

Applying Formal Specifications to Real-World Software Development

Girish Keshav Palshikar, *Tata Research Development and Design Center*

All too often in the software industry, system requirements, as stated near the end of the system analysis phase, lack clarity, focus, and confidence. The requirements might be incomplete, inconsistent, or ambiguous—or include unnecessary information about design choices and implementation details. Requirements written in informal notations can be neither rigorously analyzed for properties nor used as prototypes.

Formal techniques remain absent from most development processes. This article presents practical guidelines for applying formal methods to development projects, particularly in the requirements specification stage.

These defects affect activities throughout the software development life cycle (SDLC). Users often become aware of the lacunae in the requirements after starting acceptance or certification tests on the final software system. This leads to software maintenance—an expensive and time-consuming process—to incorporate the new and changed requirements. Researchers have suggested several strategies, besides improved processes and methodologies, to avoid requirements problems, including early prototyping¹ and traceability.²

Formal methods have gained some acceptance in the software development industry, and *formal specification*, which refers to a mathematical description of the system's requirements, can greatly benefit requirements specification. Established development notations, methodologies, and processes have yet to firmly incorporate

formal specifications, however. This article addresses that gap by outlining practical guidelines—rather than a complete methodology in the form of steps, milestones, deliverables, and so on—for applying formal specifications to software development. I have distilled these pragmatic tips from direct experience using formal specifications in industrial projects.

Benefits of formal specifications

Formal methods, described in the sidebar “Formal Specifications: A Quick Reference,” advocate the use of mathematical notations and methods throughout the SDLC and encompass formal specifications, formal program derivation through refinements, and program verification. This article focuses only on the use of formal specifications and not on other aspects of formal methods.

Formal Specifications: A Quick Reference

Formal specification techniques advocate using mathematical notations to state the requirements of a system to be built. Mathematics brings clarity and tool support for rigorous analysis. Developers have used formal specifications in a variety of application domains:

- Real-time, safety-critical, mission-critical, or embedded software, characterized by special-purpose user interface (usually simple), small and simple data, timing constraints, complex dynamic behavior, multiple modes of control and operation, distributed or concurrent components, interfaces to physical environment and devices, safety, fault-tolerance and other reliability related-requirements, and external certification. Such systems abound in avionics, aerospace, defense, railways, nuclear power plants, manufacturing and process control systems, and biomedical systems.
- Business application systems, such as accounting and library management, characterized by large and complex databases, transaction processing, complex user interfaces, distributed client-server architectures, complex business functionality, frequent changes, and long legacy.
- System software, including network protocols, operating systems, database systems, compilers, and so on.
- Hardware-oriented systems that include digital and VLSI circuits and microprocessor-based systems.

A growing body of literature reports on the industrial use of formal methods:

- Formal methods in railway applications, www.ifad.dk/Projects/fmemail.htm
- Formal methods in aerospace applications, <http://shemesh.larc.nasa.gov/fm>
- Formal methods in aviation, www.icas.edu/newresearch/des/formal.html
- *Proceedings of the World Congress on Formal Methods and Formal Methods Europe*, www.fmeurope.org

See www.cafm.sbu.ac.uk/fm for overviews and links to tools, bibliographies, and other resources. Many journals include articles on formal methods, including *IEEE Software*, *IEEE Transactions on Software Engineering*, *Information and Software Technology*, *Formal Aspects of Computing*, *Formal Methods in System Design*, and *The Computer Journal*. The Internet newsgroup comp.specification.misc contains discussions of related issues. A small collection of notations and tools for formal specifications (with an associated Internet Web site) appears below; most notations also have tools associated with them.

