IEEE Computer October 2006 pages 65-71

What Can We Expect from Program Verification?

Michael Jackson, The Open University
{jacksonma@acm.org}
Draft of 15 August 2006

Naturam expellas furca tamen usque recurret
You may drive out nature with a pitchfork, but she will always find a way back
Horace, Epistles I x 24

This note briefly discusses the relationship between program correctness and satisfaction of system requirements. The concept of program correctness assumes the existence of a formal program specification. In software-intensive systems such a specification may be hard to obtain and will unavoidably involve formalisation of the natural, non-formal problem world which can be checked by verification tools. Problem structure in such systems exhibits characteristic patterns that are not commonly found elsewhere—both patterns of individual components and patterns of their composition. These patterns affect the structure of the system development steps and documentation, including software and specification texts, and suggest potentially useful forms of verification and verification output. The inevitably imperfect formalisation of the non-formal problem world poses major difficulties, but here too appropriate verification tools can contribute to system reliability.

Introduction

In 2003 a Grand Challenge was proposed under the title *The Verifying Compiler* [Hoa03]. The following year a broader proposal, *Dependable Systems Evolution*, was made [Woo04] to develop a verifying compiler, along with a repository of realistic examples of programs and program documentation that had been, or were intended to be, verified. Examples mentioned included the system to control the Dutch Waterkering storm-surge barrier, the Mondex smart card system for financial transactions, and selected web services. Verification was understood in the sense of mathematical proof that a program satisfies its formal specification, an ideal whose history extends back at least to Alan Turing. Techniques could include both checking a given program against specifications embedded in the text or supplied in a separate document, and correctness by construction, in which a systematic and formal development procedure guarantees correctness of the developed program or in some way facilitates its verification. The richness of the possible repertoire of techniques is illustrated by the recent research roadmap [LeaAbr06] produced by one Grand Challenge project subcommittee. Reflecting this richness, the name v*erifying compiler* has recently been dropped in favour of the more general phrase *tools for program verification*.

Software Specifications

For the Grand Challenge, correctness of a software product is conformity to a formal specification. What is the subject matter and scope of a specification? One view is that a specification forms a ‘logical firewall’: it separates the ‘correctness concern’—whether the program satisfies its formal specification—from the ‘pleasantness concern’—whether a program satisfying that specification is one we would like to have. In a more earthy formulation, the separation is between ‘building the program right’ and ‘building the right program.’

The motive for this separation is self-evident. The computer has been carefully engineered so that in executing a program it constitutes a domain within which correct formal reasoning is fully reliable[1]. Whenever the problem subject matter is not entirely trivial, an additional aspect beyond purely programming concerns is unavoidably introduced because the developer must draw on knowledge of the subject matter. If this subject matter is abstract and mathematical—for example, a set of points in 3-space whose convex hull is to be computed—the required knowledge is knowledge of the relevant mathematical axioms and theorems: it is then no less formal than the program itself and leaves the applicability of formal reasoning intact. By contrast, the question whether we are building the right program—for example, whether we need the convex hull or the smallest containing cube—threatens to compromise 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 is to be computed introduces an informal element: human considerations arise for which there is no indisputably correct, provable, answer. It seems desirable, therefore, to restrict the specifications against which programs are to be developed and verified: their subject matter must be abstract and formal, and must lie strictly within the perimeter of the computer.

In a software-intensive system neither the problem nor its subject matter is formal. The problem world—that is, the particular natural, physical, environment[2] in which the software will run—is a collection of specific physical phenomena and things, including human beings and their engineered and other constructed artifacts. The problem's subject matter is not the abstract axioms and theories that we may adopt to understand and formalise the environment properties and to state and solve the problem: rather, it is the buzzing blooming confusion that is the specific real environment itself. To maintain the possibility of formally verified correct software in such a system, therefore, it seems necessary to take two distinct preparatory steps. First, the system requirement—that is, the effects that the software must bring about in the environment—must be formalised, along with relevant environment properties. Second, the formalised requirement and environment properties must be used to derive a formal specification of the desired computer behaviour at its interface with the environment. Satisfaction of the resulting formal specification is then the criterion of program correctness. The whole task of deriving the specification concerns only the environment and lies outside the firewall: it is the responsibility of engineers. The task inside the firewall concerns only the software and the formal specification, and is the responsibility of computer scientists. The fully formal nature of programming and program verification remains intact inside the firewall.

