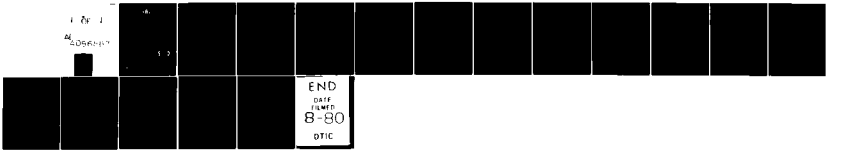
AD-A086 807 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
'REAL-WORLD' PROPERTIES IN THE REQUIREMENTS FOR EMBEDDED SYSTEM--ETC(U)
1979 P ZAVE N00014-77-C-0623
UNCLASSIFIED NL



END
DATE FILMED
8-80
DTIC

"REAL-WORLD" PROPERTIES IN THE REQUIREMENTS FOR EMBEDDED SYSTEMS*

Pamela Zave
Department of Computer Science
University of Maryland
College Park, Maryland 20742

N00014-77-C-0623

1979

*This research was supported in part by the Office of Naval Research, Mathematics Branch, under Contract N00014-77-C-0623, and by the Computer Science Center, University of Maryland, College Park.

"REAL-WORLD" PROPERTIES IN THE REQUIREMENTS FOR EMBEDDED SYSTEMS

1. Introduction

As the availability of computer hardware in all price ranges increases, more and more attention is being focused on "embedded systems", a category that includes most of the important new applications (vastly conceived communication networks, "intelligence" in every car and household appliance) of computers, as well as the least-understood current applications (ballistic missile defense systems, space software, etc.). The importance of requirements analysis and specification in orderly system development is also widely recognized ([Ross 77], [Zave 79a]). This paper addresses a central issue in the specification of requirements for embedded systems: How can the ragged and open-ended properties of the "real world" be usefully included in a coherent, closed requirements specification?

Certainly the term "embedded system" is closely associated ~~in our minds~~ with the physical properties of computing environments, such as time and hardware reliability. Section 2 argues that emphasis on real-world, or physical, properties defines embedded systems, so that the role of these properties in requirements specification for embedded systems is indeed central. Section 3 sketches an approach to the requirements problem which should be well suited to the inclusion of physical attributes, but explains that a nagging problem remains. Section 4 presents a trial solution to this problem.

2. A Working Definition of "Embedded System"

Intuitively, an embedded system is a system that is explicitly viewed as being an essential part of some larger system. The other parts of the larger system make up the environment of the embedded system, and the embedded system plays its role by interacting with its environment. But any computer program can be seen in this way, so the real question is, "When is it profitable to do so?" Airline reservation systems, patient-monitoring systems, and real-time process control systems are considered to be embedded systems; compilers, differential equation solvers, and payroll programs are not.

The common element among the "embedded" ones seems to be that their successful operation depends directly on cognizance of, and adaptation to, physical properties of their environments. The non-embedded systems, which may be called "tools", have no such requirements. They are designed as input-to-output transducers, and their quality is usually judged on the convenience and generality of those inputs and outputs.

To test this hypothesis, let us look more closely at the payroll program. If satisfying the intuitive definition were enough, this would be an embedded system--because its role in a larger system, the organization, is very clear. And precisely because of its essential relation to its environment, real-world properties are not irrelevant to its success: the checks must be printed by payday, with essentially perfect reliability. The reason the payroll program is not regarded as an embedded system is that responsibility for coping with the physical world is

taken by its environment (the accounting department staff who schedule computer time early in each pay period to run--and perhaps rerun--it), and is not part of the requirements for the program. A payroll system capable of handling all its timing and reliability problems automatically would look like an embedded system to anyone's eye.

Thus our working definition of an embedded system is one for which properties of the physical world appear somehow in its (mandatory) requirements. This definition can be used to distinguish between programs that run under an operating system, and the operating system itself.

Consider, for instance, an editor, which is a "tool" that is used in interactive mode. The response time experienced by a user of the editor has little to do with the efficiency of the editing program itself, and even if response time were under control of the editor, most users would prefer an elegant command language to very fast response. "Reliability" of the editor is equated with software correctness, leaving the reliability of everything else to the host computer, its operating system, and its operating personnel. Neither time nor reliability would be likely to appear in the requirements for an editor, and differences in speed could only discriminate among high- and low-quality products, all satisfying the same requirements. A shift in the viewpoint from which the editor is regarded could elevate response time to a requirement, of course, but in this case it would probably have to be moved off the general-purpose system to a dedicated machine.

