COVER FEATURE

What Can We Expect from Program Verification?

Michael Jackson
The Open University

Naturam expellas furca tamen usque recurret.
You may drive out nature with a pitchfork, but she will always find a way back.
Horace, *Epistles* 1.10.24

Program verification assumes a formal program specification. In software-intensive systems, such specifications must depend on formalization of the natural, nonformal problem world. This formalization is inevitably imperfect, and poses major difficulties of structure and reasoning. Appropriate verification tools can help address these difficulties and improve system reliability.

In 2003, Tony Hoare proposed a Grand Challenge to develop a verifying compiler.[1] The following year, Jim Woodcock's broader proposal for *dependable systems evolution* recommended developing, in addition to a verifying compiler, a repository of realistic examples of programs and program documentation that had been, or were intended to be, verified.[2] Examples mentioned included the Dutch Waterkering storm-surge barrier system, the Mondex smart-card system for financial transactions, and selected Web services.

The understanding of verification as mathematical proof that a program satisfies its formal specification extends back to Alan Turing in the middle part of the past century. Verification techniques could include both checking a given program against specifications contained in the text or provided in a separate document, and correctness by construction, in which a systematic and formal development procedure guarantees the developed program's correctness or facilitates its verification in some way. The research roadmap that one Grand Challenge project subcommittee recently produced illustrates the richness of the possible repertoire of verification techniques.[3] Reflecting this richness, the more general term *tools for program verification* has now replaced the term *verifying compiler*.

SOFTWARE SPECIFICATIONS

For the proposed Grand Challenge, a correct software product is one that conforms to a formal specification. Key questions concern a specification's subject matter and its scope. Edsger Dijkstra[4] viewed a specification as a logical firewall separating the *correctness concern*—whether the program satisfies its formal specification—from the *pleasantness concern*—whether the program that satisfies the specification is one we'd like to have. In a more earthy formulation, the separation is between building the program right and building the right program.

The reason for this separation is obvious. Designers have engineered computers to form a domain for program execution in which correct formal reasoning is fully reliable. However, in many cases, the developer must draw on knowledge of the specification subject matter because the necessary reasoning goes beyond purely programming concerns. If this subject matter is abstract and mathematical—for example, computing the convex hull of a set of points in 3-space—the required knowledge concerns relevant mathematical axioms and theorems. It's thus no less formal than the program itself and leaves the applicability of formal reasoning intact.

0018-9162/06/$20.00 © 2006 IEEE

By contrast, the question of whether developers are building the right program—for example, whether they need the convex hull or the smallest containing cube—threatens formality by introducing informal concerns that lie outside the world of mathematics. Even the choice of an input format for the set of points whose convex hull will be computed introduces an informal element: Human considerations arise for which there is no indisputably correct, provable answer.

Therefore, restricting the specifications against which programs are to be verified seems desirable. Specifications must have abstract and formal subject matter lying strictly within the computer's perimeter.

In a software-intensive system, neither the problem nor its subject matter is formal. The problem world—the natural environment in which the software will run—is a collection of specific physical phenomena, including human beings and their engineered and otherwise constructed artifacts. The problem's subject matter isn't the abstract axioms and theories we might adopt to understand and formalize the environment properties and to state and solve the problem. Rather, it's the specific real environment itself, in all its buzzing, blooming confusion.

Maintaining the possibility of formally verified correct software in such a system requires two distinct preparatory tasks. First, developers must formalize the system requirement—that is, the effects the software must bring about in the environment—along with relevant environment properties. Also, they must use the formalized requirement and environment properties to derive a formal specification of the desired computer behavior at its interface with the environment.

Satisfying the resulting formal specification is the criterion for program correctness. Deriving the specification is the engineers' responsibility. This task concerns only the environment and lies outside the firewall. The task inside the firewall concerns only the software and the formal specification, and is the computer scientists' responsibility. The fully formal nature of programming and program verification remains intact inside the firewall.

FROM REQUIREMENTS TO SPECIFICATIONS

The nature of software-intensive systems' requirements makes it difficult to apply this tidy separation of responsibilities. Often, developers might easily state a vague and general requirement, such as: "Close the storm-surge barrier only when dangerous tides and weather are expected in 11 hours' time (the time it takes for the barrier to open or close)," or "Provide a convenient service that lets subscribers make and receive telephone calls." However, unlike "Compute the convex hull," developers might find it impossible to formalize these requirements in detail.

The nature of software-intensive systems' requirements makes it difficult to apply a tidy separation of responsibilities.

