Sixth International Conference on Testing Computer Software

Managing the Testing Process

May 22-25, 1989 • Washington, D.C.

VOLUME II

Plenary and Track Sessions
Wednesday—May 24, 1989 & Thursday—May 25, 1989

Sponsored by
Data Processing Management Association Education Foundation

In Cooperation With
American Society for Quality Control
Association for Computing Machinery SIGBDP and SIGSOFT
International Test and Evaluation Association

Program Managers
Software Quality Engineering

S100545

Seminar Management by

USPDI

U.S. Professional Development Institute
1734 Elton Road, Suite 221, Silver Spring, MD 20903
301/445-4400 • FTS Users 202/445-4400

ENHANCING TESTABILITY WITH SCENARIO-ORIENTED ENGINEERING

Michael S. Deutsch
Hughes Aircraft Company

1. INTRODUCTION

Scenario-oriented engineering is an approach to full lifecycle software engineering that centers upon the working concept of system operations that is expressed in what I call scenarios. A scenario defines a continuous path from an external stimulus into a system through a response back into the external environment, a behavior pattern that would be visible to a system user. This paper explains the notion that the scenario is a powerful macro-object that can be used to conceive, define, design, validate, test, and maintain a system. The use of a common object in all phases of the engineering lifecycle is a major step in maximizing the resemblance between the problem solution and the problem statement. Recent advances in object-oriented software technology (e.g., see [1, 2, 3]), including Ada, allow this scenario oriented paradigm to be more readily realized in a large-scale software domain. However, the ultimate value of this approach is for full system engineering of combined software/hardware systems.

Scenario-oriented engineering is empirically derived from a fundamental lesson learned in the overall system engineering of large, complex systems consisting of interacting hardware/ software elements (e.g., satellite communications, weapons control, process control, sensor control/reporting, local area network applications): these multi-mode, multi-state systems are most successful, testable, and maintainable when operational usage patterns are visibly incorporated into their design and validation. This incorporation is a direct reflection of an in-depth understanding of the applications problem.

In my experience, the human mindset seems to conceive systems serially in a linear-like scenario-based style of thinking. System engineers can comfortably describe how they would like a system to respond to an external stimulus, such as redirecting a sensor when a target drifts to the edge of the field-of-view. Harel [4] has expressed this same view in his experiences with embedded software applications for aircraft avionics. I believe it is now possible to use the scenario not only as the object for system conception and requirements, but also as the physical design partition, as the semi-autonomous unit for testing, and as the basis for operationally visible maintenance modifications. The basic approach does not distinguish between software, hardware, or manual domains; its main leverage, however, is with complex software intensive problems.

A language such as Ada gives us a facility for expressing a scenario as a structural element. The *package* allows the physical consolidation of all resources that are necessary to implement the stimulus/response behavior of a scenario which may include procedures, abstract data, functions, and tasks. Mills [5] makes a similar reference to such an object calling it a data abstraction. The scenario thus becomes a physical object that can be directly manipulated in the design domain; the package can be tested with relative autonomy one-to-one against the original stimulus/response specification of the scenario. Maintenance modifications and retesting are more easily done by originating changes in the stimulus/response scenario specification and altering the directly corresponding package. Furthermore, the change is more likely to be localized to an individual scenario package.

Scenario-oriented engineering is a related but more global approach than object-oriented methods, the former method filling significant gaps in the latter. As Booch observes "object-oriented development is a partial-lifecycle method; it focuses upon the design and implementation stages of software development" [1]. It is also generally claimed that object oriented solutions are inherently easier to test, modify, and debug. The scenario-oriented method offers further concrete assurance of these attributes by providing:

- Direct one-to-one linkage between the user's view of system operations and implementing structural elements;
- Direct one-to-one correspondence between the test procedure (the scenario) and the structural element being tested (e.g., an Ada package);

- Ease of traceability of maintenance changes from user observable behavior, to changed structural elements, to localized retesting.

Scenario-orientation thus approaches a full-lifecycle software engineering methodology.

2. EXAMPLE

Let us consider the cruise-control system that maintains the speed of a car to demonstrate a scenario analysis. This is the same example system [6] used by Booch [1] to demonstrate object-oriented advantages over functional decomposition.

Figure 1 shows the basic inputs and outputs for the cruise-control system. An informal statement of the problem is shown in the table below.

We would approach the scenario analysis by mentally rehearsing the working profile of this system in our minds by keying on the stimuli originating in the external environment and asserting the necessary responses back into that external environment. Data events, control events, and conditions of the system and environment objects should be considered in each stimulus and response. One way to get started is to proceed chronologically focusing first on a single physical object in the external environment. Nothing happens until the driver turns-on the engine. Thus, the stimulus for the first scenario is that the driver successfully turns-on the engine and the response is that the engine is now in an "on" condition. This and subsequent scenarios are delineated on Table 1. Some scenarios key on changes of conditions and are roughly analogous to abstract state machines, but other scenarios do not. Temporal performance needs can, at this point, be assigned to each stimulus/response scenario.