The operating system, on the other hand, is an embedded system, although one whose performance (timing) requirements are difficult to formulate, being load-dependent in complex ways. Its "embeddedness" can be seen most clearly through the issue of reliability. I/O devices are an unreliable part of the environment of the operating system, but the system is required to deliver services involving them that appear perfectly reliable to the users.

3. An Approach to Requirements Specification

In [Zave 79a] an approach to specification of requirements for embedded systems is presented. It is based on a language in which a requirements specification is an interpretable (simulation) model of the required digital system interacting explicitly with its environment. This "operational" approach has the major advantages of: (a) producing a formal, unambiguous, and internally consistent requirements document; (b) providing a testable set of requirements and a test bed (the simulatable environment model) against which to test both the system requirements and the eventual system; (c) making possible the specification of a broad range of requirements, including non-logical ones; and (d) emphasizing the customer's problem environment rather than the solution system, making it much more likely that the customer will communicate what he needs and get what he wants. (More details on this approach can be found in [Zave 79b] and [Zave 79c].)

An operational approach will be worthless, however, unless the operational specification language is general enough to represent the wide variety of natural objects that can interact with an artificial system, and abstract enough to avoid forcing unnecessary design and implementation decisions into the requirements. The features of our language seem to satisfy these criteria.

The "global" system (digital system plus environment) is specified as a set of asynchronously interacting digital processes. A digital process is simply a formal object with a

state space (set of possible states) and a successor relation associating "present" states with "next" states. Its behavior over time is to make discrete (or "digital") state changes (from "present" to "next" states) at finite intervals. Thus a process can be used as a discrete simulation of a person, ballistic missile, or other non-digital object. Its power of abstraction and generality within the realm of digital systems have long been established ([Horning & Randell 73]).

In the main example to be used in this paper, we will model the environment of an on-line database system, such as an airline reservation system, as a set of "terminal" processes. Each such process represents a CRT terminal interacting with the central system, and includes the user operating the terminal. Although there are many ways to do it, the way chosen here is to make the present state of a terminal process encode its current display, and to have a process step model one interaction with the central system. Thus its successor relation would be declared as:

terminal-cycle: DISPLAY ---> DISPLAY

where "DISPLAY" is the set of all possible displays, and "terminal-cycle" maps a display onto a subsequent display reflecting one more transaction.

Within each process, the successor relation is specified using functional notation, i.e. expressions formed from constants, formal parameters, primitive functions, and functional operators such as composition. Functional notation is used, not for the mathematical property of mapping an argument to a deterministic value, but for its great power of abstraction. A

functional expression specifies exactly what computation is to be done without constraining the data, control, or processor structures used to do it (this is expounded nicely in [Backus 78]). A primitive "function" in a specification is interpreted simply as an arbitrary relation from its domain to its range, both of which must be declared. It may represent a mapping which is truly unknowable from the system's view (such as the result of an interactive user's thought), a set of deferred decisions (in which case it will be elaborated later in terms of other primitives), or an asynchronous interaction (the primitives for asynchronous interaction look like functions in this language).

"Terminal-cycle", for instance, may be elaborated as:

```
terminal-cycle(d) =
     display(display-and-transact(d,think-of-request(d)))

think-of-request:  DISPLAY ---> REQUEST

display-and-transact:
     DISPLAY x REQUEST
---> DISPLAY x (RESPONSE U ERROR-MESSAGE)

display-and-transact(d,r) =
     (display(d,r),transact(r))

transact:  REQUEST ---> RESPONSE U ERROR-MESSAGE

display:
     DISPLAY x (REQUEST U RESPONSE U ERROR-MESSAGE)
---> DISPLAY.
```

"Think-of-request" models the user's thought in choosing a next request to type in, based on the current display; it can probably never by specified deterministically. "Display" may or may not be elaborated, depending on how much detail about the CRT is needed. When "transact" is elaborated, there will necessarily be a use of interaction primitives to convey the request to the

central system, and the response back.

