syntax expressed, basically in a mathematical notation that can be given a precise meaning.

What Is a Formal Semantics ?

- A **formal semantics** of a specification language is a mathematical definition
  - ⋆ which to every proper,
  - ⋆ i.e., syntactically well-formed specification,
  - ⋆ typically, ascribes a set of mathematical values.
  - ⋆ Any element of this set
  - ⋆ is a model of the specification.

What Is a Proof System ?

- A **proof system** for a specification language
  - ⋆ is a set of axiom schemes,
  - ⋆ a set of rules of inference,
  - ⋆ and a set of theorems derivable from these,
  - ⋆ such that proofs of properties
  - ⋆ claimed (in some predicates) of a specification
  - ⋆ can be made.

- A specification language formal semantics and proof system should be related:
  - ⋆ Specification models must be interpretations of the proof system.

What Does it Mean to Do Formal Software Development ?

- There are, to paraphrase, two approaches to formal development of software:
  - ⋆ In one, the oldest (since late 1960s) approach
    - ◇ one first develops an algorithm for some software
    - ◇ and then one proves it correct with respect to some assertions.

    We shall call this **The Assertion Method.**
  - ⋆ In the other, the more modern (since early 1970s) approach
    - ◇ one first develops a formal specification of the algorithm (etc.)
    - ◇ and then one "derives" — refines — the algorithm (etc.) from the specification.

    We shall call this **The Refinement Method.**

The Assertion Approach

- An **assertion**
  - ⋆ is a **predicate**
  - ⋆ (i.e., a **true/false** statement)
  - ⋆ **placed in a program**
  - ⋆ to indicate that the developer
  - ⋆ thinks that the **predicate is always true** at that place.

$$\frac{}{\{P\}\ \mathbf{skip}\ \{P\}}$$

$$\frac{}{\{P[x/E]\}\ x := E\ \{P\}}$$

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

$$\frac{\{B \wedge P\}\ S\ \{Q\}\ ,\ \{\neg B \wedge P\}\ T\ \{Q\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{endif}\ \{Q\}}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{done}\ \{\neg B \wedge P\}}$$

The Refinement Approach

- In the refinement approach
  - ⋆ an abstract model
  - ⋆ is refined in perhaps several steps
  - ⋆ into a concrete model, i.e., the code.

On Refinement Calculi

- Refinement calculus is a formalized approach to stepwise refinement for program construction.
- The required behaviour of the final executable program is specified as an abstract and perhaps non-executable "program",
- which is then refined by a series of correctness-preserving transformations
- into an efficiently executable program.

Are There Several Formal Software Development Method ? Yes !

- There are several **formal specification languages** several with own **proof** and **model checking** tools:

  - ⋆ **Petri Nets** (1963) Concurrency [40, 39, 41]
  - ⋆ **VDM-SL** (1974) State Systems [12, 18, 17]
  - ⋆ **CSP** (1978) Concurrency [29, 44, 45]
  - ⋆ **Z** (1980) State Systems [46, 27, 28]
  - ⋆ **Statecharts** (1987) Concurrency [24, 25]
  - ⋆ **RAISE, RSL** (1989) State Systems, Concurrency [20, 22, 21, 19]
  - ⋆ **DC** (1990) Temporal Logic [47, 23]
  - ⋆ **MSCs, LSCs** (1992, 2001) Timing [30, 26]
  - ⋆ **TLA+** (1994) Temporal Logic: $\boxed{\text{Nancy Center}}$ [32, 35, 36]
  - ⋆ **B, Event-B** (1996, 2005) State Systems: $\boxed{\text{Nancy Center}}$ [1, 2]
  - ⋆ **Alloy** (1997) State Systems [31]

- and there are several additional **tools:**

  Theorem Proving

  - ⋄ **NqThm, ACL2** (1971, 1995) [48]
  - ⋄ **Isabelle/HOL** (.../1987) [49]
  - ⋄ **PVS** (1992) [50]
  - ⋄ **STeP** (1997) [33, 34, 51]

  Model Checking

  - ⋄ **SPIN** (1991) [52]
  - ⋄ **SMV** (1994) [53]

The fields are expanding !

The Software Engineering Triptych
The Triptych Dogma

- Before software
  - ⋆ (in general: the machines, i.e., systems of computers and communication and of sensors and actuators etcetera connected to them)
  - ⋆ can be designed
  - ⋆ we must understand "the" requirements.
- Before requirements,
  - ⋆ that is, prescriptions for the machine,
  - ⋆ what it should do, not how,
  - ⋆ can be prescribed
  - ⋆ we must understand the domain.