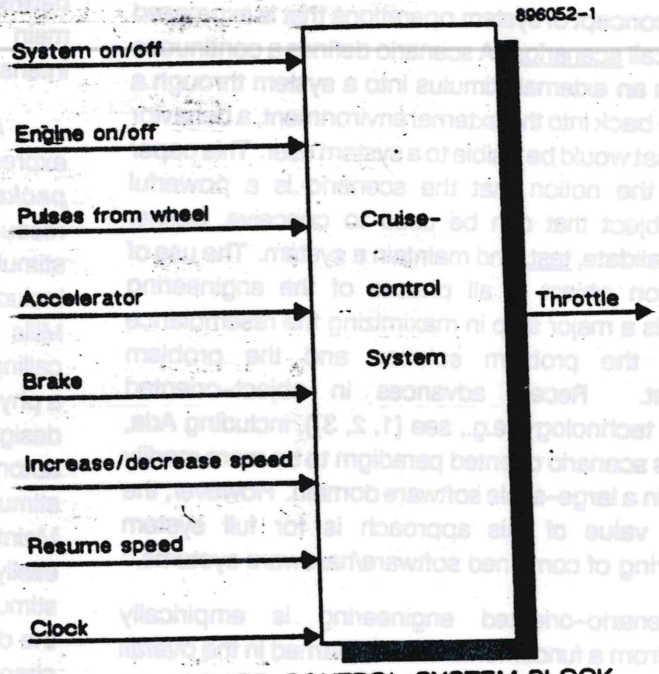

FIGURE 1. CRUISE-CONTROL SYSTEM BLOCK DIAGRAM

Inputs	
• System on/off	If on, denotes that the cruise-control system should maintain the car speed.
• Engine on/off	If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
• Pulses from wheel	A pulse is sent for every revolution of the wheel.
• Accelerator	Indication of how far the accelerator has been pressed.
• Brake	On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
• Increase/Decrease Speed	Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
• Resume	Resume the last maintained speed; only applicable if the cruise-control system is on.
• Clock	Timing pulse every millisecond.
Output	
• Throttle	Digital value for the engine throttle setting.

TABLE 1. CRUISE-CONTROL SYSTEM STIMULUS/RESPONSE SCENARIOS

Stimulus		Response
• Driver starts engine • Engine in "off" condition • System in "off" condition	1 ------------>	• Activate monitoring engine condition, driver actions • Engine in "on" condition
• Driver turns off engine • Engine in "on" condition	2 ------------>	• Deactivate control system, monitoring • Engine, system in "off" condition
• Driver turns system on • Engine in "on" condition • System in "off" condition	3 ------------>	• System in "maintain" condition • Activate monitoring of clock, wheel pulses
• Driver turns system off • System in "maintain" condition	4 ------------>	• Deactivate control system • System in "off" condition
• System in "maintain" condition • Receive clock, wheel pulses	5 ------------>	• Calculate throttle setting • Issue throttle setting
• System in "maintain" condition • Receive clock, wheel pulses • Driver increases or decreases maintained speed	6 ------------>	• Calculate new throttle setting • Issue throttle setting
• System in "maintain" condition • Driver presses brake	7 ------------>	• Deactivate control system • System in "manual control" condition • Deactivate auto throttle setting
• System in "manual control" condition • Driver requests resume speed	8 ------------>	• System in "maintain" condition • Monitor clock, wheel pulses • Deactivate auto throttle setting • Issue throttle setting
• Engine in "on" condition • System in any condition • Driver adjusts accelerator	9 ------------>	• Calculate new throttle setting • Issue throttle setting

896052-6

The scenarios are interconnected in Figure 2 to show, in a simplified fashion, the essential control flow relationships among scenarios. Only the scenario numbers referring back to Table 1 are indicated for illustration convenience. The multiple paths epitomize the asynchronous nature of event driven systems. This single diagram provides three fundamental views of this system:

1) <u>A top-level requirements specification</u> in the user's coordinate system from which more detail can be specified regarding the essential functions and interfaces.

2) <u>A top-level design architecture</u> where each scenario becomes a major physical component of the system.

