15th IEEE International Requirements Engineering Conference

RE Theory Meets Software Practice: Lessons from the Software Development Trenches[*]

Constance Heitmeyer Ralph Jeffords Ramesh Bharadwaj Myla Archer
Naval Research Laboratory, Washington, DC 20375
{heitmeyer, jeffords, ramesh, archer}@itd.nrl.navy.mil

Abstract

Based on our recent experience in four projects, each focused on either security-critical or safety-critical software, this paper evaluates several notions, widely held by RE researchers, for their utility in practical software development. It describes four notions which in our view work in practice and five others which do not.

1. Introduction

During the past three years, our research group has participated in four software projects, each involving either security-critical or safety-critical software. In the first project, we provided evidence in the form of informal and formal specifications, machine-supported proofs, and code-to-specification conformance mappings for the certification of an embedded software system [11]. In three other projects, we developed requirements specifications of safety-critical software modules of a NASA system [8]. In all four projects, the software requirements were specified in the SCR (Software Cost Reduction) tabular notation [9]. Based on our experience, this paper describes both the strengths and weaknesses of some notions advocated by many RE researchers. Section 2 presents four notions for developing requirements which we believe should be applied in practice, and Section 3 describes five other notions which, in our view, do not work in practice and how to remedy them. Section 4 presents some concluding remarks.

This paper discusses both axiomatic and operational specifications. An *axiomatic specification* describes the requirements as a set of properties, each property expressed as a formula in some logic. Goal-based methods, such as [2], typically use an axiomatic specification to represent requirements. In contrast, an *operational* or *model-based specification* describes how the system is required to operate. A state machine model, such as the model that underlies the SCR notation, is one approach to constructing an operational specification of the required software behavior.

2. RE Notions To Adopt and Apply in Practice

In our view, four notions advocated by RE researchers should be adopted and applied in practice. Applying them will in our view lead to major improvements in software quality and significant reductions in development costs. These notions are: 1) formulate axiomatic specifications, 2) state requirements in terms of environmental quantities, 3) specify requirements in a language with a formal semantics, and 4) avoid implementation bias. Although most developers are familiar with these notions, they seldom formulate requirements that adhere to them. In large part, this is because developers do not have languages and tools to help them apply the notions. To express requirements, most developers use either natural language, or languages such as Statecharts and Simulink not designed to represent requirements, making it hard to produce requirements specifications consistent with the four notions.

2.1. Value of Axiomatic Specifications

Many RE researchers advocate axiomatic specifications of software requirements. An example of an axiomatic approach is the goal-oriented KAOS method for eliciting, representing, and reasoning about requirements [2]. As an example of a property in an axiomatic specification, the following is a natural language excerpt from an axiomatic specification of a Therapy Control System [19]:

- **Overdose:** At no time should the radiation received by $\cdots$ the patient's body exceed the dose $\cdots$ in the treatment plan.

Axiomatic specifications have several advantages. First, critical system properties such as safety and security properties are expressed most naturally as axioms. In addition, stating the required system behavior as a set of properties is often an effective way to communicate with a customer to elicit and validate requirements.

2.2. Central Role of the Environment

Many researchers, including Parnas [17] and Zave and Jackson [21], emphasize the importance of representing the required software system behavior in terms of environmental quantities and explicitly stating the constraints that natural laws impose on the required behavior. One major advantage of expressing requirements in terms of the environment is that the requirements become more understandable.

In Parnas' model [17], the required system behavior is described as a relation on *monitored* and *controlled variables*, which represent environmental quantities the system

[*]This research is supported by ONR and NASA.

