

The previous chapter provides a general discussion of the different stages of specification in the software development process. It was assumed that the discrete steps of the development process have been well-defined. That is, the activities, deliverables, reviews, and analysis procedures associated with each step have already been established. It was suggested that a specification of the products and processes can be added to each step of such a well-defined development process. This chapter addresses specific issues that should be considered, activities that should be initiated, and the roles that are to be assumed when specifications are *formal* due to the integration of *formal methods* into the existing software life-cycle process for a given project.

Formal methods refer to the use of rigorous techniques founded on mathematics, in particular on abstract algebra, discrete mathematics and logic, in the representation of information necessary for the construction of software systems. The word “formal” comes from “formal logic” in which reasoning is done by virtue of “structure” and independent of “content”. The specifications in formal methods are “well-formed” statements in mathematics. Mathematical logic is used to specify a property that needs to be verified in the specified system. The property is true in the specified system if a set of inference rules in the logic can derive the property as a logical consequence. Each step of this deduction procedure follows from the preceding step when an inference rule is applied to it. Because content is not central to logical deduction, the inference procedure becomes a mechanical *calculation*. Thus, the verification process can be mechanized which in turn reduces reliance on human intuition. A formal method is to be preferred not just on its notations, but in its ability to adequately specify information and verify correctness of properties specified.

The word “method” comes from the context of engineering discipline and it denotes the *way in which a software process is to be conducted*. As stated in [20] in the context of system engineering, a method is defined to consist of an underlying *model* of development, *one or more languages*, a *defined set of ordered steps*, and *guidance* for applying these in a coherent manner. Model refers to the mathematical representation of the system. It is suggested by the chosen mathematical notation. For example, if set theory is chosen as the mathematical basis of formalism, then the expression $registered' = registered \cup \{Tom\}$ models the update operation when an item is added to a student file. By virtue of set the-

ory semantics $registered' = registered$ if $Tom \in registered$. Hence a formal specification language is a mathematically-based language, which has a well-defined syntax and semantics. The specification language must use the underlying mathematical objects as elements of the language and support formal verification. The expressive power of the language is brought out in the specification clarity, its support for removal of ambiguity and formal reasoning. These ultimately lead to an understanding of the system under development.

During the late 1980s and early 1990s, the initial stages of formal methods development, a *set of ordered steps* for system development was left undefined and no *guidance* (tool support) to practice safe specification was available. Since then formal methods have been maturing and several techniques and tools have been made available in public domain [29] to practice formal methods. They have been used successfully in engineering large complex systems [2, 3, 29].

2.1 Integrating Formal Methods into the Software Life-Cycle

Formal methods at different levels of formalization can be applied to any or all steps in the software development process. At some steps it may be sufficient to be rigorous, in the sense of being systematically precise without using the full power of mathematics and logic. As an example, in the late 1970s, Heninger and colleagues [14] at the Naval Research Laboratory introduced a tabular method to specify software system requirements. This method is rigorous, although not formal. In 1990, Van Schouwen [27] described a mathematical model and formalized the tabular methodology. Given the current maturity of formal [9, 29] methods, modern day software development practice can include formalization of behavioral specification, design specification, and program specification, and include a formal analysis of the system as well. Depending upon the type of project, it is necessary to decide the scope of formal methods to use and the level of formality desired at a particular phase in the development cycle. Based upon factors such as size of the project, application domain, scope of formal methods use, and level of formalism to be applied, it is necessary to determine the benefit-to-cost ratio of applying formal methods to the project prior to integrating formal methods in the development process.

Figure 2.1 is a process model in which formal method is integrated into all phases of life-cycle activities. This process model is to suggest the integration of formal methods in safety-critical systems development. It is assumed that domain experts construct a formal domain model which includes domain knowledge, concepts and ontologies, constraints among domain entities, and attributes of entities. A classification of different applications within the domain may also be included in the domain model. A set of requirements for a specific application is extracted from the domain model. The environment with which the system will interact is formalized and is combined with a formal specification of system requirements. The property to be formally verified in the system is formalized and is formally verified in the formalized system design. The verification is preceded by a formal

