Specification Languages for Embedded Systems: a Survey

Bonnie E. Melhart[1]
Department of Information and Computer Science
University of California, Irvine

Technical Report #88-17
June 29, 1988

[1]This work was supported by the Office of Research and Graduate Studies, University of California, Irvine.

Abstract

Requirements specification is an important part of the software development process. Use of well developed techniques, tools, and languages during requirements specification is especially crucial for complex embedded software systems. Four languages appropriate for the specification of software requirements for complex embedded systems (RSL, PAISLey, Statecharts, and SCR) are reviewed in detail here. In addition, other representation languages with features relevant to the embedded software systems domain are mentioned. Conclusions about the current status of embedded systems requirements specification and indications of further research are given.

Contents

1 Introduction

In any development process, it is essential to understand *what* you are building. This defining activity should take place before you start building, not halfway through the development process when it is too late or too expensive to make changes. A clear statement of the problem and potential solution is required for analysis and to provide a well-documented path for change and/or correction when modifications are deemed necessary. This is true for software engineering as for other system building processes. The requirements specification phase is essential to the progress of the entire software development process. It is at this time that the definition of *what* the system is required to do is established. If the requirements are specified correctly, the rest of development is made easier and cheaper. However, if the requirements are incorrect, the rest is harder and costlier. It is no simple task to get the requirements correct; indeed, it is not known how to determine whether or not they are correct. To this end, an important research area in software engineering is the development of requirements analysis techniques.

The general purpose of specifying requirements is to facilitate the understanding of the system that is under development. The goal of the endeavor is an explicit statement of the behavioral requirements for the new system, where behavioral may include both functional and non-functional (sometimes called performance) requirements. It is important that these requirements be precisely documented, not just expressed in vague terms or entrusted to someone's memory. Otherwise, at each stage of development, the requirements must be intuitively recreated for application at that level—an imprecise procedure at best [Par77]. The specification statement

should be independent of any particular implementation, and certain quality characteristics are called for: it must be possible to tell whether or not a system meets the requirements; two different systems with different functionality cannot both satisfy the requirements; internal consistency and consistency with the system's environment are required; and verifiable correctness, which includes completeness, is needed. Such a specification document is often used as a basis for contractual negotiation between the user and the developer. It also provides a *blueprint* for the next level of software development and may be used in later phases as a guide for change control.

The research discussed here is focused on the specification of requirements for complex embedded systems. Embedded systems, for our purposes, are those that are part of a larger system and have a primary purpose other than computation. Appropriate applications for this work involve process control and are characterized by concurrency, complex interfaces, and urgent performance requirements [Zav82]. There is a high cost associated with determining the correctness of such software and a still higher cost associated with its incorrectness [WL85].

Software engineering is involved in the development of many complex embedded systems, e.g., air traffic control, on-board aircraft flight control, robots, military weapons, and medical patient monitoring. Most real-time, reactive systems (i.e., those with requirements to interact with and respond to their environment during execution) will benefit from research on embedded system specification. One essential feature of such a specification is incorporation of an environmental (or plant) model. Validation of this model may be required before it is used to draw conclusions about the representation of the system that is being developed. The interface of the

embedded system with its environment and the assumed constraints on the environment should be a part of the requirements specification. Analysis for correctness of the system's specified response to external stimuli is facilitated by the inclusion of this environmental model in the specification.

Boebert described the development of embedded software systems as "One of the most rapidly growing areas of software activity and one whose characteristics and special problems are rarely considered by the computer scientist or verification communities." [Boe80]. While there are some popular requirements specification languages that may be used for less complex software (see e.g., [Egg80], [CL81], [TH77], [GMT*80], [BCF*83]), there are currently few languages that are seriously considered appropriate for specifying the requirements for complex, embedded systems. It is generally agreed that any such representation language should be formal (or at the very least semi-formal) for several reasons. The discipline required to use a formal language implies a more complete treatment of the specification. Formal models are mathematically analyzable and may even lead to automated verification. The complexity and size of most elements of this domain require considerable modularization of systems, which may increase the difficult task of interface specification; formal treatments facilitate the correctness of these specifications [WL85].