Some formal methods notations

- Z, www.comlab.ox.ac.uk/archive/z.html
- Vienna Development Method (VDM), www.ifad.dk/vdm/vdm.html
- Duration Calculus, www.iist.unu.edu/dc
- Message Sequence Charts, www.win.tue.nl/~michelr/msc.html

- Specification and Description Language (SDL), www.sdl-forum.org
- Estelle, www.estelle.org
- Lotos, www.cs.stir.ac.uk/~kjt/research/well/well.html
- Communicating Sequential Processes (CSP), www.comlab.ox.ac.uk/archive/csp.html
- Esterel, www.esterel.org
- Statecharts, www.di.ufpe.br/~lrl/statecharts
- Petri Nets, www.daimi.aau.dk/PetriNets/Tools
- OBJ3, www-cse.ucsd.edu/users/goguen/sys/obj.html
- Standard ML, www.cm.bell-labs.com/cm/cs/what/smlnj

Some formal methods tools

- Z/EVES tools for Z specifications, www.ora.on.ca/z-eves
- VDMTools for VDM, www.ifad.dk/Products/products.html
- LARCH tools for algebraic specification, www.sds.lcs.mit.edu/spd/larch
- Rigorous Approach to Industrial Software Engineering (RAISE) tools, <http://spdweb.terma.com/Projects/RAISE>
- B tools for behavioral specifications, www.b-core.com/btoolkit
- DCVALID verification tool for Duration Calculus, www.tcs.tifr.res.in/~pandya/dcvalid.html
- Pi-calculus based Mobility Workbench tools for mobile processes, www.docs.uu.se/~victor/mwb.html
- PROMELA/SPIN and SMV tools for model-checking, www.cm.bell-labs.com/cm/cs/what/spin/Man/promela.html or www.cs.cmu.edu/~modelcheck/smv.html
- HOL, Isabelle, PVS, and Coq theorem provers, www.cl.cam.ac.uk/Research/HVG/HOL, www.cl.cam.ac.uk/Research/HVG/Isabelle, <http://pvs.csl.sri.com>, or <http://coq.inria.fr>
- UPPAAL tools for verification and validation of real-time systems, www.docs.uu.se/docs/rtmv/uppaal
- Concurrency Factory tools for specification and verification of concurrent systems, <http://cs.sunysb.edu/~concurr>
- ObjectGEODEtools for SDL-based specification and design of real-time systems, www.tdr.dk/public/SDL/verilog/ogeode.html
- EventStudio tools for message sequence charts, www.eventhelix.com/EventStudio/FeatureList.htm

Hossein Saiedian and his colleagues offer a thorough discussion of the various issues related to formal methods.¹ J.M. Wing provides a simple overview of formal specification techniques,² and *Real-Time Systems: Specification, Verification and Analysis* provides an opportunity to learn and compare many formal specification notations through their use in specifying a common example of a mine-pump controller system.³

References

1. H. Saiedian et al., "An Invitation to Formal Methods," *Computer*, vol. 29, no. 4, Apr. 1996, pp. 16–32.
2. J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, vol. 33, no. 9, Sept. 1990, pp. 8–24.
3. M. Joseph, ed., *Real-Time Systems: Specification, Verification and Analysis*, Prentice-Hall, New York, 1996.

Rigorous mathematics, aided by prototyping and proofs, leads to early requirements problem detection. Tools can sometimes derive more practical benefits from formal specifications for code generation, refinement, test generation, and the like. Some researchers suggest that slow industry acceptance of formal specifications stems from the perceived difficulty of mathematical (nondiagrammatic) notations; the training, effort, and costs required to adopt the technology; and lack of tools.³ Several have debated the practice and pros and cons of applying formal methods in industrial projects.⁴⁻⁷ Despite these apprehensions, developers in many application areas now experiment with formal methods.

The growing use of formal methods in industry has generated some misperceptions and unrealistic expectations, but extensive discussion of some common myths can dispose some of these expectations for new users.^{8,9} The need remains, nonetheless, to provide more proactive guidance on building better formal specifications. This article takes a step in that direction by offering 15 guidelines for developing and applying formal specifications.