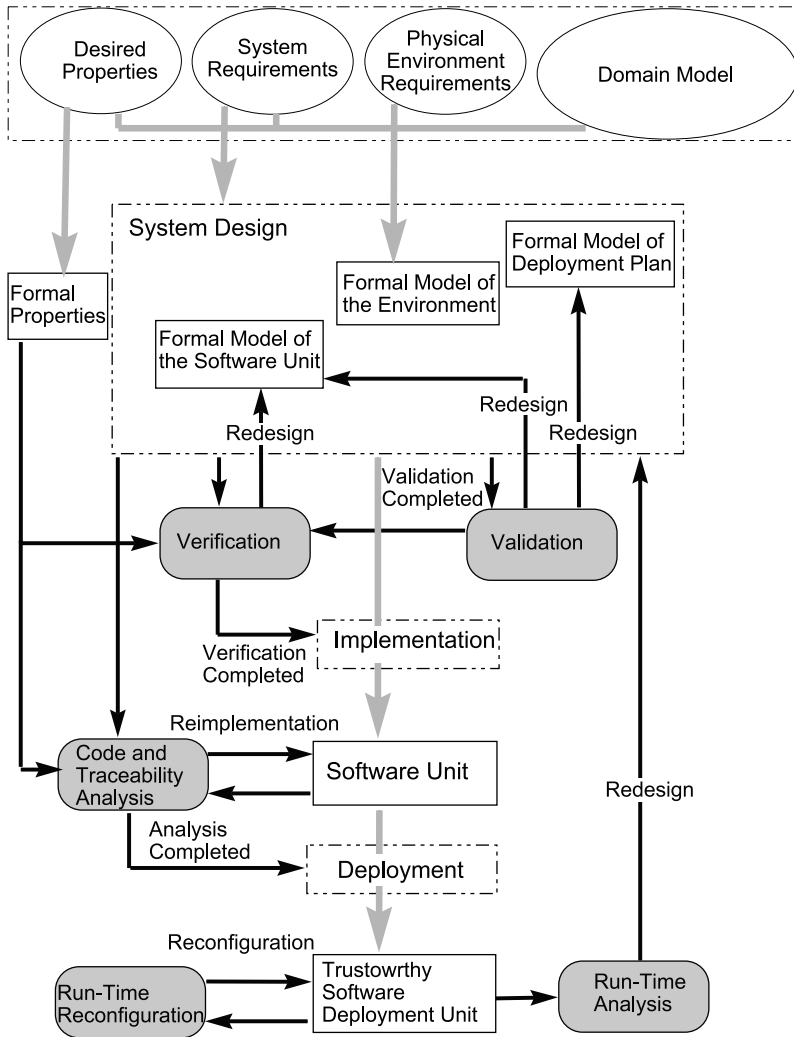


Fig. 2.1 Integrating formalism with process model

validation of system design to ensure that it has captured the stated requirements and environmental constraints. Such a validation may be an animation of the system being built. If validation fails the system design is to be redone. If verification fails the validation must be redone on the redesign. Only after verification is successfully completed the system implementation begins. Code analysis which includes tracing design units to implementation parts must be done before the system is deployed. Run-time configuration refers to the run-time system in the software deployment unit. If run-time analysis exposes errors a redesign of the system takes place. The level of formality and the choice of formalism are

important issues, although not part of the process model. The following technical factors [19, 20] influence the benefit-to-cost ratio.

Type of Application Formal methods may not be suitable for all types of applications. The characteristics of the problem domain and the complexity of their modeling should be evaluated to determine the suitability of applying formal methods to a project. If the project involves domains of high complexity, as discussed in the previous chapter, it may be advantageous to apply formal methods. However, problems over simple domains are usually less complex and do not warrant formal methods. Formal methods have been successfully applied to develop many safety-critical systems, and secure-critical systems [4, 8, 20]. It is desirable to formalize service-oriented systems, including E-commerce and web services, when trustworthiness of such systems is paramount.

Size and Structure Size and structural complexities should be evaluated prior to adopting formal methods in a project. A measure of application size used in industries is KSLOC, thousands of source lines of code. The NASA report [19] gives the following statistics:

Programs with size under 10 KSLOC have been subjected to verification. Most of the sub-systems that have been subjected to design-level specification and verification are in the range 10 KSLOC to 100 KSLOC. However, precise size figures for requirements specification are lacking. A reasonable estimate is that formal specification of requirements have been attempted on systems that eventually lead to systems on the order of 100 KSLOC.

From these statistics we may conclude that formal methods are effectively applied to systems of moderate size. It is also reported in [19] that formal methods cannot be applied in full to systems that use conventional programming techniques. To reap the full benefits of formal methods applied to large systems, they must be well-structured, and remain decomposable into well-defined components so that only critical components may be subjected to formal methods. When a system is composed of only loosely related components, which lack cohesion, formalization activity cannot be expected to be fruitful. Greater benefits result when formal reasoning conducted on each component can be composed to draw conclusions on the composite behavior of the system.