From Requirements to Specifications

One obstacle to applying this tidy separation of responsibilities to software-intensive systems is the nature of their requirements. Often a vague and general requirement may be easily stated: "Close the storm-surge barrier only when dangerous tides and weather are expected in 11 hours' time (the barrier takes 11 hours to open or close)," or "provide convenient telephone service for subscribers to make and receive calls." But, unlike the requirement "compute the convex hull," these requirements may be impossible to formalise in detail. In Simon's terminology, the requirements demand to be *satisficed* rather than formally *satisfied*. Evolution of real systems is often iterative: successive putative solutions to a partly undefined problem are proposed, examined, and tried out. This iterative process may continue indefinitely through a succession of operational versions[3]. A formalised

[1] For brevity and simplicity, we leave aside here the possibility of computer hardware failures.
[2] The terms *problem world* and *environment* are used interchangeably in this note.
[3] In established engineering branches this iteration takes place over many years. In the case of motor cars, over 120 years so far.

requirement is always very incomplete: its current form at any time is implicitly defined positively by some of the approved properties, and negatively by some of the deprecated properties of the current solution. This partial, tentative and implicit nature of the system requirements is unavoidably inherited by software specifications derived from them.

Even if an adequate system requirement could be formally stated, restricting the software specification to the behaviour of the computer 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 a road intersection. For the simplest crossroads it may be enough to designate the two pairs of lights as E-W and N-S: the road layout and the general traffic rule to be followed 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 is a stipulated protocol for switching each light between red and green. All this can be lucidly expressed in terms of the computer-environment interface[4]. But suppose instead that the intersection is complex, with many roads meeting in an elaborate layout having several closely adjacent nodes, with pedestrian crossings and pedestrian signals and request buttons, and vehicle sensors embedded in the roads. Now the specification of the necessary control program behaviour cannot be grasped without careful descriptions of the layout of roads (including road widths and the possibility of filtering on turns), of the positions and properties of lights and crossings and sensors, of pedestrian and vehicle traffic density and traversal times, and of the required scheme of safe and efficient pedestrian and vehicle flow through the intersection[5].

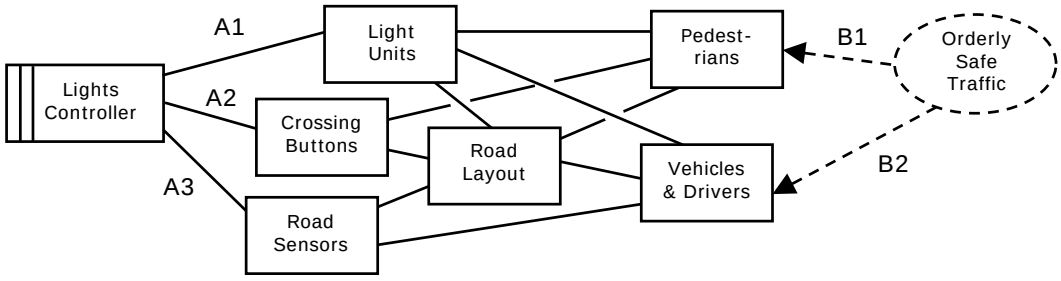

Figure 1: Requirements for a traffic system are deep into the environment

As Figure 1 shows, the system requirement of "Orderly Safe Traffic" is not located at the computer interfaces A1, A2 and A3 with the light units, crossing buttons, and road sensors, but at B1 and B2, deep in the environment with the pedestrians and the vehicles and drivers. Understanding and justification of the desired program behaviour at A1, A2 and A3 must penetrate similarly deeply. It relies on properties of parts of the world that are connected only indirectly to the computer, exactly as the justification of a program to compute the convex hull of a set of points relies on mathematical properties of Euclidean space. This depth in the environment is typical of realistic software-intensive systems. So too are the profusion in the environment of seemingly arbitrary complexities and anomalies, and a paucity of the reliable, tersely expressible, regularities that characterise abstract mathematical worlds.