The Triptych Doctrine Consequences

- In consequence we prefer to develop software professionally, that is:
  - ⋆ First we study an available — or develop ourselves an as "complete" as possible —
    - ◇ **domain description**;
  - ⋆ then we develop, from such a domain description, the
    - ◇ **requirements prescription**;

    and
  - ⋆ from the requirements prescription we carry out the
    - ◇ **software design**.

Narrative versus Formal Specifications
Three Forms of Specification

- By a specification we shall here (a bit narrowly) mean
  - ⋆ a narrated and a formal **description** of a **domain**,
  - ⋆ a narrated and a formal **prescription** of a (set of) **requirements**, or
  - ⋆ a narrated and a formal **design** (document[ation]) of some **software**.
- So the term 'specification' has three instantiations:
  - ⋆ **description**,
  - ⋆ **prescription** and
  - ⋆ **design** (document[ation]).

Interlude

- We have surveyed answers to:
  - ⋆ What is a Method ?
  - ⋆ What is a Formal Software Development Method ?
    - ◇ What is Syntax ?
    - ◇ What is Semantics ?
    - ◇ What is a Proof System ?
  - ⋆ What Do We Mean By a Formal Software Development Method ?
    - ◇ What is the Assertion Approach ?
    - ◇ What is the Refinement Approach ?
  - ⋆ Are There Several Formal Software Development Methods ?
  - ⋆ What is the Triptych Approach ?
    Domains, Requirements, Design; Narratives, Formalisations

- Now we can turn to the other compound term in the title of this talk:
  - ⋆ Formal Methods for Software-based Systems Engineering

Systems Engineering (SE)

- First we analyse the term: **System**
  - ⋆ with respect to software for such systems;
- then we analyse the term: **Engineering**
  - ⋆ with respect to how software engineers develop such software.

What is a System ?

- We shall make the distinction between
  - ⋆ Human systems, possibly with IT, and
  - ⋆ IT, that is: computer and communication systems, possibly without humans,
    - ◇ but with hardware
    - ◇ and software.
- Software-based systems are IT systems,
  - ⋆ that are to be developed,
  - ⋆ inserted in existing human systems,
  - ⋆ and include
    - ◇ the right software and
    - ◇ software that is right.
- Therefore we are concerned about
  - ⋆ 'Formal Methods for Software-based Systems Engineering'

Human Systems
A Characterisation

- By a human system we shall, in this talk, mean
  - ⋆ a collection of people,
  - ⋆ a collection of resources,
  - ⋆ interacting with one another:
    - ◇ carrying out tasks
    - ◇ in single actions
    - ◇ subject to external events
  - ⋆ exhibiting various behaviours,
  - ⋆ subject to rules & regulations and
  - ⋆ achieving or not achieving goals.

Examples of Human Systems

- ⋆ airports,
- ⋆ air traffic,
- ⋆ banking,
- ⋆ consumers/retailers/wholesaler,
- ⋆ distribution chains,
- ⋆ insurance,
- ⋆ manufacturing,
- ⋆ stock brokerage and exchange,
- ⋆ railways,
- ⋆ etcetera.

Description of Human (etc.) Systems
Domain Description

- Before we can establish requirements
  - ⋆ for an IT system which
  - ⋆ should support activities
  - ⋆ in the human system
- we must first understand it:
  - ⋆ tell the story, **informally** (the narrative), but concisely, and
  - ⋆ **formally**,
- all **entities, functions, events** and **behaviours**.
- So first we do **domain engineering**.
  - ⋆ We do so in order to achieve the **right software**.

Domain Description

- To describe the domain, as it is, is to describe the domain
  - ⋆ first rough sketch the "business" processes,
  - ⋆ then
    - ◇ intrinsics,
    - ◇ support technologies,
    - ◇ management & organisation,
    - ◇ rules & regulations,
    - ◇ scripts and
    - ◇ human behaviour —
- as much as possible,
- much more than is thought needed for the requirements.

Requirements for IT for Human (etc.) Systems
Different Requirements Parts

- The requirements is for a **machine:**
  - ⋆ **hardware** and
  - ⋆ **software.**
- The requirements prescription consists of
  - ⋆ **domain,**
  - ⋆ **interface** and
  - ⋆ **machine** requirements
- These requirements are those which can be expressed
  - ⋆ (for **domain** reqs.:) sôlely using terms from the domain,
  - ⋆ (for **interface** reqs.:) using terms from both the domain and the machine, resp.
  - ⋆ (for **machine** reqs.:) sôlely using terms from the machine.