Of the available languages, the following four are best known: RSL, part of the SREM methodology for real-time specifications developed in the 1970's; PAISLey, developed by Zave in the early 1980's for the specification of embedded systems from an operational viewpoint; Statecharts, a recent technique developed by Harel, Pnueli, and others for specifying reactive systems; and a semi-formal technique developed by Parnas, Heninger, and others for specifying the A-7 aircraft flight software, referred to in this work

as SCR.

The purpose of this paper is to provide an overview of the languages for specifying embedded systems and assess how they may be extended or augmented with other techniques to better serve the developers of embedded software systems. Progress in the area will best be promoted by careful consideration of the existing methods as a starting point for further work. This is supported by Place: "We must be wary of performing a too rapid examination of current techniques and concluding that there is no appropriate technique." [Pla85], and by Shaw: "It is extremely desirable to extend existing methods to new properties instead of developing new methods whenever possible." [Sha85]. The organization of this paper is based on this premise. After a discussion of certain standards desirable for requirements specification languages, the four previously enumerated languages (RSL, PAISLey, Statecharts, and SCR) will be surveyed. For each language, the context for its development/use, an overview of the language itself, experience with its use, and as assessment of its effectiveness for the embedded systems domain will be given. Some other less known/used languages will be mentioned in section 7, although they are not fully reviewed. Finally the current status and research areas of embedded systems requirements specification will be assessed.

2 Desirable Characteristics

While there is no absolute checklist for properties that a requirements specification must have to be good, useful, or even to satisfy the purposes set forth in the previous section, there are certain characteristics that are desirable in a software specification. Some are not only desirable, but essential according to the intended application area. Alford [Alf77] has provided an extensive list of such properties; Heninger [Hen80] has described more desirable characteristics specific to real-time applications. Their features and a few other generally accepted ones are listed in figure 1.

Correctness properties. No matter what the application domain for the software specification, it should be as correct as possible. Correctness is an evaluation of whether or not it solves the problem as put forth by the system requirements that have been allocated to the software subsystem or, in a less complex situation, as put forth by the user to the software developer.

Completeness addresses whether or not the specified system is sufficient to this required task. There is no absolute criterion for establishing the completeness of a software specification; ongoing research is addressing this area [Jaf88] [Yue87]. Two kinds of consistency are desired for a specification: static and dynamic. Static consistency means the statement of the requirements has only well defined entities, whereas dynamic consistency implies the functioning software system is well-defined. This is usually measured by simulation.

Two other correctness features are closely related. To be precise a specification must not be satisfiable by more that one distinct system; a specification is ambiguous if it is not possible to tell whether or not a system

```
Correctness properties.
      complete
      consistent
      precise
      unambiguous
      formal

Verification properties.
      traceable
      testable

Correction/modification properties.
      changeable
      extensible
      free of design

User properties.
      executable
      communicable
      usable

Real-time/embedded properties.
      address:  environment
                exception handling
                performance
```

Figure 1.

satisfies the specification. The discipline of a formal specification may be desired to achieve the other properties. Correctness properties are desirable for all specifications.

Verification properties. The next two properties on the list relate to verification of the specification and of the implementation of the specification. A traceable specification relates each requirement to the preliminary requirements from which it arose, either a user statement or a sub-system specification. A testable specification states requirements in terms of the tests that can be used to verify compliance to it.

Correction/modification properties. Correction or modification properties are desirable when the requirements for the system must be changed due to errors or otherwise. Some have argued that all design decisions should be kept out of the requirements specification [Par77], while others

argue for postponing design decisions as long as possible, but not precluding some design decisions from the specification [Hen80].

A changeable specification allows minor corrections without redoing major pieces of the work; modularity is usually a technique to encourage a changeable specification. Requirements specifications that are extensible are especially desirable for applications that are dynamic as frequent changes in the application mean frequent changes in the software capability.