Formalized requirements

In Herb Simon's terminology,[5] the requirements must be *satisficed* rather than formally *satisfied*. Evolution of real systems is often iterative. Successive putative solutions to a partly undefined problem must be proposed, examined, and tried out. This process might continue indefinitely through a succession of operational versions. In established engineering branches, this iteration takes place over years—more than 120, in the case of motor vehicles.

A formalized requirement is always incomplete. Its current form is implicitly defined positively at any time by some approved properties of the current solution and negatively by some of its deprecated properties. Software specifications derived from systems requirements unavoidably inherit this partial, tentative, and implicit nature.

Restricting the specification

Even if a developer could formally state an adequate system requirement, restricting the software specification to the computer's behavior at its interface with the environment presents another obstacle. Such a restricted specification would be unmanageably complex and humanly unintelligible.

Consider, for example, the problem of developing a control program for traffic lights at an intersection. For the simplest crossroads, designating the two pairs of lights as E-W and N-S might be enough. The road layout and the general traffic rules are obvious. The restricted formal program specification is immediately intelligible: E-W and N-S green phases of stipulated durations alternate; green never shows in both directions; every alternate phase is red in both directions for a stipulated time; and there's a stipulated protocol for switching each light between red and green.

Developers can express this formal specification very clearly in terms of the computer-environment interface. But consider a complex intersection with many roads meeting in an elaborate layout with several closely adjacent nodes; pedestrian crossings, signals, and request buttons; and vehicle sensors embedded in the roads.

Now developers can't grasp the specification of the necessary control-program behavior without careful descriptions of road layouts (including road widths and filtering on turns); the positions and properties of lights, crossings, and sensors; pedestrian and vehicle traffic density and traversal times; and the required scheme of safe and efficient pedestrian and vehicle flow through the intersection.

As Figure 1 shows, the "orderly safe traffic" system requirement isn't located at computer interfaces A1, A2, and A3 with the light units, crossing buttons, and road sensors. Instead, it's deep in the environment at B1 and

B2, with the pedestrians, vehicles, and drivers. Understanding and justifying the desired program behavior at A1, A2, and A3 must penetrate similarly deeply.

Such understanding relies on properties of parts of the world that are only indirectly connected to the computer, just as the justification of a program to compute the convex hull of a set of points relies on mathematical properties of Euclidean space. This location deep in the environment is typical of realistic software-intensive systems. A profusion of seemingly arbitrary complexities and anomalies and a paucity of the reliable, tersely expressible regularities that characterize abstract mathematical worlds are also typical.

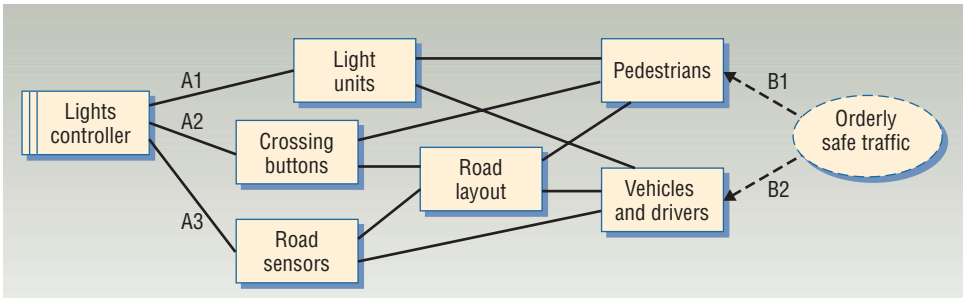

Figure 1. Orderly safe traffic. Requirements for a traffic system are located deep in the environment.

Physical world descriptions

Fortunately, restricting the specification to the computer interface isn't only impractical, it's also unnecessary. Although a physical and human environment is a nonformal domain, the engineering task of developing the system must rely—somewhat as in established branches of engineering—on formalized descriptions of the physical world and on reasoning about those descriptions. It's possible to bring these formalizations and the associated reasoning within the program specification's purview, and hence within the scope of some formal verification tools and techniques.

Four-variable model. The four-variable model[6] provides one approach to this goal. The computer is considered to be connected by sensors and actuators to the environment, through which it monitors and controls certain environment variables *M* and *C*. Relations *NAT* and *REQ* state the given environment properties and system requirements over those variables. Relations *IN* and *OUT* state the properties of sensors and actuators, which relate the environment variables to computer interface variables *I* and *O*. The development goal is to derive the relation *SOF*, which relates *I* and *O*.