Guidelines for developing formal specifications

Typically, a systems analyst produces the informal *user requirements specification* document, whereas a specification engineer prepares formal specifications based on the URS and effectively applies the formal specification technology within a project. Next, the specification engineer must construct formal specifications that are readable, well structured, reusable, validated, and correlated with the URS.

Guidelines 1 through 5 deal with the *process* of developing and using formal specifications; 6 through 15 with formal specification *contents*. These guidelines are not necessarily objective or complete. You'll better appreciate them (and perhaps improve them) by using them to write formal specifications. Occasionally, we use Z¹⁰ to illustrate a point, but the guidelines don't depend on a specific formal notation or system.

1. Clearly define the role of formal specifications in your software development process.

Industrial development processes often

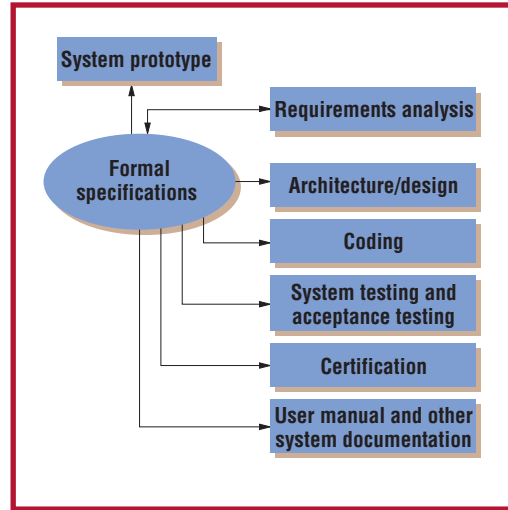


Figure 1. Using formal specifications in different stages of the development life cycle.

have no place for formal specifications,⁴⁻⁷ with a few exceptions such as SOFL, CORE, and, partially, UML. Developers often ask whether they should use formal methods in a given project. It's possible to assess this need by balancing the expected benefits against the costs and problems; the tools and training needed; and the nature and circumstances of the specific organization and project, including manpower, schedules, and certification.

Even apart from safety-critical, mission-critical, and real-time systems, most applications *do* contain some core critical requirements that will likely benefit from formal specifications. Arguably, formal specifications offer the significant possibility of early problem detection, potentially leading to large savings in rework. Building formal specifications can also lead to a somewhat longer analysis phase, though, and requires close cooperation between the specification engineer and systems analyst or even end users.

Once you've decided to develop formal specifications, you must augment the development process to include this new activity. A project can use formal methods technology in numerous ways; not all requirements must be specified formally, and more advanced formal techniques (such as refinement, theorem-proving, or code verification) might not always be needed. A common model of the formal development process, shown in Figure 1, includes building formal specifications and using them (along with derived outputs such as code, prototypes, test cases, and documentation) as references

Developers must define expectations of and ways to use the formal specifications in the development process.

in the development process.

Several questions arise when we include formal specifications in the development process. Who will build the formal specifications? Who are the stakeholders? Who will use them, for what purposes, and how? What are the tools, costs, and training requirements? Estimating the efforts needed for building formal specifications with little explicit knowledge of all the factors presents some challenges.¹⁰ As a heuristic, the efforts generally amount to about half of those for the analysis.

Developers must define expectations of and ways to use the formal specifications in the development process, keeping track of the changes to the URS based on the inputs from formal specifications. They must also adapt the change management processes to keep the URS and formal specifications in sync. In particular, we should define a specific entry point where work on formal specifications could begin, perhaps near the end of the analysis phase when most requirements are gathered and stated in some form.

We must also define the criteria for validating the formal specifications—whether through reviews, prototypes, proofs of properties, or model checking—and decide the role, if any, system end users will have in formal specifications review and understanding. Reviewers and team members might need training to understand formal specifications. The reviews must address how to compare the user requirements written in formal and informal notations. Reviews should also report all questions raised and ambiguities, inconsistencies, and other problems discovered during the process of writing the formal specifications, to help identify how the effort adds value to the project.