User properties. The requirements specification is a document for many users. Maintainers, for example, may desire a document organized as a reference tool. The inclusion of a graphical model may be desired for communication of the representation among analysts and designers or even communication to the customer (user). Some may desire an executable specification for simulation demonstrations or to build a prototype model. As with correctness, formality is desirable to help achieve an executable specification.

Real-time, embedded properties. The last set of features are key ones for the application area, i.e. complex embedded systems. The inclusion of an environment model with the software subsystem representation facilitates the specification of interfaces and of any assumptions made about inputs and outputs. Specification of non-functional constraints including performance requirements such as response time, space bounds, time outs, accuracy, and reliability as well as communication, synchronization, and security are often essential to the development of systems in this domain. In particular, safety critical system specifications demand explicit stipulation of how the occurrence of undesired events will be handled.

Of course, these properties are not without cost (in personnel, time, dollars, or all three), and their desirability is a balance between that cost

and the benefit they provide for a particular application. Specification languages and the methods for using them may incorporate features that demand, encourage, or allow such desirable properties in the resultant specification. As the details of selected languages are given in ensuing sections, care will be taken to establish whether or not the language has such features.

3 RSL

The first language to be described is the Requirements Statement Language (RSL). RSL was developed in the 1970s by the Ballistic Missile Division of TRW as a part of its Software Requirements Engineering Methodology (SREM). This description of RSL begins with a discussion of SREM and its major parts. An overview of RSL is given, and the major experiences with its use are related. These are followed by an assessment of the usefulness of this specification language in the SREM context. References for RSL and SREM are [Alf77], [BBD77], [Alf85].

3.1 Context

The intended arena for the use of RSL is within SREM. This methodology came about as the result of frustration felt by the defense industry with the *hierarchy of functionality* model used for requirements specification. In this model functions are decomposed into sub-functions which are in turn decomposed until the units are of desirable size. Frequently, a particular design is implied by the specification as subroutines are used to deliver sub-functionality. The model lacks the capability to represent conditions and sequences of processing and often leads to a specification that is ambiguous and difficult to test [Alf85].

The developers of SREM wanted a methodology that dealt with the technical issues (such as work-products to be produced, language support, the form of the requirements) and the management issues (such as scheduling and evaluation) that are essential to the development of software requirements. Their hope was that inclusion of these aspects into a formal, thorough methodology might result in reduced life cycle time and cost.

SREM is made up of four parts: RSL, the language used to represent the software requirements; Requirements Engineering and Validation System (REVS), a set of tools for maintaining, manipulating, and analyzing the requirements; a relational database called the Abstract System Semantic Model (ASSM); and a methodology for performing the requirements specification task.

REVS is responsible for information contained in the database and for analysis of the evolving requirements representation. It has many capabilities. REVS interprets and translates the RSL statements for incorporation into the ASSM, which it maintains. Included in the tools is an interactive graphics package for graphical description of R-nets and subnets (described in section 3.2 of this paper). The graphical representation is equivalent to, and interchangeable with, the formal language representation. REVS also has static analysis capability for checking consistency, completeness, and correctness of intermediate products as well as the complete specification. There is a simulation generator included that can provide discrete event simulation of the functional model processing steps or an analytical simulation using Pascal algorithms. These algorithms are supplied by the requirements engineer; the intent is that they be similar to those that will eventually be used.

Another important part of REVS is the extraction and reporting capability used to recover and report information from the ASSM that is useful to requirements engineers and to management. As will be discussed in the next subsection, RSL can be extended to meet application needs; any such extensions are supported and maintained by the tools in REVS. REVS is a tool set with a broad range of capabilities.

The relational data base, ASSM, maintained for SREM is a central

repository for all the requirements of the software system being developed. Large systems may be subdivided with subsection responsibilities assigned to many different teams. All the requirements specified by all teams are kept in the one data base for the software system. ASSM contains any extensions added to RSL for the application as well. Representations are kept for each primitive type defined in the system; instances of these are linked back to their associated primitives. Thus the data base facilitates tool support and extensibility.