Requirements for IT for Human (etc.) Systems (Continued)
How To Develop Domain Requirements

- **Domain requirements** are "derived" from the domain description:
  - ⋆ by **projection,**
  - ⋆ by **instantiation,**
  - ⋆ by **determination,**
  - ⋆ by **extension,** and
  - ⋆ by **fitting**

  of the domain description and, for fitting, with other requirements.
- These domain-to-requirements **refinements** are done
  - ⋆ together with the requirements stakeholders
  - ⋆ by "interpreting" the domain description **line-by-line**.

How To Develop Domain Requirements (Continued)

- ⋆ By carefully relating (**validating, verifying, model checking**) and documenting
  - ◇ domain requirements,
  - ◇ line-by-line,
  - ◇ to the domain description
- ⋆ and by both
  - ◇ **narrating** and
  - ◇ **formalising**

  the domain requirements prescriptions
- ⋆ we can help guarantee that the requirements
  - ◇ lead to **the right software**
  - ◇ and that **the software is right.**

How To Develop Interface Requirements

- **Interface requirements** are "derived" from the domain description:
  - ⋆ identifying all **shared**
    - ◇ **entities,**
    - ◇ **functions,**
    - ◇ **events** and
    - ◇ **behaviours**
  - ⋆ and then prescribing what is to be shared:
    - ◇ **data** (**entities**)
      - ◦ initialisation,
      - ◦ refresment and
      - ◦ display,
    - ◇ short term **interactive computation** (**functions**),
    - ◇ **event handling** (**events**) and
    - ◇ long term **man/machine interaction** (**behaviours**).

How To Develop Interface Requirements (Continued)

- ⋆ By carefully relating (**validating, verifying, model checking**) and documenting
  - ⋄ interface requirements,
  - ⋄ line-by-line,
  - ⋄ to both
    - ∘ the domain description and
    - ∘ specifications of machine facilities
- ⋆ and by both
  - ⋄ **narrating** and
  - ⋄ **formalising**

  the interface requirements prescriptions
- ⋆ we can help guarantee that the requirements
  - ⋄ lead to **the right software**
  - ⋄ and that **the software is right.**

How To Develop the Machine Requirements

- **Machine requirements** cover
  - ⋆ **Performance**
    - ◇ storage,
    - ◇ time and
    - ◇ other resources.
  - ⋆ **Dependability**
    - ◇ availability,
    - ◇ accessability,
    - ◇ reliability,
    - ◇ security,
    - ◇ etc.
  - ⋆ **Maintainability**
    - ◇ adaptive,
    - ◇ corrective,
    - ◇ perfective and
    - ◇ preventive.
  - ⋆ **Platform**
    - ◇ development,
    - ◇ testing,
    - ◇ execution,
    - ◇ maintenance and
    - ◇ demonstration.
  - ⋆ **Documentation**
  - ⋆ **Etcetera**

How To Develop the Machine Requirements (Continued)

- By carefully relating (**validating, verifying, model checking**) and documenting
  - ⋆ machine requirements
  - ⋆ to specifications of machine (hardware and software) facilities
- and by both
  - ⋆ **narrating** and
  - ⋆ **formalising**

  the machine requirements prescriptions
- we can help guarantee that the requirements
  - ⋆ lead to **the right software**
  - ⋆ and that **the software is right.**

Software for Human (etc.) Systems
Design: Refinements, Implementations, Transformations

- $\mathcal{S}$oftware is now **designed**, in stages and steps, as were the $\mathcal{D}$omain description and $\mathcal{R}$equirements prescriptions.
  - ⋆ From **higher level** (system) abstract design, $\mathcal{S}_{\mathcal{A}}$,
  - ⋆ via **intermediate level** of increasing less abstract, more concrete designs, $\mathcal{S}_{\mathcal{I}_i}$, to **final** code, $\mathcal{S}_{\mathcal{C}}$.
- A stage of development is one in which an entire specification is subject to many steps of development.
- A step of development is one in which different parts of a design is subject to
  - ⋆ refinements (hand-made transformations),
  - ⋆ implementations (posit and assertion proved), and/or
  - ⋆ transformations ("automatic" transformations).