The development team must also define how formal specifications will apply to the rest of the development process:

- Will the formal specifications be refined? If so, by what criteria, and when will the refinement terminate?
- Will the code generated from the formal specifications be used, and if so, how does the code interface with the remaining (informally developed) components?
- Will formal specifications form a reference during the design, programming,

and testing phases, and in which specific ways?

- Is it necessary to demonstrate, through reviews, tests, or proofs, that the design or code meet the formal specifications?
- Do we need to maintain a traceability document relating the formal specifications to other documents and code?
- Do we need to maintain formal specification versions and relate them to the URS?

The team must define answers to these questions before embarking on the project. A document outlining the relationship between the formal specifications and the URS document might prove helpful. It might also prove useful to build a dictionary summarizing the important components (such as terms, variables, constants, functions, schema, and modules) of the formal specifications to help with understanding and maintaining the specifications and for cross-reference and traceability. You can define metrics to measure the complexity of formal specifications; for Z, simple measures include number of types, variables, schema, and the state-space size, if finite.

2. Define the steps for developing formal specifications.

Preparing formal specifications in different notations will involve different steps. The system requirements should be prepared in two sequentially organized phases:

- a URS written in natural language and the methodology notations (for example, use cases in UML), and
- formal specifications written, say, using the Z notation.

Apart from the URS, other requirements sources might include concept notes, requests for proposals, a system requirements specification, external interfaces, a system architecture and deployment document, and background domain knowledge.

First, the systems analyst prepares the URS using standard analysis methodologies involving end-user interactions. Stakeholders (such as users, domain experts, and managers) review the URS, which often undergoes a series of revisions to take into account the review comments at each stage. The de-

development team then creates and approves a final URS version. Once (or somewhat before) URS approval occurs, the specification engineer can prepare the formal specifications of the user requirements.

The formal specifications development process is hardly monolithic and atomic; it also varies from notation to notation. The process of preparing Z specifications often includes the following steps:

- data model: constants, sets, types, functions and predicates, data model with integrity constraints, and global data state;
- data manipulation operations;
- validation and prototyping; and
- reviews and analysis.

Additional steps might follow, such as refinement and code generation, review of design and code against formal specifications, and derivation of the acceptance test cases from formal specifications. For reactive notations such as Esterel and state charts, you could begin with interface definitions and proceed top-down from high-level abstraction to more detailed levels, using these notations' hierarchical organization facilities.

3. Carefully divide specifications among broad classes such as functionality, operations, behavior, and interface.

System requirements often fall into the following somewhat orthogonal categories: functional, dynamic behavioral, operational (user interface), external interface, and quality (Figure 2). You must carefully design criteria to classify a particular system requirement.

Roughly, a functional aspect is anything that maintains and manipulates the system's data state. Dynamic behavioral aspects include control flow (state transitions) among the functional aspects, operation modes, system responses to input events (including faults), and so on. Operational (or user interface) aspects include facilities that let users view and manipulate the system: displays, reports, presentation, navigation, dialogues, configuration, and so on. The external interface deals with accessing and providing services to external systems—for example, message structure, timelines, message exchange protocols, and service inter-

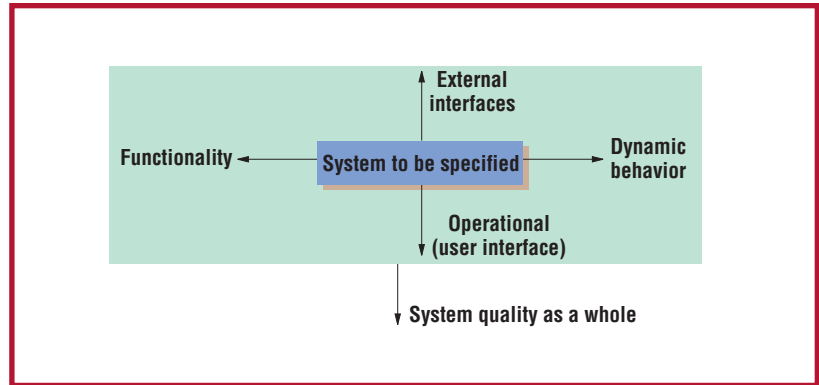


Figure 2. Classifying system requirements.