Choice of Formal Method and Type of Analysis The objectives for applying a formal method to a project must be clearly identified and documented. The development of safety-critical systems require the use of formal methods for specifying and analyzing critical components and their properties. An application which primarily uses the traditional structured development techniques may use formal methods only for the purpose of documenting data dictionaries. The objectives of these applications will have different impacts on the development process and consequently will influence different choices of formal methods.

Level of Formality Methods such as manual inspection and walk-through conducted with the help of documents written in a natural language and supplemented by diagrams, equations, and pseudo code are not formal. Table-based specifications and diagrams

used for object-oriented modeling [22] add more precision to natural language descriptions. These are only semi-formal notations. Specification languages such as Larch [11], VDM [17], and Z [24] have formal syntax and semantics, and also provide some mechanized support for syntax checking, semantic analysis and proofs. Methods such as B [1], PVS [21], and HOL [10] provide additional support for developing formal specifications, such as rigorous semantic analysis, refinement, and mechanized formal proof methods. Object-Z [23] and Alloy [16] integrate formal specification languages with graphical object modeling techniques. From the objectives of a given project, criticality of the application, project size, and available resources, the degree of formality suitable for the project must be determined and a choice be made from the above possibilities. The levels of formalization [20], in increasing levels of formality, are defined below.

1. A non-mathematical model of the system, such as data-flow diagrams, English text, and object diagrams, is translated to a mathematical description using notations from discrete mathematics and logic. An informal analysis of the specification may be done.
2. Formal specification languages with tools are used for syntactic analysis, pretty printing, and interpretation of the specification. Specification languages usually have built-in abstract data types that support specifying module interfaces, and object models.
3. Formal specification languages with formal semantics are used for specification. Tool support for analyzing specifications, say traceability analysis, and proof systems for mechanized verification are used.

Scope of Use Formal methods can be used in one or more dimensions of the development process. The degree of formality may also be varied across the different dimensions.

1. *Selecting development stages:* Although formal methods can be applied to all stages of the development process, it is usual to apply it only selectively. Depending upon the level of verification rigor appropriate to a project, a subset of requirements and high-level design may be chosen to undergo the techniques of a formal method. Integrating formal methods during the requirements and design stages has the advantage of enhancing the quality of the software. This is because errors can be detected during the early stages of the development process, and the precision injected early on leads to formal verification and validation.
2. *Choice of components:* Higher levels of rigor may be called for to assess the quality of safety-critical components. To construct such components, formalism is not only necessary but a high degree of formality should be applied. Components that are not critical may be subjected to lower levels of rigor.
3. *System functionality:* A proof of correctness is required to establish that the system has the important properties required of it. Whenever the objectives of a project include such strict requirement, the functionalities of those components designed to meet such requirements should be formally verified.

Tool It is not possible to apply formal methods with pencil and paper. To apply it with sufficient rigor, tool support is necessary. Since a tool may address one or more of the issues, developing a formal specification, syntax checking, semantic analysis, and theorem

2 proving, the choice of tools for a project depends on all the factors discussed above. Tool support and good expertise are required to refine designs into programmable modules and conduct proofs on the correctness of refinements. Many tools are now available as open source software [9, 29] for practicing formal methods at almost all stages of the software development cycle.

2.2

Administrative and Technical Roles

Once a decision has been reached on adopting a formal method to a project, general guidelines be put in place to implement this decision before the project begins. The guidelines include mechanisms for documentation standards for improved communication, configuration management, and reuse of specifications. When the existing development process has well-defined steps, formal methods can be inserted at relevant steps in the entire process or it can be applied on a small scale to some of the steps. A pilot study may also be done to integrate formal methods to understand the steps where it is most effective, and train staff for these activities. Although the roles and sequence of activities of the staff depend on the specifics of the process model, a discussion of the roles based on a rough classification of roles is given below.

2.2.1

Specification Roles

System requirements, environmental objects and their constrained interactions, and the property to be verified in the software system are specified by those assuming specification roles. Naturally, these formal specifications are developed by groups of people who have a good understanding of the formal languages used for the specifications. A specifier may be the author as well as the analyzer for a specification unit. As an author, the specifier constructs the specification corresponding to a process or a product in a formal language. This activity involves a good understanding of language abstractions and the properties of the product or process. As analyzer, the specifier demonstrates the inherence of desired properties in the specification. In particular, the analyzer resolves inconsistencies and demonstrates the coherence of the specification. A specification may be refined to include more information. Whenever a specification is refined, it is required to establish the satisfaction of the refined specification to its source. This can be done by an informal analysis; however, within a strict formal framework, a proof is required.