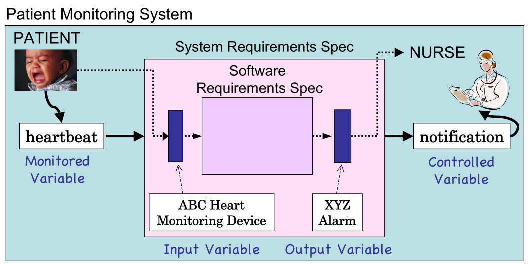

Figure 1. System and Software Requirements

monitors and controls. The model includes two relations, `REQ` and `NAT`, each defined on the monitored and controlled variables. `NAT` specifies the constraints imposed on the system by natural laws; `REQ` specifies the required system behavior as a relation on the monitored and controlled variables. In Parnas' model and in our approach [1, 7], *input* and *output variables* are used to model the relationship between the values read by sensors (written to actuators) and the values of monitored (controlled) variables. Zave and Jackson refer to "environment-controlled" and "software system-controlled" phenomena which correspond directly to Parnas' monitored and controlled variables. Further, among the environment-controlled phenomena, they distinguish phenomena shared with—and directly visible to—the software system (e.g., values read by sensors) from phenomena separate from and hidden from the system.

In the Patient Monitoring System [5] shown in Figure 1, the patient's heartbeat is a monitored quantity, and the "notification" sent to the nurse is a controlled quantity. In [21], the notification is considered to be software system-controlled and the patient's heartbeat to be environment-controlled. Both are hidden from the software system; for example, the heartbeat requires some indirect method of measurement, e.g., detection of the heartbeat sound. Identifying the appropriate monitored and controlled variables is often non-trivial. For example, while the critical property of the Therapy Control System described in Section 2.1 is stated in terms of environmental quantities (dose, patient's body, etc.), determining the monitored and controlled quantities based on this critical property can be difficult.

2.3. Need a Formal Requirements Language

Formulating requirements in a language with an explicit formal semantics reduces ambiguity and imprecision in specifying the required behavior. Further, specifications in the language can be mechanically checked for properties of interest, such as consistency and completeness, and critical application properties, such as safety properties.

2.4. Need to Avoid Implementation Bias

Both researchers and developers agree on the need to avoid implementation bias. Bias in the requirements specification not only rules out acceptable implementations but also makes more difficult 1) understanding the requirements and 2) designing and implementing the software based on the requirements. These latter difficulties arise because developers need to wade through a larger than necessary specification to separate the actual requirements from the design details that clutter the specification [12].

A language specifically designed to capture requirements makes the inclusion of implementation bias difficult.[1] It also provides the constructs needed to specify requirements, such as monitored and controlled variables. Thus, requirements languages, such as RSML [6] and SCR [9], are preferable to more general-purpose languages, such as Statecharts, Stateflow, and Simulink. Given too much freedom in specifying the required system behavior, specifiers may find it hard to avoid implementation bias.

3. Impractical RE Notions and Their Cure

This section describes shortcomings in five RE notions and remedies for each.

3.1. Insufficiency of Axioms (i.e., Goals)

Our experience is that axiomatic specifications alone are insufficient. In [14], Lamport shares this view, stating:

> Knowing what doesn't work is as important as knowing what does· · ·The lesson I learned· · ·is that axiomatic specifications don't work. The idea of specifying a system by writing down all the properties it satisfies seems perfect. We just list what the system must and must not do, and we have a completely abstract specification. It sounds wonderful; it just doesn't work in practice.

Although axiomatic specifications can play an important role in specifying requirements, model-based specifications have several advantages over axiomatic specifications. A model-based specification can be analyzed mechanically to detect inconsistent behavior and to determine whether the specification of the required behavior is complete (e.g., no missing cases) [9, 6]. If it is executable, an operational specification can be symbolically executed by domain experts to validate it. Moreover, a state machine model provides a natural basis for verifying additional properties of the specification—for example, invariant properties can be proved by induction over the reachable states.

Opponents of operational specifications claim that such specifications encourage implementation bias by including extra variables, e.g., *internal* or *auxiliary* variables; examples are the modes and terms in SCR specifications. For example, the authors of [21] point out the dangers of an explicit machine state in specifications:

> [S]pecifying a machine in terms of its states appears to introduce serious implementation bias, because its states are internal and not directly observable at the interface between the machine and its environment.

[1]For an example, see [8], which describes how translating NASA requirements into SCR exposed implementation bias.

Lamport [14] counters this argument by describing an internal variable as "one that appears in a specification but is not meant to be implemented."