[4] We are assuming that the relationship between the states of the computer's output ports and those of the traffic lights is trivially obvious. It may not be.

[5] It may be pointed out that the restricted specification can be made intelligible by the use of appropriate abstractions. But these abstractions will be effective only if they constitute—as suggested in later paragraphs of this section—the very formalisations of the non-formal environment properties and requirement that the restriction is intended to exclude.

Fortunately, restricting the specification to the computer interface is not only impracticable: it is also unnecessary. Although a physical and human environment is a non-formal domain, the engineering task of developing the system must rely to a considerable extent—somewhat as in the established branches of engineering—on formalised descriptions of the physical world and on reasoning about those descriptions. It is possible to bring these formalisations and the associated reasoning within the purview of the program specification, and hence to a considerable extent within the scope of some formal verification tools and techniques.

One approach to this goal is the Four Variable Model [ParMad95]. The computer is considered to be connected to the environment by sensors and actuators through which it monitors and controls certain environment variables *M* and *C*. The given environment properties, and the system requirement, are stated in relations NAT and REQ over those variables; the properties of sensors and actuators are stated in relations IN and OUT which relate the environment variables to computer interface variables *I* and *O*; the goal of development is to derive the relation SOF, which relates *I* and *O*.

Another approach, described in [HayJacJon03], formalises the problem world properties in terms of *rely and guarantee* conditions. Properties of each part of the world can be guaranteed provided that certain conditions—typically, some property of the behaviour of another part—can be relied on. Such an approach can be extended 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 behaviours of the environment.

A third approach, described in [MarSchBud91], introduces problem world properties into the program text itself by adding *reality variables* whose values represent problem world states. For program statements whose execution involves direct interaction with the problem world, axioms are written characterising the states of these variables before, during and after statement execution. In effect, the environment is introduced into the program as a set of specification or ghost variables.

Approaches of this kind can bring a variety of relevant formalisations within the scope of verification. In addition to program texts and program specifications there can be formal descriptions of designs, of development steps, of given properties of the environment, of system requirements, and of relationships that should hold among them. All this is already envisioned—for example, in the proposal for the repository of documentation [Woo04] and in the discussion of research directions in [LeaAbr06]. The goal of verifying program correctness with respect to a program specification has already been considerably broadened, at least by broadening the range of verification inputs to include formalisations of the problem and its environment.

Software-Intensive Systems

Broadening the range of verification inputs provides more grist to the mill of formal tools and techniques. Since the added material is itself formal, resulting specifically from formalisation of its non-formal subject matter, it can be treated in the same way as formal program texts and specifications, using all the formal and mechanised reasoning tools that can be brought to bear. Effectively, the tools can be used to check the derivation of the program specification from the system requirements.

If this were the only effect of extending the Grand Challenge scope to software-intensive systems, it would not merit extended consideration. But the effects are more substantial, and provide challenges and opportunities that should not escape attention. These challenges and opportunities arise from the characteristics of software-intensive systems that distinguish

them from systems concerned either with abstract mathematical worlds or with the carefully formalised world of program execution—as in the case of cache management, operating systems, compilers, and file systems. Of course the distinction is not rigorous. A compiler designer must consider how helpful and understandable particular diagnostic messages would be to the human user, and a file system must take explicit account of the possibility of disk error and failure. But the differences remain large, and demand to be explicitly addressed.

Three distinctive characteristics of software-intensive systems impinge most heavily on verification. First, effective understanding and analysis of the systems, and of the problems that they solve, depend on particular forms of problem decomposition and patterns of the resulting components. Second, 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 tool set. More abstract forms of some of these combinations may be already well known both in theory and practice. Third, the non-formal 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.

We discuss these characteristics and mention their possible impacts on verification in the immediately following sections.

Decomposition and Subproblem Components

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

A second source of complexity is that a software-intensive system is typically richly structured, 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 increasing demands for multiple features and for interoperability of systems. Specifying and implementing each subfunction can be considered as a subproblem within the overall problem of developing the system, and, in a loose analogy with the structure of an engineering product such as a motor car, the implemented subfunctions, together with their relevant parts of the environment, can be considered to be components.