faces. Quality requirements often include only a textual description of the intended system's quality as a whole, including timing constraints, fault tolerance, and performance. Additional requirements might include deployment, architecture, or technology platforms. Obviously, these criteria might not always help in unambiguously classifying some requirements.

Developers should use appropriate notations to specify requirements in each class. Using different notations lets you exploit the native advantages of each. Such partitioning also facilitates *specification independence*: independent analysis, maintenance, understanding, and refinement of each class. We must, however, be sure to integrate and maintain the requirements expressed in different notations.

4. Select appropriate parts of your application for formal specification.

Most systems' structure falls into some natural components; the requirements then also fall into the corresponding natural parts. Usually, formally specifying all components' requirements could result in too big an effort; developers should instead choose "critical" components or parts of the system requirements for formal specification. The rest of the requirements can be stated in the methodology notations. Components with important requirements (that is, any problem that might lead to serious rework) generally make good candidates. Developers should also restrict the formal specifications to certain types of requirements and, for example, exclude the user interface requirements. It is often neither necessary nor practical to formally specify all URS requirements, and justification for

Developers should model only those aspects of the application system integral to the system and the user's understanding of it.

choosing certain requirements classes for formal specification must be documented clearly.

Although this division of requirements into formally and informally stated reduces formal specification efforts, it also generates new problems. For instance, how do we integrate the formal and informal requirements conceptually as well as for analysis and review? What about maintenance? No easy solutions exist, but we must remember that experience shows us it's better to formally specify some requirements than none.

5. Choose appropriate formal specification notations and tools for each class of requirements.

Many notations are available for formal specifications, each based on a particular mathematical framework (such as finite-state systems, process algebra, temporal logic, abstract algebra, λ -calculus, set theory, and predicate logic). Also, theorem provers and model-checking tools offer rigorous notations useful for formal specifications.

Different notations work best for expressing different requirements classes. Factors in choosing a notation include its nature (mathematical, textual, diagrammatic), suitability for the requirements class, supporting tool capabilities, costs, training needs, resources, and available case studies. It might not always be clear which formal specification notation is suitable for which type of requirements. Broadly speaking, data-intensive (or predominantly functional) requirements are better expressed in notations based on logic, set theory, and abstract algebra. Control-intensive (or predominantly dynamic behavioral) requirements are better expressed in process algebraic or automata-theoretic notations.

Typically available tools include syntax-directed editors, type checkers, pretty printers, proof obligation generators, theorem provers, verification tools, refinement tools, prototyping and simulation tools, code generators, test-case generators, and documentation generators. Tools differ greatly in their capabilities, utility, and costs, and the team must evaluate them based on how they plan to use formal specifications in the rest of the life cycle. For instance, if they will refine requirements into design, they'll need good refinement tools; if end users must approve the formal specifications, visual nota-

tion or prototyping facilities might be useful.

Of course, no single formal specification notation can capture all requirements of complex, large-scale, real-life applications. Developers might need to choose more than one notation and then find a way to integrate and analyze the requirements expressed in different notations.¹¹

6. Maintain high-level abstraction by avoiding design decisions and implementation details.

Formal specifications serve only to clearly state system requirements. Hence, developers should model only those aspects of the application system integral to the system and the user's understanding of it, pushing away all thoughts of how the system can, will, or should be implemented. Strive to write the specifications in terms as abstract and high-level as required, away from design and implementation. Avoid mentioning any details about the system's architecture, design, or implementation details such as data structures, algorithms, or technology platforms.