Software for Human (etc.) Systems (Continued)
Verification, Model-checking and Formal Testing

- The abstract design is, $\mathcal{S}_{\mathcal{A}}$, **proven**, **model-checked** and **formally tested**
  - ⋆ to show that: $\mathcal{D}, \mathcal{S}_{\mathcal{A}} \models \mathcal{R}$
  - ⋆ that is, that the abstract design is correct wrt. $\mathcal{R}$equirements and in the context of the $\mathcal{D}$omain.
- At each level
  - ⋆ we can **prove, model-check** and **formally test** the designs and relations between stages of design:

    $\mathcal{S}_{\mathcal{A}} \rightarrow \mathcal{S}_{\mathcal{I}_1}, ..., \mathcal{S}_{\mathcal{I}_i} \rightarrow \mathcal{S}_{\mathcal{I}_{i+1}}, ..., \mathcal{S}_{\mathcal{I}_n} \rightarrow \mathcal{S}_{\mathcal{C}}$
- We do this to help guarantee that the design
  - ⋆ lead to **the right software**
  - ⋆ and that **the software is right.**

What is Engineering. ?
From Science to Technology — and Back !

- The engineer **walks the bridge** between science and technology
  - ⋆ to construct artifacts based on scientific insight and
  - ⋆ to analyse technology for scientific properties.

- The software engineer **walks the bridge** between computing science and information technology
  - ⋆ to construct software based on computing science
  - ⋆ and to verify, model-check and formally test that software.

From Ideals to Reality

- An extreme interpretation of the Triptych paradigm is ideal:
  - ⋆ first extensive, generic and wide-coverage domain engineering,
  - ⋆ then specific requirements engineering,
  - ⋆ finally software design —
  - ⋆ all this with verification, model-cheking and formal testing.
- It may very well not be feasible.
- The engineers are the persons who make approximations to the ideal. Who decides
  - ⋆ how much of a domain to describe,
  - ⋆ how to follow the domain-to-requirements transformations,
  - ⋆ adherence to refinement, implementation and formal testing,
  - ⋆ which tools to use, and many related matters.

What is Systems Engineering. ?

- What distinguishes systems engineering from software engineering ?
  - ⋆ The software engineer, strictly speaking, is concerned "only" about software development, from domains via requirements.
  - ⋆ The systems engineer, broadly speaking, is concerned about both the hardware and the software systems development:
    - ◇ its integration into the domain,
    - ◇ business process re-engineering, with all that entails:
      - ○ new intrinsics, new support technologies, new mgt. & org.,
      - ○ new rules & regs., new scripts, changed human behaviours,
      - ○ new sensors, actuators and IT equipment,
    - ◇ etc.
  - ⋆ But the **professional systems engineer** uses **formal techniques**.

Conclusion: Formal Methods for Software-based Systems Engineering

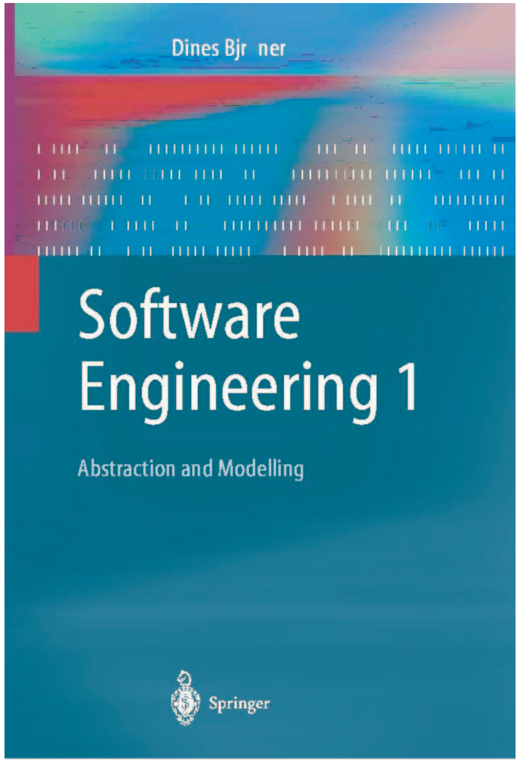
- We have answered the questions implied in the title of this talk:
  - ⋆ What is a Method ?
  - ⋆ What is a Formal Method ?
  - ⋆ What is a Software Development ?
  - ⋆ What is a Formal Software Development ?
  - ⋆ What is are Systems ?
  - ⋆ What is Engineering ?
  - ⋆ What is Systems Engineering ?
  - ⋆ And: Why Formal Methods for Software-based Systems Engineering ?