Consider a control system intended to provide safe lift service between floors in a hotel. The developer must identify the environment properties on which the solution to the lift service subproblem will rely—for example, the causal chains that connect the motor's polarity and on-off switches to the movement of the lift car in the shaft, and connect the car's position in the shaft to the states of the floor sensors: it is impossible to provide lift service without relying on these properties. But there is also a safety subproblem, requiring a recognition that these properties, however carefully chosen and formalised, are not fully reliable. The power may fail; the switch contact may fail; the motor may burn out; the cable may snap; a floor sensor may stick; there is no bound to the possibilities of failure. The safety subproblem requires a separate component, for which the important environment properties are those that allow detection and diagnosis of faults in the equipment. Running concurrently with lift service, safety monitors equipment functioning and executes appropriate action when it detects a fault. Another subproblem in the system might provide the display in the hotel lobby that shows impatient guests where the lift cars are, and how many floors each must visit before it reaches the lobby. Another might provide the facility

for the hotel manager to change the lift scheduling priorities to reflect changing patterns of usage. Yet another might provide manual control regimes to be used by the lift engineers during maintenance of the equipment. In a realistic system there would be many such subproblems, each with its own requirement, its own relevant subset of the environment, and its own software specification.

Problem decomposition can have many aims, according to the particular system and its context. One important aim is to achieve simplicity in each subproblem, perhaps by applying a repertoire of design heuristics. One such heuristic restricts the subproblem requirement to a single level of desirability. In the lift example the separation of lift 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 formalisation of the environment. Again in the lift example the environment properties necessary for lift service are different from those for fault detection and diagnosis. A third heuristic may prohibit the presence of irreconcilable periodicities within the same subproblem. In a library management system, for example, this would lead to separating the management of subscriptions from the management of book borrowing. Given suitable formalisations of requirements and environment properties, heuristics like these could be checked by verification tools, and the results presented to the developer's judgment.

A second aim of problem decomposition is to ensure that as many subproblems as possible fall into known classes having known solutions. Mastering heterogeneity and complexity, in software-intensive system problems as elsewhere in engineering, depends heavily on accumulated specialised knowledge. In an illuminating book [Vin93] about engineering, W G Vincenti distinguishes normal from radical design. 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, minimise the likelihood of unwelcome surprises, contributing hugely to dependability.

Adapting Vincenti's view to software-intensive systems, we may regard the subproblem solutions—each with its software component, its requirement, and its relevant subset of the problem world—as devices. With increasing experience it should be possible to develop a repertoire of known device types [Jac00]. The basic verification of each component individually would not be very different in principle from the verification of any program against a specification that contains a formalisation of the environment. But it might have additional aspects.

Normal design of a component is expected to conform to a standard pattern of the software itself, its problem world, and their interactions. Such patterns can be explicitly named and described, just as object-oriented design patterns have been, and a component's conformity to its pattern might be checked as a part of the verification process. For example, a pattern might stipulate that one particular part of the subproblem's environment must be

passive, in the sense that it never initiates events or state changes but only responds to external stimuli. Such a property could be checked against the pattern by analysis of the environment formalisation.

Further, each pattern is associated with a number of concerns that must be addressed to avoid failures of known kinds. For example, the initialisation concern, well known for program variables, is important in several classes of software-intensive system component: when the software execution begins, the relevant parts of the environment must be in, or be brought into, a compatible state. If the software is specified so that any possible state of the environment is compatible, the initialisation concern has been fully discharged. But if a stronger assumption is made about the environment state—for example, an assumption by the lift service module that the lift is initially at a floor with the motor off—then even if the environment formalisation asserts a compatible initial state it is 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 is easy to make assumptions about it that are not justified in the reality of the problem world.

Composition of Components

Vincenti points 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, devices, even of known types, can also be regarded 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 there are uncertainties arising from novel and unanticipated forms of interaction.

Few software-intensive system problems fit comfortably into a large regular structure characterised by an architectural style such as procedure hierarchy, pipe-and-filter, or layers of abstraction. An interesting illustration—the user control of a digital oscilloscope—is discussed in [ShaGar96], where the authors describe the difficulties of fitting the software design into any one of several regular architectural styles. One reason is the heterogeneity of the system components, and the inherent complexity of their interactions. Component interactions cannot be understood solely in terms of the interactions of their software parts within the program execution: components interact not only at their software interfaces, but also indirectly by their interactions with the same or different aspects of common parts of the problem world.