Rely and guarantee. Another approach formalizes the problem-world properties in terms of *rely* and *guarantee* conditions.[7] Properties of each part of the world can be guaranteed provided that certain conditions—typically, some property of the behavior of another part—are reliable. Such an approach can extend as far as the program itself. Its specification is that it must guarantee to satisfy the overall system requirements while relying on certain properties and behaviors of the environment.

Reality variables. A third approach introduces problem-world properties into the program text by adding *reality variables* whose values represent problem-world states.[8] For program statements whose execution involves direct interaction with the problem world, developers write axioms characterizing the variables' states before, during, and after statement execution. In effect, this introduces the environment into the program as a set of specification or ghost variables.

Approaches of this kind can bring a variety of relevant formalizations within the scope of verification. In addition to program texts and specifications, developers can use formal descriptions of designs, development steps, given properties of the environment, system requirements, and relationships that should hold among them. All this is already envisioned in a proposal for the repository of documentation[2] and a discussion of research directions.[3]

Extending the range of verification inputs in this way, to include formalizations of the problem and its environment, considerably broadens the goal of verifying program correctness with respect to a program specification.

SOFTWARE-INTENSIVE SYSTEMS

Broadening the range of verification inputs provides more grist for the mill of formal tools and techniques. Since the added material is itself formal, resulting from formalization of its nonformal subject matter, developers can treat it the same way as formal program texts and specifications, using all the formal and mechanized reasoning tools. Effectively, developers use the tools to check the program specification's derivation from the system requirements.

This wouldn't merit extended consideration if it were the only effect of extending the Grand Challenge scope to software-intensive systems. But the effects are more substantial. They provide notable challenges and opportunities arising from the characteristics of software-intensive systems that distinguish them from systems concerned either with abstract mathematical worlds or the carefully formalized world of program execution—as in the case of cache management, operating systems, compilers, and file systems.

Of course, the distinction isn't rigorous. A compiler designer must consider how human users would understand and be helped by particular diagnostic messages,

and a file system must take explicit account of the possibility of disk error and failure. But the differences remain large, and must be explicitly addressed.

Three distinctive characteristics of software-intensive systems impinge most heavily on verification. First, understanding and analyzing the systems and problems they solve depends on particular forms of problem decomposition and patterns of the resulting components.

Second, the combination or composition of the components in a realistic system is heterogeneous. It presents the need for many particular forms of composition, which justify specific support in a verification toolset. More abstract forms of some of these combinations may already be well known in theory and practice.

Third, the nonformal nature of the underlying reality in the environment has an important effect on the role of formal reasoning about it, and suggests opportunities to help with the consequent difficulties.

DECOMPOSITION AND SUBPROBLEM COMPONENTS

Software-intensive systems exhibit complexity that must be mastered by decomposition. The absence of regularity in the problem environment and the requirements is one source of this complexity. For example, only weak generalizations can hold over the whole set of traffic light units in the complex intersection system because each unit occupies a unique position in the layout, with unique relationships to nearby lights and crossings.

The rich structure of a software-intensive system is a second source of complexity, in the sense that its functionality combines many large subfunctions of different kinds working together in many different ways. This richness and heterogeneity increase with more demands for multiple features and system interoperability.

Environment properties

We can regard specifying and implementing each subfunction as a subproblem within the overall problem of developing the system. Also, in a loose analogy with the structure of an engineering product such as a motor vehicle, we can regard the implemented subfunctions, together with their relevant parts of the environment, as components. Consider a control system intended to provide safe elevator service in a hotel. Developers must identify the environment properties the solution to the elevator service subproblem needs—for example, the causal chains that connect the motor's polarity and on-off switches to the car's movement in the shaft, and connect the car's position in the shaft to the states of the floor sensors. It's impossible to provide elevator service without relying on these properties.

Safety and other subproblems

But there's also a safety subproblem that requires designers to recognize the unreliability of these properties, however carefully they're chosen and formalized. The power might fail, the switch contact might fail, the motor might burn out, the cable might snap, a floor sensor might stick. There's no bound to the failure possibilities.

The safety subproblem requires a separate component. Environment properties that allow detection and diagnosis of equipment faults are important for this component. Running concurrently with elevator service, safety oversees equipment functioning and executes appropriate action when it detects a fault.

Another subproblem might provide a lobby display that shows which floor the elevator cars are on and how many floors each must visit before reaching the lobby. Another subproblem might let the hotel manager change elevator schedules to reflect new usage patterns. Yet another might provide a manual control regimen that an elevator engineer can use during maintenance. A realistic system would feature many such subproblems, each with its own requirement, its own relevant subset of the environment, and its own software specification.