SREM is intended for systems that are predominantly stimulus-response (S-R), having a required set of actions, possibly ordered, for a particular input. To establish these requirements, the methodology follows eight steps. The steps are not necessarily independent of one another in content or time. The assumption is made that the overall system requirements have been allocated so that preliminary requirements exist for the software subsystem.

1. Define the kernel elements consisting of input messages, output messages, basic processing nodes, and requirements networks using RSL.

2. Establish the baseline database, including plots of the requirements nets, and do checking for completeness and consistency.

3. Assign the data inputs and outputs for the processing nodes.

4. Establish the traceability of requirements to and from preliminary requirements, determine validation points for performance requirements, and their traceability.

5. Perform functional simulation by REVS. Specially defined *artificial data* may be used for simulation inputs, or a simulation of the proposed system's environment can provide the interface data.

6. Identify testable performance requirements; determine appropriate validation points for these, and establish their traceability.

7. Perform analytical simulation to demonstrate feasibility for critical algorithms.

8. Analyze the dynamic behavior for time delays (including time-outs), communication and synchronization procedures, and failure identification methods.

Originally this automated system was run on a Texas Instruments Advanced Scientific Computer. In 1981 the tools were moved to a Vax 11/780, and in 1983 the software managing ASSM was rewritten in Pascal to improve performance. By 1985, all of SREM had been extended to include specification of requirements at the system level through software specification. This front end extension addresses system requirement definition and allocation to subsystems. A state hierarchy is established that allows concurrent and/or sequential states.

Some work has also been done to establish a computing design system. This will add to SREM a transition phase for establishing the design specification from the requirements. The planned multipart phase includes allocation of processing to code units and data structures as well as interface decomposition and allocation in a distributed system. Fault tolerance functions may be incorporated and allocated. The design system ideas remain experimental. The ultimate goal is to extend RSL, REVS, and the methodology to a complete software engineering environment for embedded system development.

```
A patient monitoring program is required for a hospital.
Each patient is monitored by an analog device which
measures factors such as pulse, temperature, blood
pressure, and skin resistance.
The program reads these factors on a periodic basis
(specified for each patient) and stores these factors
in a data base.
For each patient, safe ranges for each factor are
specified (e.g. patient X's valid temperature range is
98 to 99.5 degrees Fahrenheit).
If a factor falls outside of the patient's safe range,
or if an analog device fails, the nurse's station is
notified.
```

Figure 2: [SMC74].

3.2 Overview

The Requirements Statement Language (RSL) for SREM is a flow-oriented language, i.e. the specified operations are expressed as flows through the processing system. It is a formal language, leading to reduced ambiguity and increased automatability. In fact RSL is machine processable. The system being developed is described in terms of stimulus-response using a highly structured finite state machine underlying model. This model allows the expression of the S-R relationships, promoting response and accuracy requirements as well as static analysis. To aid in the discussion of RSL, a simple example has been included. Preliminary requirements for a patient monitoring program taken from [SMC74] are given in figure 2, and some RSL requirements developed from these are in figures 3 and 4 (adapted from [Alf77]). Special considerations for system startup have been omitted here for simplicity.

RSL employs hierarchical conceptual networks called Requirements Nets (R-nets) to specify flows. Each R-net specifies processing flow by describing the transformations of an input to one or more outputs and also specifies the accompanying changes in the system state. Figure 3a shows the R-net

for input FROM_DEVICE. Basic nodes of these R-nets may be processing nodes, called alphas, or subnets which are R-nets at the next lower level in the network hierarchy. All basic nodes have a single entrance and a single exit point. Other nodes used in R-nets are structured nodes; they enable the description of conditional, parallel, and synchronized flow processing. The four types of structured nodes are:

AND- represents two or more mutually order independent paths with a synchronized end point.