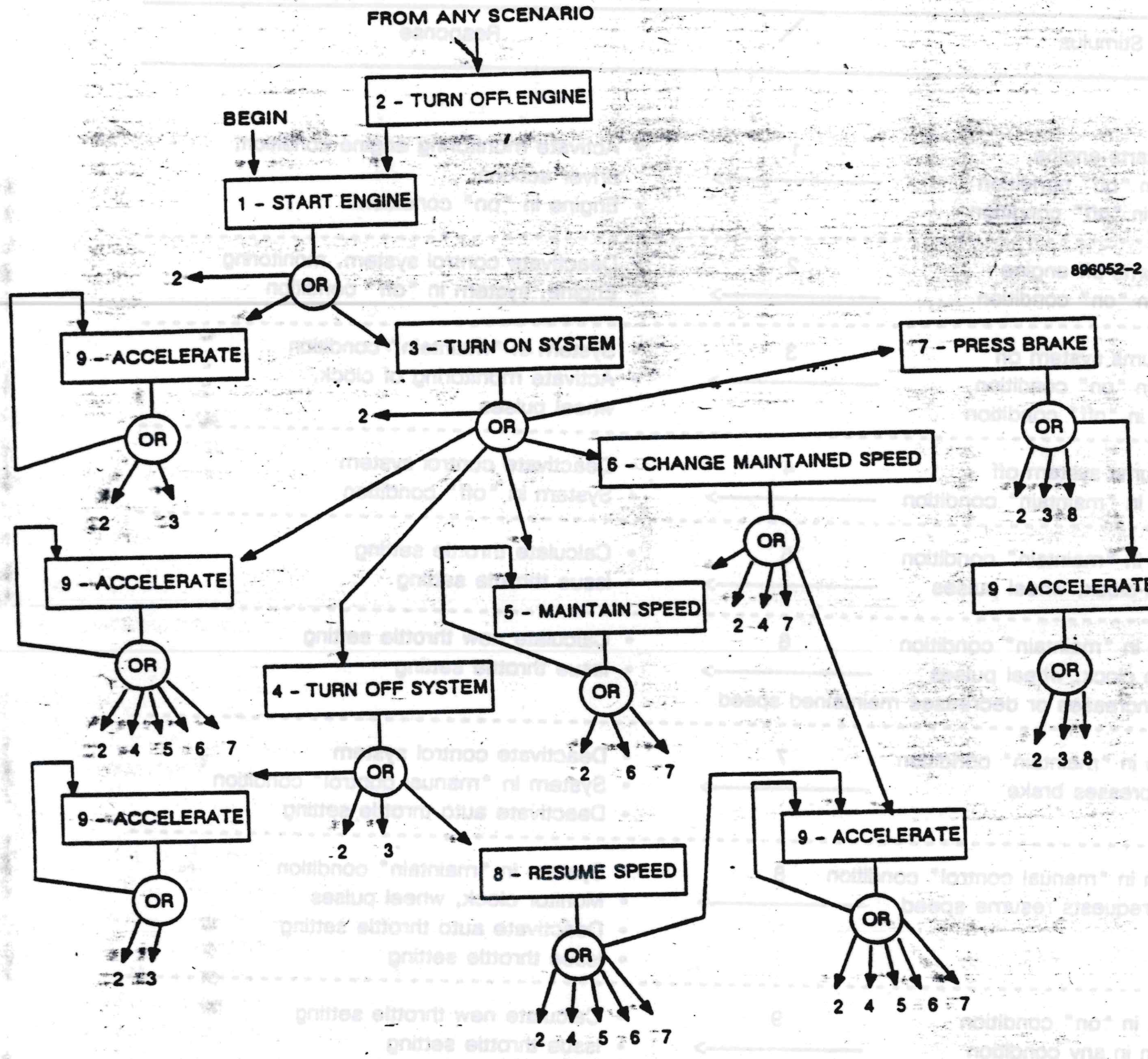

FIGURE 2. CRUISE-CONTROL SYSTEM STIMULUS/RESPONSE SCENARIO DIAGRAM

3) The top-level system test plan where the stimuli roughly define input test cases and the responses approximate the expected system behavior to the test cases.

The use of Ada packages permits us to avoid many aspects of the "semantic gap" between requirements and design notations by providing a one-to-one correspondence.

Suppose the cruise-control system includes two microcomputers (as Booch postulates), one for managing the current and desired speeds and the second to manage the throttle. Several scenarios receive their stimuli in the first machine and respond in the second. Ada packaging for these scenarios would mask this inter-machine interface from the concern of the software designer assuming that a network operating system were available to support the Ada environment. With a traditional operating system, the

[illegible]

packaging would have to be partitioned but would identify a thread that is a candidate for early integration. My practical experience on large systems using non-Ada languages has concentrated on using scenarios for integration threads. This background strongly suggests that the full scenario-oriented method scales up to large systems. A recent event-driven communications system consisting of 500,000 source lines of code and large amounts of diverse hardware was originally specified in thirty-five stimulus/response scenarios.

3. PROCESS OF SCENARIO-ORIENTED SOFTWARE ENGINEERING

The process of scenario-oriented engineering is now looked at more closely and synthesized into a paradigm. In this section I explain the overall process with occasional reference to our stereotype implementing mechanism, Ada. A specific scenario-oriented design approach keyed to Ada expands on the design philosophy in Section 5. A more precise exploration of the properties of scenarios is in Section 4.

The scenario models the combined automated and manually implemented behavior sequences thus expressing the fundamental needs and reasons for existence of the system. Scenarios then represent a high fidelity model of reality, and we would like to implement them with minimal distortion. The major steps of the scenario-oriented paradigm are as follows:

- Identify objects in system environment that will participate in stimulus/response behaviors;
- Form scenarios connecting stimuli with responses;
- Define details of functional and performance requirements;
- Validate requirements against scenarios;
- Define system design components around scenarios;
- Validate system design against scenarios forming threads;
- Incrementally implement, test, and integrate system in units of threads; and
- Maintain and modify system.

The whole intent of the method is to retain the scenario intact across this sequence of events so that the eventual system is the model of reality. We would expect feedback and iteration among these steps although the net movement is forward. As I will explain in a moment, the scenario-oriented approach supports experimentation, early prototyping, and evolutionary development.

The first step, ***identify objects in the system environment,*** entails recognizing the elements that provide stimuli into the processing system and which may require responses. These elements may include sensors, people, or other devices that participate in stimulus/response behaviors. At this time, these objects are treated as interfacing directly with the processing system without regard to physical interface media. In the cruise control example these objects would include the driver, the wheel rate sensor, brake, accelerator, and resume control.