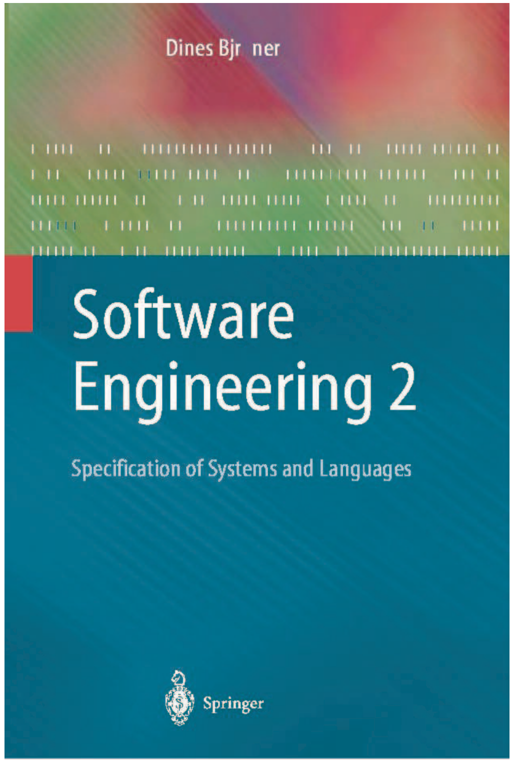
Conclusion (Continued)

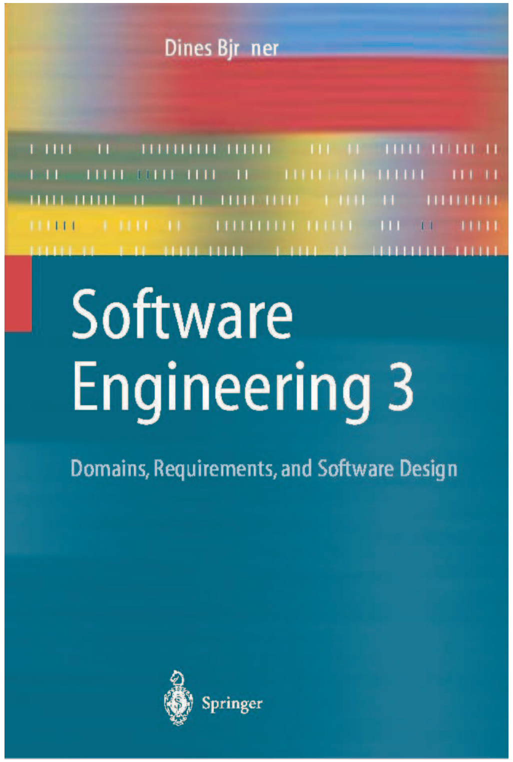
- Of course, the answers have been mere indications.
- It is now up to those industries who are not following the advice to do so:
  - ⋆ by hiring MSc and PhD candidates who know how,
  - ⋆ to integrate them into perfor,ing development teams,
  - ⋆ and to offer the right systems — that are right !
- It is great fun !
  - ⋆ Yoy can sleep at night.
  - ⋆ Your industry can say: overtime is a failure of management.
  - ⋆ You can deliver on time, at cost estimate.
  - ⋆ Your staff is continuously being reeducated through own work.

Any Questions ?

Please Buy My Book !







[3, 4, 5]

Bibliography

In [9, to appear] I give a concise overview of domain engineering; in [8, to appear] one of domain and requirements engineering as they relate; and in [7, to appear] I relate domain enginering, requirements engineering and software design to software management. In [6] I present a number of domain engineering research challenges. In [10, to appear] — which also covers research challenges of domain engineering — I additionally present a rather large example of the container line industry domain.

Bibliography

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1996.

[2] C. Métayer, J.-R. Abrial and L. Voisin Event-B Language, ETH Zürich, Switzerland and ClearSy, Marseille, France, 31st May 2005, IST-511599, EU Information Society Technologies; RODIN Deliverable 3.2

[3] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[4] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[5] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006.

[6] Dines Bjørner. Domain Theory: Practice and Theories, Discussion of Possible Research Topics. In *ICTAC'2007*, volume 4701 of *Lecture Notes in Computer Science (eds. J.C.P. Woodcock et al.)*, pages 1–17, Heidelberg, September 2007. Springer.

[7] Dines Bjørner. Believable Software Management. *Encyclopedia of Software Engineering*, 1(1):1–32, 2008. Taylor & Francis, New York and London.