Software engineers will use the formal specifications in later SDLC stages for design and implementation. The specifications should give them freedom to make the design and implementation decisions best suited for the system's characteristics and desired quality requirements. Design decisions include data structures, algorithms, detailed error-handling mechanisms, database table design, class library design, and division of functionality among clients and servers. Such decisions are best left out of the formal specifications.

This is, of course, an ideal. In practice, such details might be necessary, particularly in specifications refinements, to take advantage of technology platforms or facilities of abstraction, generalization, and modularity in the programming language. Reviewing completely abstract requirements can also be difficult. Therefore, seek a pragmatic balance between abstraction and necessary details.

7. Avoid over- and underspecification as well as nondeterministic and partial specification.

A developer might err toward *overspecification* when including information that either is unnecessary or unnecessarily constrains the later life cycle phases. As in

Guideline 6, including design information is overspecification. To avoid another kind of overspecification, model only that data directly visible to or understood by the end user or essential for meeting system requirements. Do not, in general, specify internal or bookkeeping data.

Underspecification might result from excluding some necessary information in a system specification, leaving it incomplete. For example, an operation in a functional specification might be incomplete if it does not adequately handle all input values. If the specification engineer does not describe all necessary operations on the data, the system is incomplete from the user's point of view. In general, it is exceedingly difficult to determine whether a given specification suffers from any over- or underspecification. The proofs of system properties and system prototype based on the formal specifications help in detecting such problems.

Nondeterminism offers a useful, succinct way to specify the arbitrary nature of internal system choices. Unfortunately, it often confuses end users and is better avoided. Partial specifications should also be avoided for the sake of clarity and reviews, as should given (unspecified) sets or infinite sets, wherever possible. When completing the Z schema, be sure to use appropriate and easily understood error messages.

8. Build modular specifications containing “good” and “natural” structure.

Intuitively, we see a program (or a related document) as “good” or “bad” depending on its how much structure it has and how natural that structure is. Similarly, developers can write good or bad formal specifications. A good formal specification has plenty of structure and is built from numerous structural units using notational facilities such as compositions or hierarchy. Be careful, however, that formal specification structure does not reflect possible architectural or design choices to implement the system. A good formal specification's structure should reflect the system's natural structure. A well-structured specification is easy to understand, document, and argue about.

What constitutes a natural structure is a hard and subjective question. To progress toward a well-structured formal specification, show that it closely resembles the re-

quirements' conceptual structure, which we can express using notations such as concept graphs.¹²

9. Always try to find many alternate representations and then choose the one best suited for the problem.

In any modeling exercise, it is tempting to quickly accept the first description that seems suitable for a particular aspect. The first choice might not always be best suited for the aspect; it might not even be correct. For example, you could model a bookshop's book stock as a function stock: $\text{BOOK} \rightarrow \text{seq PLACE}$ from a book ID to an ordered list of shelves where they can be found; or as a binary relation stock: $\text{BOOK} \leftrightarrow \text{PLACE}$ between books and places. The first representation helps if the places are ordered in some sense (such as floors); it directly records all places at one place, whereas the second does not. However, the second representation is mathematically neater and simpler; it facilitates easier statement of certain stock operations.

Developers must think of alternate representations and then choose that which is simple and natural to the system. Do not overspecify by adding unnaturally constraining representations, and provide clear justification for the representation choices.

10. Build specifications for reusability using libraries of useful metaphors and patterns.

Acquire a library of useful metaphors for describing a system, perhaps distilled from past experience in developing formal specifications. These ready-to-use conceptual tools can simplify the task of developing formal specifications; you just need to recognize a system aspect that can be modeled in terms of the ready metaphor. For example, consider an abstract system of multiple resources, users of these resources, and a resource controller that provides controlled access to these resources. A formally specified metaphor might take the form of scheduling the use of berths in a port for ships or allocation of operation theaters in a hospital. Other examples of such metaphors are masters and slaves, clients and servers, and so on.