Software-intensive systems exhibit complexity that must be mastered by decomposition.

Problem decomposition

Problem decomposition can have many goals, depending on the system and its context. Achieving simplicity in each subproblem, perhaps by applying a repertoire of design heuristics, is one important aim. One such heuristic restricts the subproblem requirement to a single level of desirability. In the elevator example, separating elevator service from safety avoids a requirement of the form: "Provide service, but if that's not possible, ensure safety."

The separation defers the combination of service and safety to another development task, but for the combining task the complexities of the separated subproblems are hidden. Another heuristic demands a consistent formalization of the environment. For example, the formalized environment properties needed for elevator service are different from those for fault detection and diagnosis.

A third heuristic might prohibit the presence of irreconcilable periodicities within the same subproblem. In a library management system, for example, this would lead to separating subscription management from book-borrowing management. Given suitable formalizations of requirements and environment properties, designers can use verification tools to check heuristics like these.

Known subproblem classes

Problem decomposition also aims to ensure that as many subproblems as possible fall into known classes

having known solutions. In software-intensive system problems as elsewhere in engineering, mastering heterogeneity and complexity depends heavily on accumulated specialized knowledge. In an illuminating book[9] about engineering, W.G. Vincenti distinguishes normal from radical design. According to Vincenti, in normal design, "the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."

In radical design, by contrast, "how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

The canons of normal design, established by long experience for each known device type, minimize the likelihood of unwelcome surprises, contributing hugely to dependability.

When the software execution begins, the relevant parts of the environment must be in a compatible state.

Device repertoire

Adapting Vincenti's view to software-intensive systems, we could regard the subproblem solutions—each with its software component, requirement, and relevant subset of the problem world—as devices. With increasing experience, we should be able to develop a repertoire of known device types.[10] Basic verification of each component individually wouldn't be very different in principle from verification of any program against a specification containing a formalization of the environment. But it might have additional aspects.

We expect normal component design to conform to a standard pattern of the software, its problem world, and their interactions. Such patterns could be explicitly named and described, just as object-oriented design patterns have been, and a component's conformity to its pattern could be checked as part of the verification process. For example, a pattern might stipulate that one particular part of the subproblem's environment be passive, in the sense that it never initiates events or state changes but only responds to external stimuli. Designers could check such a property against the pattern using a tool that analyzes the environment formalization.

Initialization concern

Each pattern is associated with several concerns that developers must address to avoid failures of known kinds. For example, the *initialization concern*, well known for program variables, is important in several classes of software-intensive system components. When the software execution begins, the relevant parts of the environment must be in a compatible state.

If developers specify the software so that any possible environment state is compatible, they've fully discharged the initialization concern. But if a system component relies on a stronger assumption about the environment state—for example, if the elevator service module assumes that the elevator is initially at a floor with the motor off—then even if the environment formalization asserts a compatible initial state, it's still appropriate to bring the matter to the developer's attention.

The initial environment state is that which holds when the subproblem software begins execution. It's easy to make assumptions about the initial state that aren't justified in the problem world reality.

COMPOSITION OF COMPONENTS

Vincenti pointed out that radical design is called into play not only for novel devices, but also more generally for systems, which are large heterogeneous assemblages of devices and other components and participants. Of course, we can also regard devices, even of known types, as small systems. But for a device of a known type, normal design specifically includes the composition of its parts. In a system, by contrast, even if all the component devices are of known types, the novelty of their composition imposes uncertainties. In addition to simple combinatorial explosion, uncertainties arise from novel and unanticipated forms of interaction.

Few problems with software-intensive systems fit comfortably into large, regular structures exhibiting architectural styles such as *procedure hierarchy*, *pipe and filter*, or *layers of abstraction*. In an interesting illustration—user control of a digital oscilloscope—researchers Mary Shaw and David Garlan described the difficulties of fitting the software design into one regular architectural style.[11]

System components' heterogeneity and their complex interactions make this task difficult. Designers can't understand component interactions solely in terms of their software parts' interactions within the program execution. Components interact not only at their software interfaces, but also indirectly by interacting with common parts of the problem world.

Component relationships in the environment

The elevator service and safety subproblems, for example, are related in a nontrivial way in the environment. They rely on different formalizations of the environment. They are concerned with overlapping but distinct subsets of the world. For example, only elevator service is concerned with the request buttons, and only safety is concerned with the emergency brake.