[8] Dines Bjørner. Deriving Requirements from Domains. In *Festschrift for Ugo Montanari*, to appear, *Lecture Notes in Computer Science (eds. Nico de Rocola et al.)*, page 30, Heidelberg, May 2008. Springer.

[9] Dines Bjørner. Domain Engineering. In *BCS FACS Seminars*, Lecture Notes in Computer Science, the BCS FAC Series (eds. Paul Boca and Jonathan Bowen), pages 1–42, London, UK, 2008. Springer. To appear.

[10] Dines Bjørner. Domain Engineering. In *The 2007 Lipari PhD Summer School*, *Lecture Notes in Computer Science (eds. E. Börger and A. Ferro)*, pages 1–102, Heidelberg, Germany, 2008. Springer. To appear. To appear.

[11] Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages — see [43, 15, 23, 19, 36, 17, 28]*. EATCS Monograph in Theoretical Computer Science. Springer, Heidelberg, Germany, December 12, 2007.

[12] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer–Verlag, 1978.

[13] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[14] Dominique Cansell and Dominique Méry. Logical Foundations of the B Method. *Computing and Informatics*, 22(1–2), 2003.

[15] Dominique Cansell and Dominique Méry. *Logics of Specification Languages*, chapter The event-B Modelling Method: Concepts and Case Studies, pages in [11], 47–152. Springer, December 12, 2007.

[16] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19:45–80, 2001.

[17] John S. Fitzgerald. *Logics of Specification Languages*, chapter The Typed Logic of Partial Functions and the Vienna Development Method, pages in [11], 453–487. Springer, December 12, 2007.

[18] John S. Fitzgerald and Peter Gorm Larsen. *Developing Software using VDM-SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.

[19] Chris George and Anne E. Haxthausen. *Logics of Specification Languages*, chapter The Logic of the RAISE Specification Language, pages in [11], 349–399. Springer, December 12, 2007.

[20] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[21] Chris W. George and Anne E. Haxthausen. The Logic of the RAISE Specification Language. *Computing and Informatics*, 22(1–2), 2003.

[22] Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[23] Michael R. Hansen. *Logics of Specification Languages*, chapter Duration Calculus, pages in [11], 299–347. Springer, December 12, 2007.

[24] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[25] David Harel. On visual formalisms. *Communications of the ACM*, 33(5), 514–530 1988.

[26] David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[27] Martin C. Henson, Steve Reeves, and Jonathan P. Bowen. Z Logic and its Consequences. *Computing and Informatics*, 22(1–2), 2003.

[28] Martin C. Henson, Moshe Deutsch, and Steve Reeves. *Logics of Specification Languages*, chapter Z Logic and Its Applications, pages in [11], 489–596. Springer, December 12, 2007.

[29] Tony Hoare. Communicating Sequential Processes. Published electronically: `http://www.usingcsp.com/cspbook.pdf`, 2004.

[30] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.

[31] Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[32] Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[33] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications*. Addison Wesley, 1991.

[34] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety*. Addison Wesley, 1995.

[35] Stephan Merz. On the Logic of TLA+. *Computing and Informatics*, 22(1–2), 2003.

[36] Stephan Merz. *Logics of Specification Languages*, chapter The Specification Language TLA$^+$, pages in [11], 401–451. Springer, December 12, 2007.

[37] Ben C. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.

[38] Carl Adam Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[39] Wolfang Reisig. *A Primer in Petri Net Design*. Springer Verlag, March 1992.

[40] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, May 1985.

[41] Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer Verlag, December 1998.

[42] Wolfgang Reisig. The Expressive Power of Abstract State Machines. *Computing and Informatics*, 22(1–2), 2003.

[43] Wolfgang Reisig. *Logics of Specification Languages*, chapter Abstract State Machines for the Classroom, pages in [11], 15–46. Springer, December 12, 2007.

[44] A. W. Roscoe. *Theory and Practice of Concurrency*. C.A.R. Hoare Series in Computer Science. Prentice-Hall, 1997.

[45] Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, January 2000.

[46] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[47] Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2004.

Additional Tools

[48] NqThm/ACL2: http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html

[49] Isabelle/Hol: http://www4.informatik.tu-muenchen.de/~nipkow/LNCS2283/

[50] PVS: http://pvs.csl.sri.com/

[51] STeP: http://rodin.stanford.edu/

[52] SPIN: http://spinroot.com/spin/whatispin.html

[53] SMV: http://www.cs.cmu.edu/%7Emodelcheck/