Keep looking for such reusable parts of the specifications, and carefully specify reusable parts such that they are self-contained and can be cleanly removed from the original specifications and reused somewhere else. A

**Acquire
a library
of useful
metaphors for
describing
a system.**

**Use your
available tools
well and
productively to
successfully
apply formal
specifications
in a real
project.**

reusable component is rarely used as is; its formal specification should facilitate extensibility to enable its use in different circumstances. Be careful, however, not to overload and stretch a metaphor to suit a particular system—if a metaphor is not natural to the system, don't use it. Although reusability is desirable in specifications, realize that developing anything for reuse or extensibility requires extra effort and experience.

11. Be true to the spirit of the notation.

Each formal specification notation is based on a particular mathematical theory and usually serves to describe specific system aspects. Use it appropriately, therefore. Don't unnaturally stretch it to describe other aspects for which it is not suited. For example, use Z to specify the system's functional aspects, writing the Z specification in a declarative style focusing on what is needed rather than how to do it. Its logic-based formalism doesn't work well for writing specifications of dynamic system behavior. On the other hand, notations such as Esterel are inherently behavioral; because they deliberately do not provide sophisticated data description and manipulation facilities, they aren't useful for describing functional system aspects. Write specifications in Statecharts or Esterel in a reactive style, focusing on input and output signals and relationships between them. Exploit facilities such as abstraction, modularity, composition, and hierarchical decomposition to the maximum extent.

12. Review and test the specifications thoroughly; document the test cases.

Subject formal specification of any real-life industrial-scale system to rigorous technical and user reviews to make sure it correctly describes the expected system. The system prototype built from the formal specification can play an important part in letting users validate their expectations from the specifications. When using executable notations, be careful to keep any unnecessary operational bias out of the requirements.^{13–15} Design formal specification reviews to help eliminate over- and underspecifications and improve bad structure and documentation clarity while identifying the properties and their proofs. Prepare a good comprehensive checklist of problems for the reviewer to look for.

13. Document the specifications well and provide explanations.

Documentation is essential to the formal specification development process, serving purposes such as conveying understanding to users, defining a reference point for system requirements to be used in the rest of the SDLC, and recording requirements changes. Such documentation should include a written, natural-language explanation of the formal specifications' structure and details and should also include examples, arguments about system properties, and the like. Comments should provide explanations at particular points and include a glossary, diagrams, charts, tables, and an index, and summaries as required. You might need to evolve a standard for such documentation or adopt an existing standard such as DoD2167A. Define appropriate standards and control tools to maintain multiple versions of formal specifications and their associated documentation.

14. Effectively use available tools.

Use your available tools well and productively to successfully apply formal specifications in a real project. Tools can add great value and provide many effective services throughout the project. However, many tools, such as theorem provers, require specialized training and much effort and tenacity to achieve any significant gain. Other tools, such as code generators, might not offer satisfactory output. With a little programming effort, you might augment existing tools' capabilities; efforts spent adapting a tool to deliver effective services are generally well rewarded in the end.

15. State and prove (or argue about or demonstrate) all the specification's necessary properties.

Endeavor to state all desired system properties. You can usually independently obtain system properties from user expectations. Your challenge in building formal specifications then becomes identifying interesting system properties and demonstrating that the specifications meet them. Close interaction with users lets you identify all such system properties or laws; then you can argue that these properties are really consequences of the formal specifications. These arguments might be *rigorous*, involv-

ing mathematical proofs, or *semiformal*, involving appeals to human reasoning. Finding mathematical proofs can be hard, so you can also use the system prototype derived from the formal specifications to test the specifications for some of these properties. Prototypes usually help you test the requirements in typical use cases and also greatly help in establishing a user's confidence in them. Be sure to identify and demonstrate all (or most) system properties.