The next step, ***form scenarios,*** first concentrates on one of the objects in the environment to identify stimuli in the form of data, events, and existing conditions. The response back into the environment is postulated consisting of resulting data, events, and conditions. The response may be directed back to the originating object, other objects, or some combination. Empirically, it appears that it is best to start dealing with the most dominant of the objects in the environment and then sequentially proceed to the other objects; this is usually a multi-pass procedure with each iteration around the objects contributing to a more complete and consistent depiction of the application domain. In the cruise control example, we would begin with the driver as the dominant object. As shown in Figure 2, the scenarios are interconnected with sequential and Boolean "and/or" connectors to provide an overall view of how the system works in an informal notation that can be conveyed and reviewed by a non-technical client. Davis [7] notes how more formal state-oriented notations such as finite state machines, statecharts, or Petri nets are not intuitive to most non-computer trained persons; most users are applications experts but not likely to be computer trained.

The scenario approach inflicts early consideration of control flow. Lavi [8] discusses the fallacies of partitioning embedded systems into subsystems without early consideration of the control flow that determines the joint logic behavior of the entire system. The early emphasis on control behavior illuminates certain non-functional derived requirements such as the need for a system controller. Other derived requirements like the need for device drivers and interrupt handlers are brought out as the external interfaces are identified through the stimulus/response analysis. The stimulus/response scenarios also provide a baseline for postulating scenario mutations that could result from safety critical failures and the need for fault tolerant or fail safe responses.

In the third step, ***define functional and performance requirements,*** we characterize the details in these areas. It is at this time that we assign performance properties, such as a response time requirement, to each scenario. The scenarios can be simulated, singly and in aggregate, to determine if predicted system dynamic properties conform with requirements. The scenario definitions inherently identify functions and data inputs/outputs that play a role in each stimulus to response transformation. In the cruise control system, for example, we note required functions such as "calculate throttle setting" and "issue throttle setting" in scenarios 5, 6, 8, and 9. As one way of proceeding, we could simply expand into detail the processing required of each function identified in the scenarios along with their inputs and outputs. Another more powerful option is to use the scenarios in combination with an established specification method like structured analysis to expand the functional details progressively in a data flow analysis as explained in [9].

The ***requirements are validated against the scenarios*** as soon as the details of the processing functions and their inputs/outputs are available in draft form. You trace each scenario through the detailed requirements text by correlating each requirements paragraph number (or other identifier) with the stimulus/response behavior pattern. You can now correct any omissions, contradictions, or redundancies revealed in either the requirements or the scenarios by this paper execution of the scenarios.

Design components are defined around scenarios by either encapsulation or allocation. Encapsulation of the required procedures, tasks, and abstract data within a single structural element, such as an Ada package, is the most visible and preferred implementation of the scenario. Traditional design methods, such as structured design, and programming languages, such as FORTRAN, do not directly support visible encapsulation. Instead, the scenario behavior must be allocated to individual design components that will be involved with the stimulus-to-response transformation. Consideration of each stimulus-to-response thread during the design stage will contribute heavily to a testable design and implementation.

After identifying each scenario design object, such as an Ada package, the design must be further developed to consider upper level control. In a simple situation, one or more scenarios may operate asynchronously without hierarchical control perhaps engaging in rendezvous to exchange messages. An explicit control function may be present to guide overall system operational behavior. Representative configurations are shown in Figure 3; the downward arrows denote the scenario resources that are inherited by each controller. The controller may be implemented as a centralized software procedure, may be a hardware element, may be multilevel implementing hierarchies of decisions, or may be conceptual and distributed among scenarios [8]. We would normally expect intimate coupling between the controllers and human/machine interface of the system, thus, pointing to the need for defining human and automated activities in the original scenarios. In general, the scenario packaging results in a flatter control hierarchy in comparison with modular decomposition techniques.

You ***validate the system design*** by testing, on paper, the design with the stimulus/response scenarios as test cases. You form a thread of components, both software and hardware, that provides the external behavior demanded by each scenario. For an Ada packaging implementation, this "threading" should be without complication. The formation of threads through other implementations commonly yields less than a clear correlation between scenarios and design components; further design iteration is in order should this occur. The validated threads and scenarios are the initial step in planning system integration and the final acceptance test. You can also simulate each thread to verify response-time requirements.

Each scenario thread is ***incrementally implemented and integrated*** into an evolving baseline of previous threads. The thread scenario is the basis for the test procedure. Each incremental integration test is a partial dress rehearsal for the final acceptance test. The use of a common object, based upon external