In this role some staff are expected to field questions about tools, domain issues, and sufficiency of coverage, which arise during formalization and validation. They may assist users and other members of the development team in understanding the specification document. This may be done through natural language expositions and graphical

illustrations to convey the meaning of formal constructs. In a typical walkthrough session, the specification staff shall demonstrate that there exist requirements in the SRD corresponding to every formal specification unit and vice versa. The goal of such sessions is to have demonstrated to the user that the formal specification document fully captures the requirements stated in SRD. The staff, in collaboration with other members of the team, may develop appropriate tools for traceability and reuse of specifications.

A formal specification, however, is not a panacea. Constructing a formal specification involves characterizing the intended correctness conditions. Missing on them, failing to state them correctly, and stating inappropriate conditions lead to surprising situations. Thus a specification will not, on its own, insure that no errors will be made, nor that the final product will be error-free. Since errors in a specification will have a detrimental effect on all future stages of software development, the specification must be analyzed to eliminate all errors. The kinds of errors to look for in walkthroughs and other forms of analysis include the following:

- missing, incorrect, redundant requirements,
- errors due to misuse of the specification language,
- logical errors in specifying correctness conditions,
- inappropriate inclusion of constraints in correctness conditions, and
- incorrect translation of requirements.

The role to be played here is to accumulate sufficient evidence to show that the formal specification is free of these types of errors. This role, commonly known as to *validate*, must be played with the collaboration of the end users. Ideally, the specification staff validate the specification by *executing* the specification in the presence of the customer.

Another important activity that the specification staff must undertake is to assist the test team understand and use the specification for designing functional tests for the software product. Since, after validation, the specification contains expected and correct functionalities of the product, the test team has the necessary information for testing the final product. The main advantage in playing out this activity is that the large amount of work involved during a *posteriori* error detection is replaced by a more scientific effort spent *a priori* during the construction of software.

2.2.2

Design Roles

An important role of a project staff is to be a design engineer. Design is concerned with constructing artifacts and assembling them together to produce the intended effect of a software product. Kapor [18] and Holloway [15] liken software designers to architects. The rationale is that architects have the overall responsibility for constructing buildings and engineers play a vital role in the process of construction. It is the architect who, upon receiving the requirements for constructing a house, produces a design which ultimately

2 produces a “good” building that “pleases” the client. Engineers, taking directions from the architect, put things together by choosing components that are well tested. Similarly, in software systems, the designer is the architect who receives the validated requirements specification and produces a design to meet the overall needs of the user. In the house case, the components are the building materials, while in software development they are akin to the rooms and corridors of a house. The building materials of software development include previously built, tested and certified, programming languages, utilities, and middleware.

Although the design of the system begins with system requirements analysis, the design activity takes shape only after the components to be used in building the system are identified. The design activity proceeds according to project guidelines, and technical considerations planned for integrating formal methods in this stage. A formal design would involve several levels of *refinements*, starting from requirements specification. At each level, a refinement adds more details to the specification in the preceding level. The details may include data structuring details and/or algorithmic details. Part of refinement process is a proof that shows the behavioral satisfaction of the refinement to its predecessor. However, when the design is not fully formal the design team should ensure that critical components of the design are formalized. This is done by inventing state invariants, contracts on interface specifications, and other assertions on the design components, and expressing them in the same formalism used for requirement specification. This will enable a formal validation of the critical components in the design. However, if the chosen formalism cannot specify all aspects of the design, then different specification formalisms may have to be mixed. In [6], Object-Z and CSP are mixed to specify object-based system design. In [25], Z and timed CSP are combined to specify safety-critical systems. These two hybrid specification methods have successfully exploited the state machine semantic domains of Z and CSP. In situations when the underlying semantic domains are different the design team will face a difficult task in establishing semantic consistency and proving that design satisfies the requirements.

Regardless of whether a design is fully or only partially formal, a good design is one which can easily adapt to changes in the requirements and the environment. Toward achieving a good design, the design team should give strong considerations to the misgivings of domain specialists, even if they cannot substantiate these misgivings [15]. The design team should periodically meet with clients and specification staff to confirm changes in design caused by changes in specification. Thus, design staff play a pivotal role in software construction—they interact with clients, specification staff, and programming teams.