These pragmatic guidelines should help development teams efficiently apply formal specifications to real-life software projects. I offer these guidelines as suggestions rather than hard-coded solutions and hope they will, ideally, serve as a starting point for building your own set. ☞

Acknowledgments

I thank Mathai Joseph for his support. I also thank colleagues in TRDDC for discussions and feedback, the referees for their perceptive comments, and Manasee Palshikar for her tenacity, hopes, and vision.

References

1. J.A. Arthur, *Rapid Evolutionary Development: Requirements, Prototyping and Software Creation*, Wiley and Sons, New York, 1992.
2. B. Ramesh et al., "Requirements Traceability: Theory and Practice," *Annals of Software Eng.*, vol. 3, Sept. 1997, pp. 397–415.
3. K. Kinney, "Mathematical Notations in Formal Specifications: Too Difficult for Masses?" *IEEE Trans. Software Eng.*, vol. 22, no. 2, Feb. 1996, p. 158.
4. P.G. Larsen, J. Fitzgerald, and T. Brookes, "Applying Formal Specifications in Industry," *IEEE Software*, vol. 13, no. 5, May 1996, pp. 48–56.
5. M.D. Faser, K. Kumar, and V.K. Vaishnavi, "Strategies for Incorporating Formal Specifications in Software Development," *Comm. ACM*, vol. 37, no. 10, Oct. 1994, pp. 74–86.
6. S. Esterbrook et al., "Experiences Using Lightweight Formal Methods in Requirements Modeling," *IEEE Trans. Software Eng.*, vol. 24, no. 1, Jan. 1998, pp. 4–14.
7. J.P. Bowen and M.G. Hinchey, "Ten Commandments of

Formal Methods," *Computer*, vol. 28, no. 4, Apr. 1995, pp. 56–63.

8. A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, vol. 7, no. 9, Sept. 1990, pp. 11–19.
9. J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software*, vol. 12, no. 7, July 1995, pp. 11–19.
10. I. Hayes, *Specification Case Studies*, 2nd ed., Prentice-Hall, New York, 1995.
11. P. Zave and M. Jackson, "Where Do Operations Come From? A Multiparadigm Specification Technique," *IEEE Trans. Software Eng.*, vol. 22, no. 7, July 1996, pp. 508–528.
12. W.R. Cyre, "Capture, Integration, and Analysis of Digital Systems Requirements with Conceptual Graphs," *IEEE Trans. Knowledge and Data Eng.*, vol. 9, no. 1, Jan. 1991, pp. 8–23.
13. M.B. Ozcan, "Use of Executable Formal Specifications in User Validation," *Software Practice and Experience*, vol. 28, no. 3, Nov. 1998, pp. 1359–1385.
14. A. Gravell and P. Henderson, "Executing Formal Specifications Need Not be Harmful," *Software Eng. J.*, vol. 11, no. 2, Feb. 1996, pp. 104–110.
15. I. J. Hayes and C. B. Jones, "Specifications Are Not (Necessarily) Executable," *Software Eng. J.*, vol. 4, no. 6, 1989, pp. 330–338.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

Prototypes usually help you test the requirements in typical use cases and also greatly help in establishing a user's confidence in them.

About the Author



Girish Keshav Palshikar is a scientist at the Tata Research Development and Design Centre (TRDDC) in Pune, India, the R&D division of Tata Consultancy Services. His research interests include formal methods and artificial intelligence, and he has worked on numerous projects involving formal methods, including a safety management kernel in an automatic train operation system. He received an MSc in physics

from the Indian Institute of Technology, Mumbai (Bombay), and an MS in computer science and engineering from the Indian Institute of Technology, Chennai (Madras). Contact him at TRDDC, 54 Hadapsar Industrial Estate, Pune 411013, India; girishp@pune.tcs.co.in.