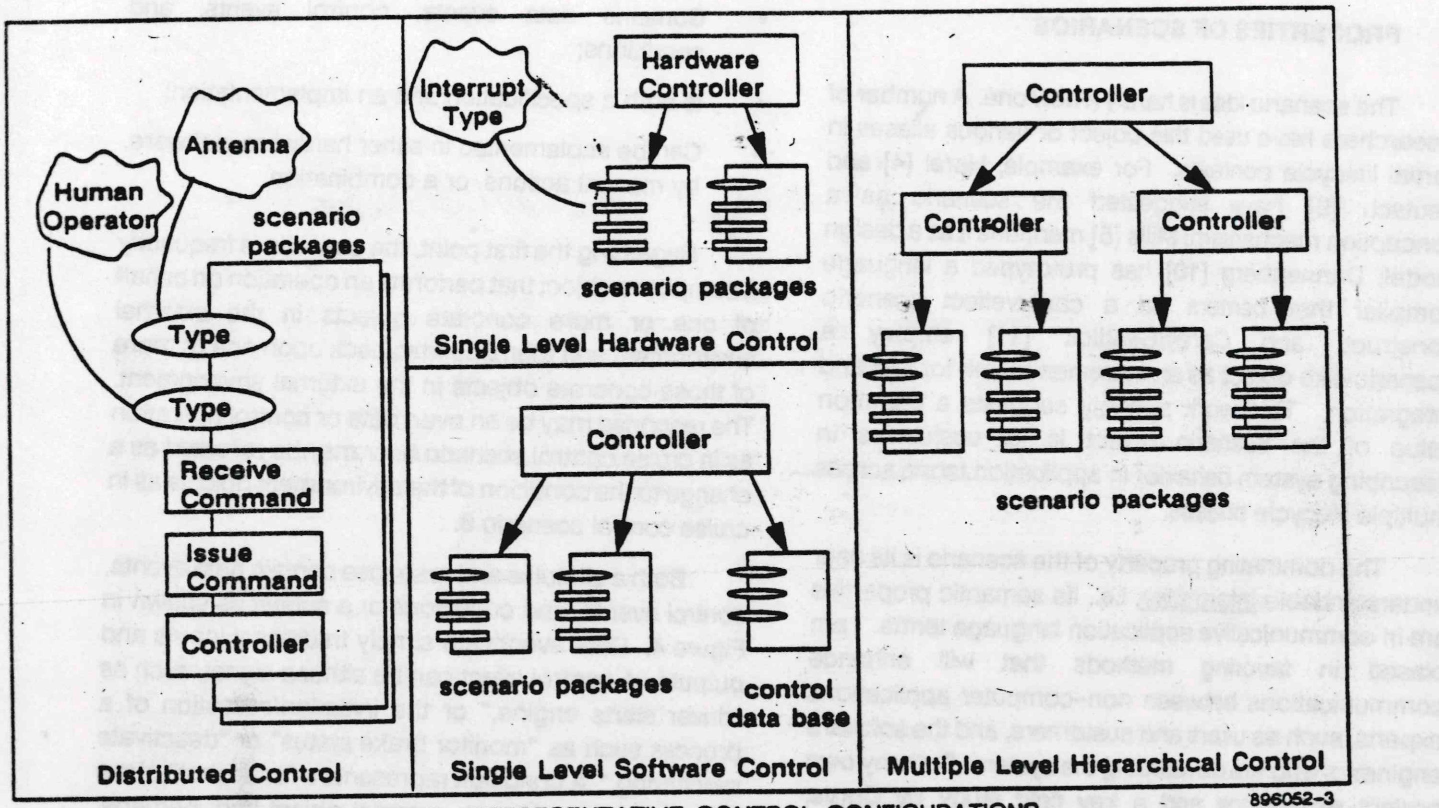

FIGURE 3. REPRESENTATIVE CONTROL CONFIGURATIONS

behavior, allows simpler and more visible management planning and feedback.

There is as yet no significant field experience in ***maintaining and modifying systems*** using this full paradigm with Ada. Yet, we can extrapolate from non-Ada systems. It appears that by maintaining a crisp equivalence of system physical structure with changing operational usage patterns while performing modifications, the system will maintain clarity with a more visible level of correctness. Much of the "blurring" that occurs in long-lived and frequently-modified software systems has resulted from the absence of clear correspondence between software architecture and usage scenarios; this indefinity tends to erode further with time when the correspondence is initially weak. The overall maintenance sequence is to change the scenario specification and existing test cases, modify the directly corresponding package, and test the package against the scenario. Because the scenario is relatively autonomous, encapsulating a full path through the external interfaces, retesting should usually be localized requiring a lesser degree of regression testing of other scenarios. The overall thesis is that modifications can be made as simple and obviously correct as possible, with minimum unpredictable behavior, when this clear correspondence between architecture and operational scenarios is present.

A further value of the scenario approach is to enable experimentation and evolutionary system development. This value is derived from the characteristic that a scenario contains a semi-autonomous working subset of system behavior. Early "fuzzy" conceptual thinking can be documented in representative scenarios. Mental rehearsal of these initial scenarios, while postulating differing underlying conditions, suggest less typical, atypical, and pathological mutations that require different responses to the same basic stimulus. Risky or uncertain scenarios become candidates for early prototypes or further in-depth analysis. The major typical scenarios can be implemented first to form a system skeleton or operational prototype. Additional scenarios are added to evolve this baseline in an incremental integration sequence. As a surgical mechanism for isolating working subsets of system behavior from concepts to implementation, scenarios would appear to be an enabling mechanism for a practical realization of the spiral lifecycle model concept.

4. PROPERTIES OF SCENARIOS

The scenario idea is hardly a new one. A number of researchers have used this object or various aliases in partial lifecycle contexts. For example, Harel [4] and Deutsch [9] have suggested the scenario as a conception mechanism; Mills [5] mentions it as a design model; Dannenberg [10] has prototyped a language compiler that centers on a cause/effect scenario construct; and Carey/Bendick [11] employ a scenario-like object as an incremental unit for test and integration. This work strongly suggests a common value of the scenario object is its usefulness in describing system behavior in application terms across multiple lifecycle phases.