Thus the combination of digital processes and functional notation seems to be general and abstract enough to make even an operational language suitable for requirements (another example of this general type is [Smoliar 79]). The operational approach also seems ideal for introduction of physical attributes, because the requirements specification naturally contains the logical structures to which they should be attached.

Timing and reliability requirements are specified by attaching timing and reliability attributes to functions in the logical specification. A timing attribute is a random variable whose values represent evaluation times, and any information about its distribution may be given. A reliability attribute can only be attached to a function in the logical specification with different sets of range values for "success" and "failure" outcomes. It is a random variable whose values represent the occurrences of "success" vs. "failure" outcomes, and any information about its distribution may be given.

The following performance requirements would be appropriate for an airline reservation system, for example:

T/transact/

maximum is 3 seconds

R/transact/

minimum probability of "success" is .99

The former says that the response time (which corresponds to the total evaluation time of "transact") must be less than or equal to 3 seconds; the latter says that at least 99 out of 100

evaluations of "transact" must yield values in the set "RESPONSE" rather than values in the set "ERROR-MESSAGE".

The examples in [Zave 79b] show that just timing and reliability requirements, including those attached to the environment model as well as the system model, are capable of specifying all the types of system requirements presently recognized as performance requirements (response times, system loads, "fail-safe" operation, "fail-soft", etc.). This is very satisfactory, because we have added only a small set of features to the language, all compatible with its goals. These non-logical requirements can be formally defined, simulated, tested, and checked for internal consistency just as the logical ones can.

There are still unmanageable examples, however. What about distance in a distributed system? We can certainly attach distance measures to the points of asynchronous interaction in the specification (and these distances do affect the feasibility of certain timing requirements), but is there any real value in adding them to the language? They are meaningless as far as simulation is concerned (since time constraints are separately specified), and would seem to have no more force than informal comments, which would be allowed regardless of the language definition.

Even worse problems are raised by the following requirements statement: "An on-board control program for a satellite is to be written. It must fit in 8K. After the primary function has been taken care of, it should check for as many error conditions as it

can, as constrained by the size of the program." Here the requirements are affected by the size of a physical memory, which is in turn limited by volume and weight. It is easy to imagine similar examples where power consumption, heat dissipation, or just about anything else could become relevant.

The problems raised by the satellite program are serious for two reasons. One is that the list of potentially relevant physical attributes has been shown to be open-ended. The other is that there are no structures in the specification to which these requirements can be attached, nor should there be until well into the design phase, when computational resources and the allocation thereof are properly introduced.

These observations discourage hope that requirements can ever be specified in a formal, unified language, in which all the requirements for a system can be inter-related in an effective, testable manner. The next section, however, proposes a possible solution to the dilemma.

4. The "Real World" and the Computational Domain

"Computer science" is concerned with knowledge in the domain of computation, which is based on two fundamental concepts: discrete mathematics (logic) and relative time. It is easily distinguished from mathematics by the notion of relative time, or sequence, which is what makes an algorithm different from a mathematical function. It is easily distinguished from electrical engineering or physics by the lack of such "real-world" concepts as voltage and resistance. In the scope of our present discussion, "real-world" refers to the open-ended, informally characterized domain we perceive with our senses, and is distinguished from the computational domain--the only domain to which the goals, methods, and results of computer science apply.

This distinction reveals the folly of trying to encode properties of the real world in formal requirements specifications, which are objects belonging to the computational domain. All we need and want are means with which to encode all the possible effects of the real world on what goes on in the computational domain. Surprisingly enough, the language features presented above seem necessary and sufficient to do this, based on the examples we have encountered so far.

Quantified, continuous time is necessary to encode conveniently the effects of real time on relative time within a digital system. Although "event A precedes event B" is sufficient for the computational domain, "event A occurs at time t" is a statement from which many relations of the preceding form

can be derived. A stochastic notion of reliability is necessary to encode the fact that sometimes components do not produce the results formally defined for them, for reasons forever beyond the reach of computational reasoning.

Our "unmanageable" examples become manageable in this light. Distance, for instance, can have many effects on computations: the relative time for inter-process interactions becomes great, the reliability of a component which must bridge great distances is less than that of a localized one, logical complexity must often be increased to cope with these special timing and reliability problems, etc. But all these effects are straightforwardly expressed in the current language!