2.2.3

Implementation Roles

The term “implementation” is used in a broad sense to include code generation, code analysis, and deployment. A staff in implementation team is a programmer whose role will vary greatly depending upon the level of formality used and tool support. In case code generation is automated with the use of tools, as in B method, the programmer should

undertake a traceability analysis to ensure that code covers the expected behaviors expressed in the design. Typically the generated code is written in some imperative language. For example, in B method the code is written using a B language sub-assembly, similar to an imperative programming language. In order to facilitate code generation on any target system, either the programmer has to translate installations or translation is done automatically to standard programming language. The programs obtained can then be compiled and assembled on the target machine to produce the executable software. In case the code generation is not automatic, the programmer's task is to take the designer's output and write programs consistent with the detailed design. It is the programmer's responsibility to ensure both correctness and efficiency in translating the design decisions into source code. However, it may happen that certain aspects of the design are difficult or impossible to implement. The programmer brings those design aspects which cannot be implemented to the attention of the design and specification staff. The rationale for unimplementable design aspects may be due to strong constraints and/or stringent requirements. They may traced back to the design and requirements, and rectified.

The programmer and the test engineer who developed specification-based testing collaborate in testing programs. The outcome of testing determines whether or not a program correctly interprets the requirements of the customer. In case of errors, the specification team is brought in for consultation. In particular, the programmer will make changes after the specification staff and the design team work out a new design. Model checking tools and test case generation tools are available [9] for this purpose. Traceability and run-time analysis are central to deployment of software. These activities exist even when formal method is not integrated with the development process.

2.3 Exercises

1. Enumerate the essential requirements of a communication channel between two processes. Identify safety and security properties.
2. Assume that software for Automated Teller Machine (ATM) is to be constructed following the process model in Fig. 2.1. Answer the following questions:
 - What is the domain of application? Enumerate a few domain entities, the attributes for each entity, and the relationship among the entities. What properties of entities govern security? Develop a property to be verified in the software implementing the ATM.
 - Describe the environment for ATM to function. What constraints are to be imposed by the software on the environment? What constraints the environment may impose on the software?
 - What software components should be formalized? For each formal unit state the property to be verified in it.

2.4

Bibliographic Notes

The factors to be considered for choosing specific tasks for formal methods application is discussed in the NASA reports [19, 20]. These reports outline the technical and administrative considerations that must be reviewed before integrating formal methods into a development process. It is stated in the report that for an effective application of formal methods to a project, the team responsible for applying formal methods must be trained in formal methods and tools. In a debate held at the Tri-Ada '94 [26], panel discussion the panelists argued that the expertise required to use formal methods can be gained through education and training courses spanning a few weeks. An understanding of the activities, skills and responsibilities associated with the roles involved in software development using formal methods is obtained from looking at the success stories [3] as well as learning from the failures [15].

Analysis and interpretation of formal specifications are enhanced with the help of tools. Several tools are available as open source software [29]. Tools specific to Z, Object-Z, VDM, and B method can be found in [9]. For Larch specifications, Guttag et al. [11] describes a syntax checker and a theorem prover. Larch Prover (LP) is a proof assistant incorporating several proof techniques for rewrite-rule theory. PVS [21] provides an integrated environment, formalism and methods supported by tools, for the analysis and development of formal specifications. Tool components include parser, type checker, browser, specification libraries, and integrated proof checker. The early work by Dick and Faivre [7] on black-box testing based on VDM specifications and Hierons [13] for generating test cases from Z specifications have led to study model-based testing methods. The report [12] compares many such tools from industry and academia.

Two excellent treatises on design are the book by Winograd [28], which is a collection of articles showing the diverse perspectives of software design, and the book by Dasgupta [5], which explores the logic and methodology of design from the computer science perspective. Holloway [15] draws lessons to software engineering from design failures in building physical structures, such as bridges and rockets. They are

- *Lesson 1: Relying heavily on theory, without adequate confirming data, is unwise.*
- *Lesson 2: Going well beyond existing experience is unwise.*
- *Lesson 3: In studying existing experience, more than just the recent past should be included.*
- *Lesson 4: When safety is concerned, misgivings on the part of competent engineers should be given strong consideration, even if the engineers cannot fully substantiate these misgivings.*
- *Lesson 5: Relying heavily on data, without an explanatory theory, is unwise.*