The dominating property of the scenario is its user understandable informality, i.e., its semantic properties are in communicative application language terms. I am biased in favoring methods that will enhance communications between non-computer applications experts, such as users and customers, and the software engineers who are developing the system. Both my own project experience and a key field study by Curtis, Krasner, and Iscoe [12] illuminate three key management problems in large software system development (in terms of additional effort or mistakes attributed to them): 1) the thin spread of application domain knowledge, 2) fluctuating and conflicting requirements, and 3) communication and coordination breakdowns. Thus, we are seeing emerging concern over the underlying behavioral aspects of those who sponsor and create software as well as the technical process models of software development. The communicative properties of the user oriented scenario object has appeal to both the behavioral and technical process issues of large scale software engineering.

A scenario schema can be converted into a formal mathematical object, such as a Petri net, with minor effort if formal analysis and/or execution is required. In the case of a Petri net, each scenario roughly represents a transition and contains enough information to define the enabling and firing conditions.

More concisely, we can say that a scenario is an object that:

- Connects an external stimulus with an external response to depict a user perceivable behavior path;
- Contains data events, control events, and conditions;
- Is both a specification and an implementation;
- Can be implemented in either hardware, software, by manual actions, or a combination.

Regarding the first point, the scenario is frequently a composite object that performs an operation on behalf of one or more concrete objects in the external environment and then operates back upon one or more of those concrete objects in the external environment. The response may be an overt data or control operation as in cruise control scenario 6, or may be reflected as a change to the condition of the environment objects as in cruise control scenario 8.

Both a stimulus and response contain data events, control events, and conditions or a subset as shown in Figure 4. Data events are simply traditional inputs and outputs. A control event can be either a signal, such as "driver starts engine," or the initiation/cessation of a process such as "monitor brake status" or "deactivate monitoring." A condition represents a "snapshot" of the temporal status of each external object and, perhaps, internal objects, that play a role in the scenario. Thus, the parallel conditions of multiple relevant objects are represented. The composite of the conditions represent the state of the system at the moment of the stimulus and the changed state at the moment of the response; "state" is used here as an informal term and not in any hard mathematical sense. The relationship of the response in a predecessor scenario with the stimulus in successor scenarios influences the concurrency structure of event driven systems. For example in the cruise control system, scenario 6 is a possible successor to scenario 3 (see Figure 2). The control system must have achieved a "maintain" condition for scenario 6 to operate (see Table 1) which could occur in the response in scenario 3 indicating a serial dependency. However, scenario 3 has initiated several "monitoring" processes which continue to operate concurrently with scenario 6.

A one-to-one correspondence is preferred between the specification (a scenario) and an implementing structural element such as an Ada package; we can at this level hide further lower level implementing details and deal with them only when necessary. If hardware or non-Ada software modules are involved, then the modules of the thread should be a

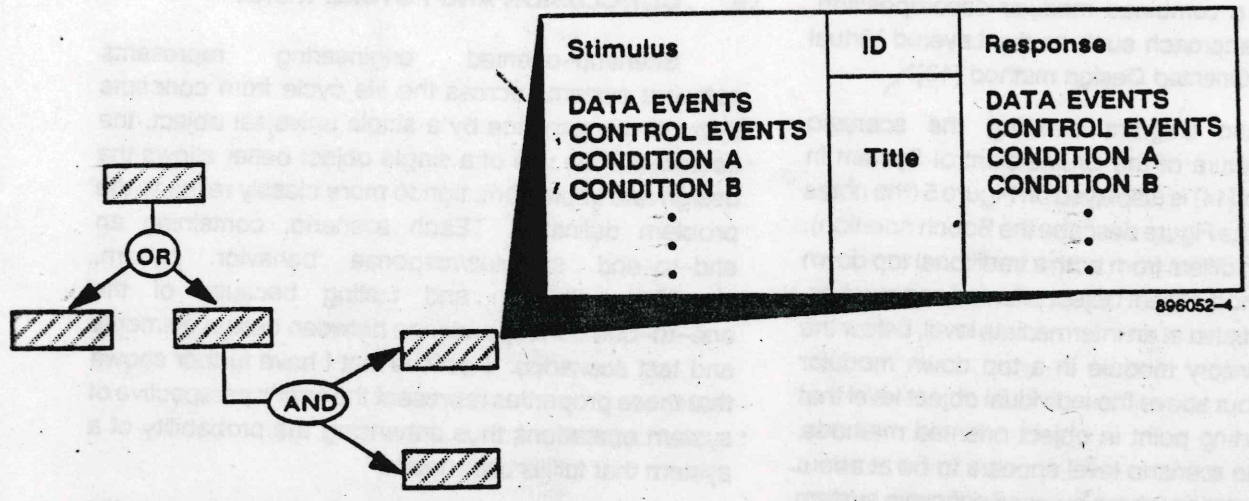

FIGURE 4. CONTENTS OF SCENARIO

member of a single scenario whenever possible. The guiding system engineering principle is to attain a direct correspondence between system architecture and operational usage scenarios.

A scenario may be wholly or partially implemented by manual actions. An allocation of functional behavior to the manual domain may yield derived requirements for software or hardware support. The interaction of all three domains must be factored into system validation. The scenario object provides significant support to these system engineering considerations.