They have different control relationships to certain environment phenomena. Elevator service controls the

motor polarity, while safety only observes it. Their requirements might come into conflict. For example, in the event of a fault, safety requires the motor to shut down, while service might require it to run. Their requirements are related also by precedence, with safety taking precedence over service.

Component software implementations are ordered by criticality. The safety module must execute correctly with maximal reliability, and must not depend in any way on the service module. Some of these relationships demand verification of the paths between the software modules, and others require verification of the composition of their effects in the environment.

Formalizations of environment properties and system requirements are necessarily imperfect.

Types of composition

Some component compositions may be of well-known types such as the *interleaving* between the writer and readers of a shared environment part, or *switching* between subproblems handling different system modes, such as taxiing, takeoff, climbing, and cruising in an avionics system.

In a switching composition, control of an environment domain is handed over from one subproblem to another. The relinquishing subproblem must leave the domain in a state permitting handover, and the receiving subproblem must receive it in a suitable initial state or be able to put it into such a state.

In an interleaving composition, establishing atomicity and mutual exclusion in the software isn't enough. Designers must also examine the effects in the environment. For example, interleaving the subproblem in which the hotel manager edits the elevator-scheduling priorities with the elevator-service subproblem that's governed by those priorities demands more than mutually exclusive access to the priorities' data structure. Designers also must determine whether editing should always be permitted and when and how elevator service will change over to the newer priorities.

Established software verification forms deal with structural innovations of modern programming languages, such as encapsulation, inheritance, exceptions, and concurrency. Similarly, verification should deal with structural patterns evolved specifically for software-intensive systems. If an engineer could adopt a developed discipline of such patterns, verification tools could recognize and exploit them as readily as they'd recognize and exploit inheritance or exception-handling in a Java program.

REASONING ABOUT THE ENVIRONMENT

Formalizations of environment properties and system requirements are necessarily imperfect. First, definition and interpretation of formal terms is unavoidably fuzzy. Second, designers must approximate values of continuous phenomena. Third, there can be no frame conditions, since there's no limit to the phenomena or properties that might affect an assertion's truth or falsity. And fourth, effects that developers can ignore for each individual physical property might play a critical role in their composition.

Finding potential inadequacies

This doesn't affect pure logical reasoning on the basis of these formalizations. However, developers can't fully rely on the results of such reasoning reinterpreted in the environment. In reasoning about a physical world, logic can show only the presence of errors, never their absence. Formal verification tools can't examine the reality of the problem world to check the truth of their conclusions, but they might be able to indicate particular potential inadequacies.

For example, if part of the environment is formalized as having two distinct state components, each with its own protocol for external control, the verification tool might, by the rule of $\wedge$–introduction, deduce that any interleaving of the two protocols by execution of their corresponding software components will produce correspondingly interleaved state changes.

The verification tool might apply this reasoning to a machine tool with a longitudinal and a transverse motion. But in reality, some particular interleaving might cause the whole domain to reach an unanticipated and impermissible combined state—for example, one in which the power supply is overloaded because both motors are being started simultaneously under full load. In such a case, the tool could usefully point out that it has relied on a specific assumption of compositionality in computing the effects of program execution. It might even identify and enumerate the combined states that, according to some heuristic rule, will most likely be problematic. The developer would respond by checking that the state components are orthogonal, that none of the enumerated combined states is impermissible, and that the assumption of compositionality holds in reality.

Checking reasoning

If a program's specification and supporting documents include descriptions of formalized development steps, developers could deploy the power of verification to check the reasoning in those steps. One example is *problem reduction*[12] or *requirement progression*.[13] When a problem requirement is deep in the world—in the sense that it's separated from the computer by more than one problem-world domain, as it is in the complex traffic lights problem—developers can often start establishing a software specification by reasoning about the outermost domain to obtain a restated requirement expressed only in terms of domains closer to the computer.

A verification tool could check such reasoning steps for logical correctness. Drawing an analogy with a program-refinement step, or establishment of a lemma needed for a proof, is attractive.

The need to handle new programming-language features and constructs has long influenced development of verification tools and techniques. In the other direction, our desire for verification has influenced language development toward greater clarity and simplicity in programming.[3]

The history of types in programming languages illustrates the symbiosis between enhanced languages and verification methods very well. A similar symbiosis could exist in a wider context. Although verification (as opposed to testing) is concerned with the formal, it can address formalizations of requirements and problem worlds no less than formalizations of programs.