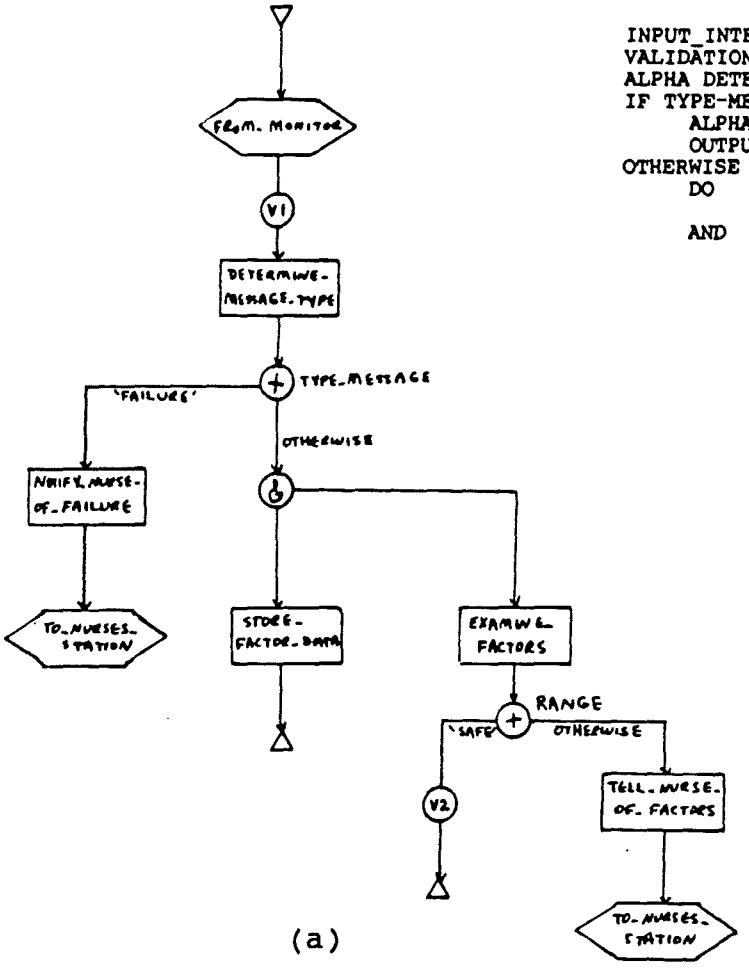
OR- represents conditional paths, each with an explicit condition. The first path with a true condition is taken, where first is determined implicitly or explicitly. An otherwise path is required by RSL for OR nodes.

FOREACH- represents iteration of a path for each element of a specified set.

SELECT- is similar to FOREACH except iteration is subject to an explicit conditional.

Figure 3 contains examples of AND and OR structured nodes.

The formal language of RSL describes the 2-dimensional R-nets in a 1-dimensional computer input. Four primitive building blocks are used to describe the system processing: elements, relationships, attributes, and structures. Elements are the specification objects such as functional processing units (alphas), conceptual data, and processing flow specifications (R-nets). A relationship specifies a certain non-commutative association between two elements; e.g., alpha EXAMINE_FACTORS inputs data SAFE_RANGES.




```
INPUT_INTERFACE FROM_DEVICE
VALIDATION_POINT V1
ALPHA DETERMINE_MESSAGE-TYPE
IF TYPE-MESSAGE IS FAILURE
     ALPHA NOTIFY_NURSE-OF-FAILURE
     OUTPUT-INTERFACE TO-NURSES-STATION
OTHERWISE
     DO
          ALPHA STORE-FACTOR-DATA
     AND
          ALPHA EXAMINE-FACTORS
          IF RANGE-SAFE
               VALIDATION_POINT V2
          OTHERWISE
               ALPHA TELL_NURSE_OF_FACTORS
               OUTPUT-INTERFACE TO-NURSES-
                                  STATION
```

(b)

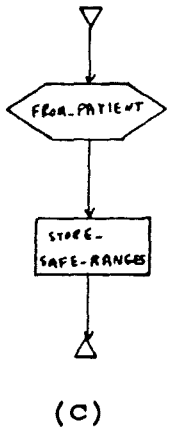


Figure 3.

Attributes specify properties of elements; e.g., HI_TEMPERATURE is an attribute of SAFE_RANGES. Among its attributes, each element has a description which is used to document its purpose. Structures specify the flow model in RSL. The graphical representation of the R-nets in figure 3a and 3c is equivalent to the formal language description in RSL, shown in figure 3b and in figure 4.