References

1. Abrial J-R (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge

2. Clarke EM, Wing J (1996) Formal methods: state of the art and future directions. *ACM Comput Surv* 28(4):626–643
3. Craigen D, Gerhart S, Ralston T (1995) Industrial applications of formal methods to model, design and analyze computer systems
4. Crow J, De Vitto BL (1996) Formalizing space shuttle software requirements. In: *ACM SIGSOFT workshop on formal methods in software practice*, San Diego, USA
5. Dasgupta S (1991) *Design theory and computer science*. Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge
6. Derrick J, Boiten E (2002) Combining component specifications in object-Z and CSP. *Form Asp Comput*, pp 111–127
7. Dick J, Faivre A (1993) Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock JCP, Larsen PG (eds) *FME93: industrial-strength formal methods, formal methods Europe*. Lecture notes in computer science, vol 670. Springer, Berlin
8. De Vito BL, Roberts L (1996) Using formal methods to assist in the requirements analysis of the space shuttle GPS change request. NASA Report 4752, Prepared for Langley Research Center
9. Formal methods web page. http://formalmethods.wikia.com/wiki/Z_notation, May 2010
10. Gordon MJC, Melham TF (eds) (1993) *Introduction to HOL*. Cambridge University Press, Cambridge
11. Guttag JV, Horning JJ, Garland SJ, Jones KD, Modet A, Wing JM (1993) *Larch: languages and tools for formal specifications*. Springer, Berlin
12. Hartman A AGADIS: model-based generation tools. Technical report. http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf
13. Hierons RM (1997) Testing from a Z specification. *Softw Test Verif Reliab* 7:19–33
14. Heninger KL et al. (1978) Software requirements for the A-7E aircraft. NRL Report 3876, Naval Research Laboratory
15. Holloway CM (1999) From bridges and rockets: lessons for software systems. In: *17th international system safety conference*, Orlando, Florida, USA, pp 598–607
16. Alloy DJ (2002) A lightweight object modeling language. *ACM Trans Softw Eng Methodol*, 11(2):256–290
17. Jones CB (1990) *Systematic software development using VDM*, 2nd edn. Prentice-Hall International, Englewood Cliffs
18. Kapur M (1996) A software design manifesto. In: Winograd T (ed) *Bringing design to software*. ACM Press, New York, pp 1–9
19. Formal methods specification and verification guidebook for software and computer systems, vol I: Planning and technology insertion. NASA Report NASA-GB-002-95, Release 1.0, July 1995
20. Formal methods specification and analysis guidebook for the verification of software and computer systems, vol II: A practitioner's companion. NASA Report NASA-GB-001-97, Release 1.0, May 1997
21. Owre S, Rushby JM, Shankar N (1992) PVS: a prototype verification system. In: Kapur D (ed) *Proceedings of the eleventh international conference on automated deduction (CADE)*, Saratoga, New York, June 1992. Lecture notes in artificial intelligence, vol 607. Springer, Berlin, pp 748–752
22. Rumbaugh J, Blaha M, Pramerlani W, Eddy F, Lorensen W (1991) *Object-oriented modeling and design*. Prentice-Hall, Englewood Cliffs
23. Smith G (2000) *The object-Z specification language*. Kluwer Academic, Norwell
24. Spivey JM (1988) *Understanding Z, a specification language and its formal semantics*. Cambridge University Press, Cambridge
25. Sühl C (2000) Applying RT-Z to develop safety-critical systems. Lecture notes in computer science, vol 1783. Springer, Berlin
26. TRI-Ada '94 formal methods panel summary. <http://shemesh.larc.nasa.gov/fm/paper-tri-ada.html> (04/16/2010)

27. van Schouwen AJ (1990) The A-7 requirements model: re-examination of real-time systems and an application to monitoring systems. Technical report 90-276, Department of Computing and Information Science, Queens University, Kingston, Canada
28. Winograd T (1996) Bringing design to software. Addison Wesley, New York
29. Wheeler DA (2010) High assurance (for security or safety) and Free-Libre/Open Source Software (FLOSS) ... with lots on formal methods/software verification. <http://www.dwheeler.com/essays/high-assurance-floss.html> (04/16/2010)



<http://www.springer.com/978-0-85729-276-6>

Specification of Software Systems

Alagar, V.S.; Periyasamy, K.

2011, XXVI, 646 p., Hardcover

ISBN: 978-0-85729-276-6