5. ADA AND SCENARIO ORIENTED ENGINEERING

The benefits of Ada to the process of scenario-oriented software engineering (Section 3) are illuminated here by proceeding further with the design of the example Cruise Control system. The Ada package permits the incorporation of all design resources required to depict a scenario, i.e., abstract data, procedures, tasks, and functions. Thus, each high level design element directly represents a behavior pattern in the user's perspective. Further, each package can also be reviewed as representing a test scenario.

The heuristics for creating a scenario-oriented design in Ada are somewhat intuitive and relate to a natural problem solving sequence:

1) Define a package specification for each scenario including inter-scenario communication plus identification of serial procedures, parallel tasks, and abstract data.

2) Define a package specification for a system controller, or set of controllers, that provides procedural control between scenarios and within scenarios; the major control configuration options were shown on Figure 3.

3) Validate this architecture design by "threading" the stimulus/response scenario specifications into the Ada design components.

4) Define package bodies that detail the design of each package specification and repeat validation.

5) Develop lower levels of design down to the executable code level and validate by testing according to the original scenarios.

This approach inherently provides early attention to system control, both within scenarios and across scenarios; this attribute is especially important to the architecture design of embedded event-driven systems.

The design of lower levels of organization below the package scenario level does require a different strategy. One possibility is to decompose each scenario package into interacting objects, each represented by a package. The lower level objects thus "inherit" the properties of one or more higher level scenarios. Another possibility is to further decompose the scenarios using a combined modular decomposition, object oriented approach such as the Layered Virtual Machine/Object Oriented Design method [13].

A schematic diagram showing the scenario package architecture of the Cruise Control System in Booch's notation [14] is displayed on Figure 5 (the notes at the bottom of the Figure describe the Booch notation). This organization differs from both a traditional top down design and the more recent object oriented approaches. This design is initiated at an intermediate level, below the top level supervisory module in a top down modular decomposition but above the individual object level that is the usual starting point in object oriented methods. This intermediate scenario level appears to be at about the level of abstraction where humans conceive system behavior thus becoming a more natural fit to the way people solve problems. The design from the scenario packages then proceed upward to the controller level and then downward to either individual object packages or to more granular procedure subprograms whose details are hidden within the higher level procedures embedded in the scenario packages. An interesting aspect of the design diagram on Figure 5 is that scenario packages 6 and 9 "use" package 5. This is because all three have identical responses.

This section constitutes an embryonic description of a scenario oriented engineering approach as implemented by Ada. Considerable further conceptual work is required particularly in the non-trivial area of lower level design organization methods. Ada thus offers a design component, the package, that can naturally encapsulate a stimulus/response scenario pattern, a feature not easily available in other operational high level languages. The overall benefit is easier validation and testing because of the one-to-one correspondence between user perceived stimulus/response behavior scenarios, the top level design components, and test scenarios.

What is an ADA package?

6. CONCLUSION AND FUTURE WORK

Scenario-oriented engineering represents software systems across the life cycle from concepts through maintenance by a single universal object, the *scenario*. The use of a single object better allows the design and implementation to more closely relate to the problem definition. Each scenario, containing an end-to-end stimulus/response behavior pattern, simplifies validation and testing because of the one-to-one correspondence between design elements and test scenarios. I believe that I have further shown that these properties represent the user's perspective of system operations thus enhancing the probability of a system that fulfills user needs.

Significant additional work, including some in the research domain, is needed before the concepts promulgated in this paper can be realized in a practical dimension. Use of scenarios as the design elements within distributed system architectures would hold particular promise if it were possible to hide from the designer the details of processor allocation.

Significantly more intelligent distributed operating systems and support software are needed before the preceding advantage can be achieved. Barbacci [15] is presently developing a task description language tool for heterogeneous distributed machines that would be contributory to the scenario concept. Algorithms for automating selection of parallelism within loosely distributed and closely coupled distributed architectures, such as supercomputers, are a major component of this overall issue. Furthermore, it is possible to envision design semantics more powerful that those provided by Ada that are a better fit to the scenario orientation. Dannenberg [10], as an example, has contributed to research in this area through the design of the Arctic real-time language whose basic semantical construct is based on a cause/effect relationship.