Whether the use of explicit states and internal variables in a requirements specification leads to implementation bias depends on how one establishes that a given "concrete" machine implements a specified machine. Many specification methods defining an explicit machine state avoid implementation bias by defining an implementation to be a concrete machine with the same observable behavior as the specified machine. Two specification methods that represent complex values of state variables concretely are SCR [9] and TIOA [3]. In these methods, a concrete machine is shown to be an implementation—i.e., exhibits the "same observable behavior"—by establishing either a homomorphism or, more generally, a simulation relation between machines.

Remedy: Our view is that both an axiomatic specification (restricted to the critical system properties) and an operational specification should be developed, thus obtaining the advantages of both. The construction of operational specifications by software developers is clearly feasible; for examples of such specifications produced by Navy and NASA contractors, see [10, 8]. Further, many NASA requirements documents contain "shall statements," natural language statements of axioms or goals [8]. One major advantage of building two specifications at different abstraction levels (but with the same vocabulary) is that finding inconsistencies between them can detect errors in one or both.

3.2. Deriving Specifications Is Impractical

Some researchers have shown how an axiomatic specification may be transformed into an operational one. For example, Pavlovic and Smith provide a category theory framework and tools for semi-automated refinement of axiomatic specifications not only to operational specifications but for refinement to code [18]. As another example, Letier and van Lamsweerde present rules for transforming goal specifications into specifications of software operations [15].

While systematically deriving an operational specification from a set of properties is sometimes feasible (as demonstrated by [18, 15]), like Lamport, we doubt that this approach scales to practical systems. Moreover, in many cases, deriving the operational specification from properties is infeasible; for example, in the security-critical system described in [11], the distance between the five abstract security properties and the module operations was enormous, thus making such a derivation impossible.

Remedy: Our view is that developers should build an operational specification directly rather than formally deriving it from an axiomatic specification. Our experience is that developing an operational specification and a set of properties independently is much more natural and cost-effective than deriving a more concrete specification from a more abstract specification.

3.3. Inadequacy of RE Terminology

Many RE researchers (e.g., [21, 19]) and some textbooks, e.g., [4], use the term 'requirements' to represent an abstract axiomatic specification that is not "implementable" and 'specification' to refer to a precise description that is "implementable". For example, in the Therapy Control System [19], the 'requirements' are in terms of the patient's dosage and excess radiation, abstract notions that are not implementable, while the 'specifications' are in terms of dose units and radiation bursts, concrete quantities that are implementable. In this view, the term 'specification' captures all the information needed to implement the software. This information includes the attributes of I/O devices which measure the values of monitored variables and set the values of controlled variables.

Remedy: Like Parnas [17], we have a broader view of the term 'specification'. We construct and reason about "requirements specifications" (as opposed to, e.g., design and program specifications) and distinguish two different requirements specifications, both operational: the *System Requirements Specification* (SysRS), which describes REQ and NAT, and the *Software Requirements Specification* (SofRS), which describes how values read from (written to) I/O devices are used to estimate (set) the values of monitored (controlled) variables [1, 7, 16].

3.4. Including I/O Devices in Specifications

To obtain an implementable 'specification,' Zave and Jackson refine the original, unimplementable 'requirements' by replacing all "hidden" environmental phenomena by implementable phenomena (i.e., those "visible" to the software system).

In the Patient Monitoring System in Figure 1, a specific heart monitoring device must be selected to check the patient's heartbeat and another device selected to notify the nurse. In [5], a microphone is chosen as a sensor for the software-system-visible environmental variable "heart sound," and a "buzzer" for the software-system-visible environmental variable. The problem is that the 'specification' is now at a less abstract level—strictly in terms of these system visible variables. The requirement has been transformed into the specification "If the heart sound falls below some threshold, then sound the buzzer." Thus the decisions regarding sensors and actuators have been integrated into the implementable specification, preventing easy change of sensors and actuators.

Remedy: As Parnas states [17]:

> Usually, one obtains the clearest and simplest documents [i.e., requirements, not necessarily implementable yet] by writing them in terms of the variables of interest to the user [e.g., heartbeat] in spite of the fact that the system will monitor other variables [e.g., heart sound] in order to determine the value of those mentioned in the document [e.g., heartbeat].