The availability of helpful verification tools could drive development of languages for capturing and refining the concepts of problem structure and analysis. Earlier work explored the concepts of problem structure and analysis mentioned here.[10,14,15] Interaction between such work and existing and future work on formal verification could be fruitful.

Much of what I've suggested is based on informal considerations, some implying judgments about the relative likelihood of different error classes in system development. This informality isn't alien to the spirit of the Grand Challenge. Intuition about human capacities is important, as it is for interactive theorem provers, and we shouldn't ignore it when applying verification to software-intensive systems.

Software verification is an attractive goal. Proponents of the Grand Challenge envision a world in which computer programs are always the most reliable component of any system or device that contains them. Verification tools and techniques can help make sure that the programmer builds the program right. They could also contribute to building the right program.

Acknowledgments

I am very grateful to Anthony Hall, Jon Hall, Daniel Jackson, Butler Lampson, Gary Leavens, Fred Schneider, and Michel Sintzoff, whose generous comments on an earlier version of this article helped clarify my thoughts. Responsibility for the remaining deficiencies is, of course, entirely mine.

References

1. T. Hoare, "The Verifying Compiler: A Grand Challenge for Computing Research," *J. ACM*, Jan. 2003, pp. 63-69.
2. J. Woodcock, "GC6: Dependable Systems Evolution," *Grand Challenges in Computing Research*, T. Hoare and R. Milner, eds., British CS, 2004, pp. 25-28.
3. G.T. Leavens et al., *Roadmap for Enhanced Languages and Methods to Aid Verification*, tech. report TR-06-21, Computer Science Dept., Iowa State Univ., July 2006; ftp://ftp.cs.iastate.edu/pub/techreports/TR06-21/TR.pdf.
4. E.W. Dijkstra, "On the Cruelty of Really Teaching Computing Science," *Comm. ACM*, Dec. 1989, pp. 1398-1404.
5. H. Simon, *Models of Bounded Rationality: Behavioral Economics and Business Organization*, MIT Press, 1982.
6. D.L. Parnas and Jan Madey, "Functional Documents for Computer Systems," *Science of Computer Programming*, Oct. 1995, pp. 41-61.
7. I.J. Hayes, M.A. Jackson, and C.B. Jones, "Determining the Specification of a Control System from That of Its Environment," *Proc. FME 2003*, Springer-Verlag, 2003, pp. 154-169.
8. K. Marzullo, F.B. Schneider, and N. Budhiraja, "Derivation of Sequential, Real-Time Process-Control Programs," *Foundations of Real-Time Computing: Formal Specifications and Methods*, A.M. van Tilborg and G. Koob, eds., Kluwer Academic Publishers, 1991, pp. 39-54.
9. W.G. Vincenti, *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*, The Johns Hopkins Univ. Press, 1993.
10. M. Jackson, "Problem Analysis and Structure," *Eng. Theories of Software Construction*, T. Hoare, M. Broy, and R. Steinbruggen, eds., *Proc. NATO Summer School Marktoberdorf*, IOS Press, 2000, pp. 3-20.
11. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996, pp. 39-42.
12. L. Rapanotti, J. Hall, and M. Jackson, "Problem Transformations in Solving the Package Router Control Problem," tech. report TR2006/07, Open University, July 2006.
13. R. Seater and D. Jackson, "Requirement Progression in Problem Frames Applied to a Proton Therapy System," *Proc. 14th IEEE Int'l Requirements Eng. Conf.* (RE 06), forthcoming, 2006.
14. M. Jackson and P. Zave, "Deriving Specifications from Requirements: An Example," *Proc. 17th Int'l Conf. Software Eng.* (ICSE 95), ACM and IEEE CS Press, 1995, pp. 15-24; http://doi.ieeecomputersociety.org/10.1109/ICSE.1995.10007.
15. J.G. Hall and L. Rapanotti, "A Reference Model for Requirements Engineering," *Proc. 11th Int'l Conf. Requirements Eng.* (RE 03), IEEE CS Press, 2003, pp. 181-187; http://doi.ieeecomputersociety.org/10.1109/ICRE.2003.1232749.

Michael Jackson *has visiting research posts in the Department of Computing at The Open University and at the Department of Computing Science at the University of Newcastle. His current research interests are problem analysis and structure and the relationship between problems and solutions. He received bachelor's degrees in classics and mathematics from Oxford and Cambridge universities. He is a Fellow of the Royal Academy of Engineering. Contact him at jacksonma@acm.org.*