Need this

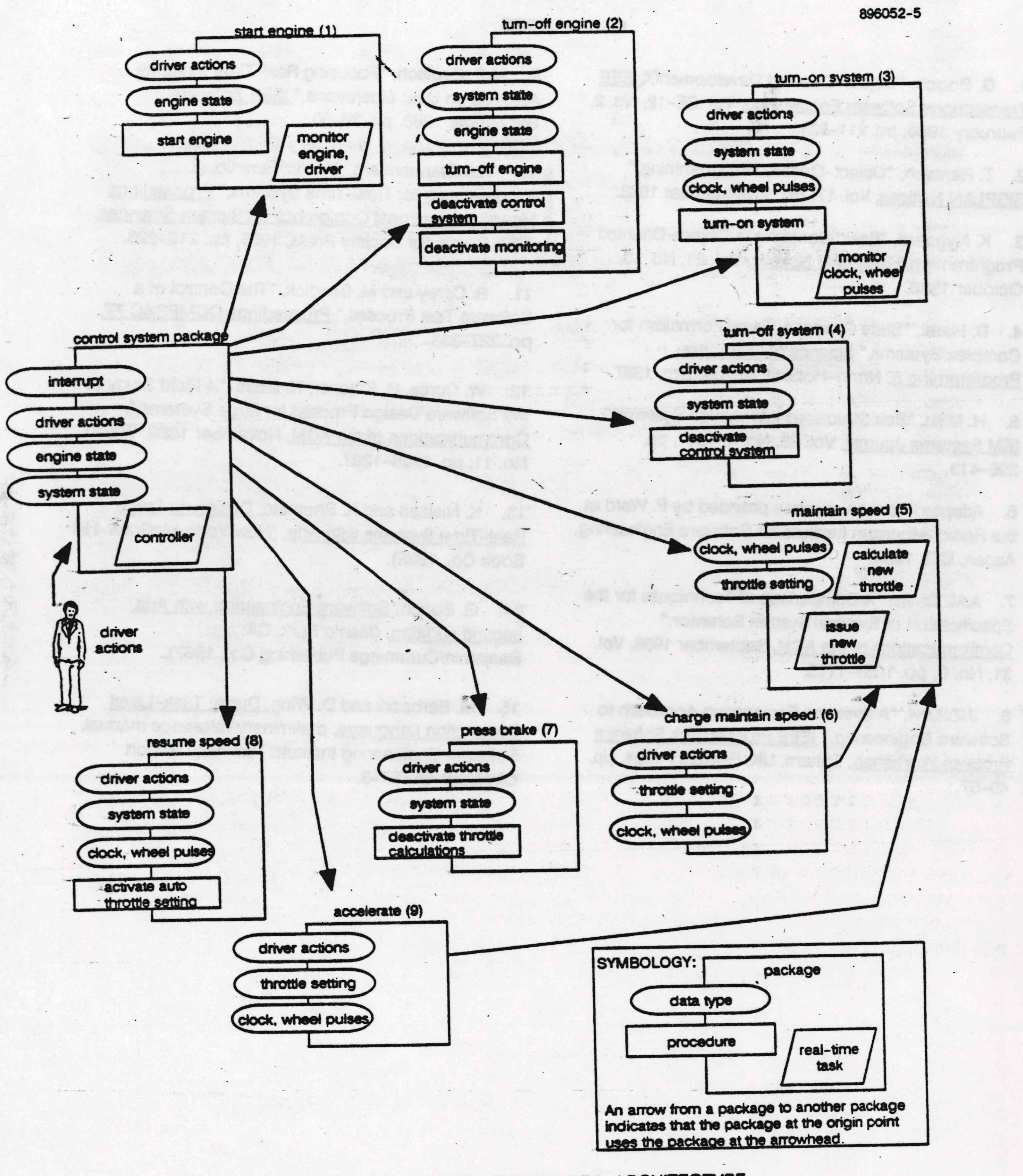

FIGURE 5. CRUISE CONTROL SYSTEM ADA ARCHITECTURE

REFERENCES

1. G. Booch, "Object-Oriented Development," IEEE Transactions Software Engineering, Vol. SE-12, No. 2, February 1986, pp. 211-221.

2. T. Rentsch, "Object-Oriented Programming," SIGPLAN Notices, Vol. 17, No. 9, September 1982.

3. K. Nygaard, "Basic Concepts in Object-Oriented Programming," SIGPLAN Notices, Vol. 21, No. 10, October 1986.

4. D. Harel, "State Charts: A Visual Formalism for Complex Systems," Science of Computer Programming 8, North-Holland, Amsterdam, 1987.

5. H. Mills, "Box Structured Information Systems," IBM Systems Journal, Vol. 26, No. 4, 1987, pp. 395-413.

6. Adapted from an exercise provided by P. Ward at the Rocky Mountain Institute for Software Engineering, Aspen, CO, 1984.

7. A.M. Davis, "A Comparison of Techniques for the Specification of External System Behavior," Communications of the ACM, September 1988, Vol. 31, No. 9, pp. 1098-1115.

8. J.Z. Lavi, "A Systems Engineering Approach to Software Engineering," IEEE Proceedings Software Process Workshop, Egham, UK, February 1984, pp. 49-57.

9. M.S. Deutsch, "Focusing Real-Time Systems Analysis on User Operations," IEEE Software, September 1988, pp. 39-50.

10. R.B. Dannenberg, "Arctic: Functional Programming for Real-Time Systems," Proceedings Hawaii International Conference on System Sciences, IEEE Computer Society Press, 1986, pp. 216-226.

11. R. Carey and M. Bendick, "The Control of a Software Test Process," Proceedings COMPSAC 77, pp. 327-333.

12. W. Curtis, H. Krasner, N. Iscoe, "A Field Study of the Software Design Process for large Systems," Communications of the ACM, November 1988, Vol. 31, No. 11, pp. 1268-1287.

13. K. Nielsen and K. Shumate, Designing Large Real-Time Systems with Ada, (New York: McGraw-Hill Book Co., 1988).

14. G. Booch, Software Engineering with Ada, second addition, (Menlo Park, CA: Benjamin/Cummings Publishing Co., 1987).

15. M. Barbacci and D. Wing, Durra: Task-Level Description Language, preliminary reference manual, Software Engineering Institute Technical Report CMU/SEI-86-TR-3.