RSL is extensible; new elements, relationships, and attributes appropriate for a particular application may be defined and added to the language. Structures are not, however, extensible. The developers have provided a base language of primitives (21 elements, 23 relationships, 20 attributes) considered useful for all applications. The extension capability has a lock-out feature for controlling change as management may want to constrain casual additions.

It is possible to specify accuracy and response time requirements in RSL. Special nodes called validation points are placed along paths in the R-nets, used in figure 3a and 3b to determine how frequently readings are taken. Accuracy is then specified, and determined, by the state at a particular validation point. Response times are expressed by specifying maximum and/or minimum times to complete the path between two validation points. Both accuracy and response time specifications are given in terms of a test to avoid ambiguity.

3.3 Experience

Several experiences with the use of RSL have been reported. Two early attempts and a more recent case study are discussed here. The developers of SREM tried out their ideas for the methodology and RSL on a real-time

```
                    RSL Data Descriptions

ORIGINATING_REQ.:  INPUT_1
        DESCRIPTION:  "Defines analog device measurements."
        REQ. TRACES TO:  MESSAGE MONITOR_REPORT

MESSAGE:  MONITOR_REPORT
        HOW PASSED:  INPUT_INTERFACE FROM_MONITOR
        REQ. TRACES FROM:  ORIGINATING_REQ. INPUT_1
        MADE BY:  DATA MONITOR_ID,
                  DATA TYPE_MESSAGE,
                  DATA INPUT_DATA

DATA:  INPUT_DATA
        INCLUDES:  DATA PULSE,
                   DATA TEMPERATURE,
                   DATA BLOOD_PRESSURE,
                   DATA SKIN_RESISTANCE

ENTITY_CLASS:  PATIENT
        ASSOCIATES:  DATA PATIENT_ID,
                     DATA SAFE_RANGES,
                     FILE HISTORY

DATA:  SAFE_RANGES
        INCLUDES:  DATA LOW_PULSE, DATA HI_PULSE, DATA LOW_PRESSURE,
                   DATA HI_PRESSURE, DATA LOW_TEMPERATURE, DATA
                   HI_TEMPERATURE, DATA LOW_SKIN_RESISTANCE, DATA
                   HI_SKIN_RESISTANCE, DATA PATIENT_MONITOR_ID

FILE:  HISTORY
        CONTAINS:  DATA MEASUREMENT_TIME, DATA HPULSE, DATA HTEMP,
                   DATA HPRESSURE, DATA HRESISTANCE
        REQ. TRACES FROM:  "Preliminary statement"

ORIGINATING_REQ.:  INPUT_2
        DESCRIPTION:  "Defines safe factor ranges for each patient."
        REQ. TRACES TO:  MESSAGE PATIENT_REPORT

MESSAGE:  PATIENT_REPORT
        HOW PASSED:  INPUT_INTERFACE FROM_PATIENT
        REQ. TRACES FROM:  ORIGINATING_REQ. INPUT_2
        MADE BY:  DATA SAFE_RANGES

ALPHA:  DETERMINE_MESSAGE_TYPE
        INPUTS:  MONITOR_REPORT
        OUTPUTS:  TYPE_MESSAGE
        DESCRIPTION:  "Separates incoming message to discern type."

ALPHA:  NOTIFY_NURSE_OF_FAILURE
        INPUTS:  DATA MONITOR_ID
        OUTPUTS:  MESSAGE FAILED_MONITOR
        DESCRIPTION:  "Forms message to notify nurses station."

ALPHA:  STORE_FACTOR_DATA
        INPUTS:  DATA INPUT_DATA
        OUTPUTS:  None
        DESCRIPTION:  "Store the input in a data base for future reference."

ALPHA:  EXAMINE_FACTORS
        INPUTS:  DATA DEVICE_DATA
                 DATA SAFE_RANGES
        OUTPUTS:  RANGE
        DESCRIPTION:  "Compares device data to safe ranges and determines
                       if range is safe or not."

ALPHA:  STORE_SAFE_RANGES
        INPUTS:  PATIENT_REPORT
        OUTPUTS:  None
        DESCRIPTION:  "Store safe ranges for this patient monitor."
```

Figure 4.

test control. A software specification for the control was already available in another form, but it did not include some desired enhancements. The specification was rewritten in terms of stimulus-response relationships and paths, incorporating the extensive modifications for enhancement. Positive results were achieved: the new design-free specification required no change during subsequent design modification; time for requirements maintenance and change was cut in half.