In Parnas' and related approaches [17, 1, 7, 16], three specifications are written. The first, the SysRS, contains no information about I/O devices. The second, the *System Design Specification* (SDS), contains descriptions of the selected I/O devices and the rationale for selecting a particular set of devices. The third, the SofRS, contains sections listing the attributes of the I/O devices needed to construct an implementation (see [13, 16] for examples) and specifying how the software uses values read from input devices (input variables) to estimate monitored variable values and sets values of the output devices (output variables) to change the controlled variable values. Thus, the SofRS contains information the developers need to build the device drivers [16].

The major reason for excluding information about I/O devices from the SysRS is to facilitate change. For example, if one decides to switch from a microphone placed on the patient's chest to an automated sphygmomanometer to detect heartbeats, the SysRS need not change. Only sections in the SDS and the SofRS which describe attributes of the device that measures heartbeat will change.

3.5. RE Neglect of Module Specifications

Most RE researchers focus on software systems embedded in a physical environment. Yet, in software practice, there is usually insufficient time and funding to specify and analyze the requirements of an entire software system.

Remedy: Identify software modules whose correctness is crucial to the security or safety of the software system, and specify the required externally visible behavior of only these modules. While this might be considered software design (rather than requirements specification), our experience is that the same principles apply in specifying the required externally visible behavior of a module as apply in specifying a software system embedded in a physical environment. In the security-critical project in which we participated [11], the focus of interest was a separation kernel. In two of the NASA projects [8], the modules of interest detected, diagnosed, and recovered from system faults and thus were critical to the safety of the overall spacecraft.

4. Concluding Remarks

Formal requirements specifications are still quite rare in software practice. One approach to overcoming this problem is for specification experts to work with practitioners to produce an initial requirements specification and then to provide support when practitioners extend the specification to include more behavior. Another promising approach is to develop technology for use in synthesizing an operational specification from a set of scenarios (see, e.g., [20]). While the operational specification will be incomplete, it does provide an initial specification that software developers can extend and modify to produce a more complete specification.

References

[1] R. Bharadwaj and C. Heitmeyer. Developing high assurance avionics systems with the SCR requirements method. In *Proc. 19th Digital Avionics Sys. Conf.*, 2000.

[2] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Sci. of Comp. Prog.*, 20(1-2):3–50, 1993.

[3] S. Garland. TIOA User Guide and Reference Manual. Technical report, MIT CSAIL, Cambridge, MA, 2006.

[4] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2002.

[5] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3), 2000.

[6] M. P. Heimdahl and N. G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Trans. on Softw. Eng.*, 22(6):363–377, 1996.

[7] C. Heitmeyer and R. Bharadwaj. Applying the SCR requirements method to the light control case study. *J. Univ. Comp. Sci.*, 6(7), 2000.

[8] C. Heitmeyer and R. Jeffords. Applying a formal requirements method to three NASA systems: Lessons learned. In *Proc. 2007 IEEE Aerospace Conf.*, 2007.

[9] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Trans. on Softw. Eng. and Methodol.*, 5:231–261, July 1996.

[10] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), 1998.

[11] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proc. 13th ACM Conf. on Comp. and Comm. Sec.*, 2006.

[12] C. L. Heitmeyer and J. D. McLean. Abstract requirements specification: A new approach and its application. *IEEE Trans. on Softw. Eng.*, SE-9(5):580–589, 1983.

[13] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[14] L. Lamport. Verification and specifications of concurrent programs. In *REX School/Symp.*, pages 347–374, 1993.

[15] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. 10th ACM SIGSOFT Symp. on Found. of Softw. Eng.*, 2002.

[16] S. P. Miller and A. Tribble. Extending the four-variable model to bridge the system-software gap. In *Proc. 20th Digital Avionics Sys. Conf.*, Oct. 2001.

[17] D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. of Comp. Prog.*, 25(1):41–61, 1995.

[18] D. Pavlovic and D. Smith. Software development by refinement. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*. Springer, 2003. LNCS 2757.

[19] R. Seater and D. Jackson. Requirement progression in problem frames applied to a proton therapy system. In *14th IEEE Intern. Req. Eng. Conf. (RE'06)*, pages 169–178, 2006.

[20] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE*, 2007.

[21] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1), 1997.