The property of distance seems very much a special case, but that is precisely the point. Using an operational language can give us confidence that when new and unforeseen special cases arise, the existing language will be capable of coping with their effects on the computational domain--just because the language is already capable of very general expression within the computational domain.

Before we can deal with the physical properties mentioned in the satellite problem, we must go back to the concept of "environment". In a requirements specification for a system, the environment processes would represent objects such as people, terminals, missiles, etc. But another projected application of this specification language is to system design: we hope to develop designs for embedded systems from their requirements specifications, in the same language, by the explicit elaboration

of resource management structures. In the elaborated specification produced by such a design effort, there would be new processes explicitly representing computational resources, including memories, disks, and special processors. Those that were not to be developed by the system developers would be, by definition, part of the environment--but playing a somewhat different role than the environment processes specified as part of the requirements.

Obviously, memory size, weight, volume, power consumption, and heat dissipation all concern computational resources used to implement the satellite control system. Thus the specification problem divides into two subproblems: (1) How can these physical properties of the computational resources be specified when, as in design, the computational resources themselves are explicitly represented? (2) How can constraints on these physical properties be handled during requirements phases, when resources are not and should not be specified?

The answer to (1) is that the only way the weight, volume, etc. of a resource could affect matters in the computational domain would be to make the implemented system too heavy, large, etc., necessitating removal of that resource. Removal of the resource is easily reflected in the formal specification by removal of its representation.

The second problem is far more grave, because it brings up the hardest engineering problem of all: having to make decisions (e.g. requirements decisions) before their consequences (on performance or resources or logical complexity) can be predicted.

Fortunately, specifying these decisions is easier than making them. Since the only effect that the physical properties of resources can have on the system is to dictate fewer resources, the only effect they can have on the system requirements is to dictate fewer functions. In the satellite program, for instance, resource limitations may eliminate some desirable, but non-essential, error checking.

These non-essential functions can be specified in the requirements as follows: Every possible error check should be thought through and formulated in the specification as a primitive function capable of performing that check. The "success" range for each such function would be { ok, not-ok }, and the "failure" range would be { ok }. Thus failure of the error-checking component has the same effect as doing no checking. If the component cannot be implemented because of resource limitations it always fails. The non-essential nature of such a component can be specified by the reliability requirement "minimum probability of success is 0".

It should be noted that in this language, timing and reliability requirements are independent of one another. This means that when a component fails, not only must it produce a well-defined and previously anticipated value, but also must follow the same time constraints as if it did not fail. Dealing with performance constraints and unreliability individually are already so difficult that there seems to be little chance of success if this sort of discipline is not enforced.

5. Conclusion

This paper has presented some speculations about the nature of embedded systems, the real-world properties that affect them, and how these properties can be taken into account in a methodology for requirements and design.

This work is particularly difficult because the current literature does not even contain a provocative set of examples against which ideas can be tried (the satellite problem is real, but my hearing about it was sheer chance). I hope that these speculations will at least stimulate further discussion, so that the real problems will define themselves, and the real solutions begin to emerge.

6. Acknowledgment

Dick Hamlet's helpful comments on an earlier draft of this paper improved it substantially.

7. References

[Backus 78]
Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs." Comm. ACM 21, August 1978, pp. 613-641.

[Horning & Randell 73]
Horning, J. J., and Randell, B. "Process Structuring." Computing Surv. 5, March 1973, pp. 5-30.

[Ross 77]
Ross, Douglas T., ed. Special Collection on Requirement Analysis. Trans. Soft. Engr. SE-3, January 1977.

[Smoliar 79]
Smoliar, Stephen W. "Using Applicative Techniques to Design Distributed Systems." Proc. Specifications of Reliable Software Conf., Cambridge, Mass., April 1979, pp. 150-161.

[Zave 79a]
Zave, Pamela. "A Comprehensive Approach to Requirements Problems." Proc. COMPSAC, 1979, pp. 117-122.

[Zave 79b]
Zave, Pamela. "Formal Specification of Complete and Consistent Performance Requirements." Proc. Texas Conf. on Computing Systems, 1979, pp. 4B-18 - 4B-25.

[Zave 79c]
Zave, Pamela. "Formal Specification of Asynchronous Processes and its Application to the Early Phases of System Development." University of Maryland Computer Science TR-775, 1979.