The lift service and safety subproblems, for example, are related in a non-trivial way. They rely on different formalisations of the environment. They are concerned with overlapping but distinct subsets of the world: only lift service is concerned with the request buttons; only safety is concerned with the emergency brake. They have different control relationships to certain phenomena of the environment: lift service controls the motor polarity, while safety only observes it. Their requirements may come into conflict: in the presence of a fault, safety requires the motor to be shut down, while service may require it to run. Their requirements are related also by precedence: safety must take precedence over service. Their software implementations are ordered by criticality: correct execution of the safety module must be maximally reliable and must not depend in any way on the service module. Some of these relationships demand verification of the paths between the software modules, others of the composition of their effects in the environment.

Some component compositions may be of types well known to computer science, 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, take-off, climbing and cruising. In a switching composition the 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 either receive it in a suitable initial state or be able to put it into such a state. In an interleaving composition it is not enough to establish atomicity and mutual exclusion in the software. It is necessary also to examine the effects in the environment. The interleaving of the subproblem in which the hotel manager edits the lift scheduling priorities with the lift service subproblem that is governed by those priorities demands more than mutually exclusive access to the priorities data structure. It is also necessary to determine whether editing is always permitted and when and how lift service is to change over from the older to the newer priorities.

Just as established forms of software verification deal with the structural innovations of modern programming languages, such as encapsulation, inheritance, exceptions and concurrency, so in the same way it is desirable to deal with structural patterns evolved specifically for software-intensive systems. If a developed discipline of such composition patterns is achieved, verification tools could recognise and exploit them as readily as they would now recognise and exploit inheritance or exception handling in a Java program.

Reasoning about the Environment

The formalisations of environment properties and system requirements are necessarily imperfect. First, because formal terms will be unavoidably fuzzy in their definition and interpretation. Second, because values of continuous phenomena must be approximated. Third, because there are no frame conditions: the natural world allows no bound to the phenomena or properties that may prove relevant to the truth or falsity of an assertion. Fourth, because physical properties are not in general compositional: effects that can be properly ignored for each property individually may play a critical role in their composition.

Pure logical reasoning on the basis of these formalisations is unaffected, but the results of such reasoning, reinterpreted in the environment, are not fully reliable. In reasoning about a physical world, logic can show only the presence of errors, never their absence. Formal verification tools cannot examine the reality of the problem world to check the truth of their conclusions, but they may be able to indicate particular potential inadequacies. For example, if a part of the environment is formalised as having two distinct state components, each with its own protocol for external control, the verification tool may, by the rule of $\wedge$–introduction, deduce that any interleaving of the two protocols by interleaved execution of their corresponding software components will produce correspondingly interleaved state changes. This reasoning might be applied to a machine tool with a longitudinal and a transverse motion. But in reality some particular interleaving may 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 it could be useful for the verification tool to point out that it has relied on a specific assumption of compositionality in computing the effects of program execution. It might even be possible to identify and enumerate the combined states that, according to some heuristic rule, are most likely to be problematic. The developer would respond by checking that the state components are indeed orthogonal, that none of the enumerated combined states is impermissible, and that the assumption of compositionality holds well enough in reality.

If a program's specification and supporting documents include descriptions of formalised development steps in reasoning about the problem world, the power of verification could be deployed to check the reasoning in those steps. One example is what is called *problem*

reduction in [RapHalJac06] and *requirement progression* in [SeaJac06]. When a problem requirement is deep in the world, in the sense that it is separated from the computer by more than one problem world domain, a step towards developing a software specification can often be made by reasoning about the outermost domain to obtain a restated requirement expressed only in terms of domains closer to the computer. Such a reasoning step could be checked for logical correctness by a verification tool. Analogy with a program refinement step, or even the establishment of a lemma needed for a proof, is attractive.