The language was further tested on the tracking loop portion of the terminal defense program of an anti-ballistic missile. The specification was written in RSL; REVS was not available yet for automated analysis. The outcome of this trial indicated weaknesses in RSL for translating preliminary requirements and 2-dimensional R-nets. After upgrading the language, the specification was rewritten with a better result. Ambiguities were found, in particular, by having to specify performance in terms of a test. This completed specification was eventually used as a test case for the developed REVS system.

More recently, Martin Marietta Denver Aerospace undertook to test the applicability of RSL and SREM for expressing requirements for Command, Control, Communication, and Intelligence (C^3I) embedded systems [SSR85]. The reported study was done in the early 1980s as a follow up to a preliminary study, by Rome Air Development Center in the late 1970s, which determined SREM adequate and useful for C^3I. The purpose for the experiment was 3-fold: to study the ability of RSL/SREM to do C^3I, to study where in the Air Force life cycle SREM is useful, and to determine training needed for using the methodology.

Two systems with existing specifications for different levels of development were used. The first was the communication switch interface develop-

ment system which was part of an Air Force communications system and already had a software design specification. The other, the advanced sensor exploitation system for handling and distributing sensor data, was described in a software requirements specification. Both of these were translated into RSL and analyzed using REVS.

The case study results supported SREM as a viable approach for C^3I software requirements; the formal discipline of RSL resulted in early error detection. It was felt that the method is best applied to requirements specification, but it could and perhaps should be extended for use during design specification. However, there was some disappointment with RSL's capabilities for expressing parallel and distributed processing, and REVS was felt to be inefficient. Finally, the cost of learning to use the methodology was significant. It is unclear whether this is a reflection upon the RSL/SREM system in particular or the nature of requirements engineering methods in general.

3.4 Assessment

An assessment of the Requirements Statement Language is actually an assessment of SREM, as its only use is within that context. As previously mentioned, there are plans to extend SREM with design capabilities and eventually to create an entire software life-cycle development system.

At the current level of capability, there are positive and negative aspects to the language and methodology. On the positive side: consistency and some completeness verification is provided; performance specifications are testable; and the requirements are traceable. RSL is a formal language that encourages precise and unambiguous requirements; the extensibility of the language is certainly beneficial for the dynamic domain considered here,

and the simulation generators are equally valuable. With the centralized data base, change control and recording should be expedited. SREM is appropriate for the generation of real-time software requirements.

On the other hand, extensions should be provided for specification of other non-functional requirements such as fault tolerance, reliability, synchronization, time-outs, etc. Also, the performance requirements are not addressed in the methodology until step 6; they ought to be considered earlier in the specification process. The dynamic behavior analysis step in the methodology is not clearly defined. More work remains to be done here, especially for the sensitive dynamic application area of embedded systems. Additionally, the methodology does not address the explicit determination of requirements for undesired event handling, and no environmental representation is included.

The experiences of requirements engineers who have reported using SREM indicate a lot of effort expended during the requirements phase. This may be due to their unfamiliarity with stimulus-response networks versus their familiarity with hierarchy of function models, which as previously indicated are often not design-free.

4 PAISLey

The next language to be described is the Process-oriented, Applicative, and Interpretable Specification Language, better known simply as PAISLey. Most of the work on PAISLey was done by Pamela Zave at AT&T. The original paper described the language [Zav82]. This was followed by two other papers in which the context for the language was refined [Zav84], and further details of the language features and the environmental model were given [ZS86]. The discussion of PAISLey is in four parts: context, language overview, experience, and finally assessment of PAISLey.