Closing Comments

Development of verification tools and techniques has long been influenced by the need to handle new programming language features and constructs. In the other direction, the desire for verification has influenced language development towards greater clarity and simplicity in programming. The subcommittee report mentioned earlier [LeaAbr06] is entitled *Roadmap for Enhanced Languages and Methods to Aid Verification*. The history of types in programming languages illustrates this symbiosis very well.

A similar symbiosis could exist in a wider context. Although verification (as opposed to testing) is necessarily concerned with the formal, it can address itself to formalisations of requirements and problem worlds no less than to formalisations of programs. The potential availability of relevant help from verification tools could give impetus to further development of languages for capturing concepts of problem structure and analysis, and to refining and improving those concepts. The concepts of problem structure and analysis mentioned here have been explored in earlier work and discussed, for example, in [JacZav95, Jac00, HallRap03]. Interaction between such work and existing and future work on formal verification could be fruitful.

Much of what is suggested here is based on very informal considerations, some implying judgments about the relative likelihood of different error classes in system development. This informality is not alien to the spirit of the Grand Challenge. Intuition about human capacities is important, as it is for interactive theorem provers, and should not be ignored when applying verification to software-intensive systems.

The software verification goal is self-evidently attractive. Proponents of the Grand Challenge have written: “We 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 contribute to ensuring that the programmer builds the program right: they could also contribute to building the right program.

Acknowledgements

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

References

[HalRap03] Jon G Hall and Lucia Rapanotti; *A Reference Model for Requirements Engineering*; in Proceedings of the 11th Joint International Conference of Requirements Engineering (RE’03), 2003.

[HayJacJon03] Ian J Hayes, Michael A Jackson and Cliff B Jones; *Determining the specification of a control system from that of its environment*; in Keijiro Araki, Stefani

Gnesi and Dino Mandrioli eds, Formal Methods: Proceedings of FME2003, pages 154-169, Springer Verlag, Lecture Notes in Computer Science 2805, 2003.

[Hoa03] Tony Hoare; *The Verifying Compiler: A Grand Challenge for Computing Research*; Journal of the ACM Volume 50 Number 1, pages 63-69.

[Jac00] Michael Jackson; *Problem Analysis and Structure*; in *Engineering Theories of Software Construction*, Tony Hoare, Manfred Broy and Ralf Steinbruggen eds; Proceedings of NATO Summer School, Marktoberdorf; IOS Press, Amsterdam, Netherlands, August 2000.

[JacZav95] Michael Jackson and Pamela Zave; *Deriving Specifications from Requirements: An Example*; in Proceedings of the 17th International Conference On Software Engineering, pages 15-24, ACM and IEEE CS Press, 1995.

[LeaAbr06] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump; *Roadmap for Enhanced Languages and Methods to Aid Verification*; Iowa State University TR #06-21, July 2006.

[MarSchBud91] Keith Marzullo, Fred B Schneider and Navin Budhiraja; *Derivation of Sequential, Real-Time Process-Control Programs.* In Foundations of Real-Time Computing: Formal Specifications and Methods, A M van Tilborg and G Koob, eds, Kluwer Academic Publishers, 1991, pages 39-54.

[ParMad95] David Lorge Parnas and Jan Madey; *Functional Documents for Computer Systems*; Science of Computer Programming Volume 25 Number 1, pages 41-61, October 1995.

[RapHalJac06] Lucia Rapanotti, Jon Hall, Michael Jackson; *Problem Transformations in Solving the Package Router Control problem*; Open University Technical Report No 2006/07, 5th July 2006.

[SeaJac06] Robert Seater and Daniel Jackson; *Requirement Progression in Problem Frames Applied to a Proton Therapy System*; Proceedings of the 14th International Requirements Engineering Conference (RE06), Minneapolis USA, 2006.

[ShaGar96] Mary Shaw and David Garlan; *Software Architecture: Perspectives on an Emerging Discipline*; Prentice-Hall 1996, pages 39-42.

[Vin93] Walter G Vincenti; *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History;* The Johns Hopkins University Press, Baltimore, paperback edition, 1993.

[Woo04] Jim Woodcock; *GC6: Dependable Systems Evolution*; in Tony Hoare and Robin Milner eds; *Grand Challenges in Computing Research*; BCS 2004, pp25-28.