4.1 Context

The context for PAISLey is the operational approach to software development. Here the conventional approach is viewed as top-down decomposition of black boxes organized around the problem solution. In contrast, the organization of a specification in the operational approach is based on the problem itself. Implementation independent structures are used to describe all mechanisms of the system. These may be viewed as virtual structures; they may or may not be present intact in the implementation. Optimizations are accomplished during the development process. Operational languages are formal which leads to a machine processable representation, i.e., the resultant requirements specification is executable. While it is not a prototype, a useful prototype can often be rapidly generated from the specification. Efficiency is not a concern of the specification process, but rather the conversion process from the specification to an implementation. The emphasis of the approach is on constructing an operating model of the system functioning in and interacting with its environment. PAIS-

Ley, GIST [BCF*83], and Jackson System Development (JSD) [Jac83] are instances of the operational approach. PAISLey is intended for embedded systems, while JSD is best applied to data processing systems and GIST does not include apparatus for real-time systems.

Also part of the context for PAISLey are transformational methods. The intent is for requirements analysis and definition to be followed by a transformational phase. The formal PAISLey specification will be subjected to transformations that preserve the external behavior of the specification description and yield an implementation-oriented specification, indeed ultimately result in an implementation. The hope is for automation of the transformation phase with human intervention to guide the process. This may turn out to be a very complex human task. Examples of transformations that may be performed are structure movement (for modifiability and comprehensibility) and additions. For example, whereas the intermediate results of a Fibonacci series calculation may be recomputed in the original requirements specification, the specification may be transformed so they are saved for efficiency reasons rather than recomputed. As another example, a resource allocation mechanism that was not needed for specification might be added during transformation. Similarly, the example in figure 8 enlists several monitor processes that might be combined through transformations into one that splits cycles.

4.2 Overview

PAISLey is actually a set of tools embedded in the UNIX operating system (see figure 5). The parser accepts a specification written in PAISLey and forms the internal specification representation. The interpreter tool is

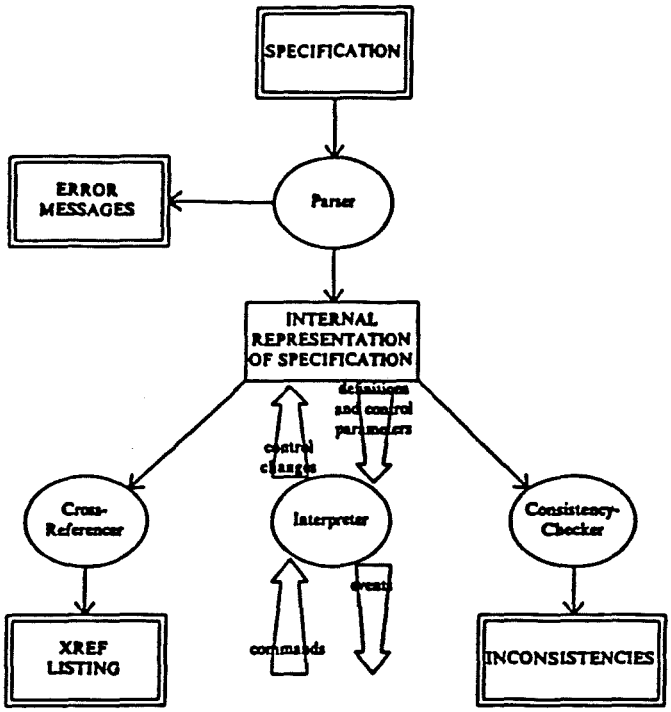


Figure 5: [ZS86].

an important one; it is interactive and is responsible for the display and any break points. The consistency-checker verifies domains and ranges and checks timing constraints for system feasibility. This tool was not implemented as of the 1986 writing, but seems to be present for the technology transfer experiment reported in 1987 [Zav82].

A system is declared in PAISLey as a set of asynchronous processes. Each process is specified by giving a state space (domain and range of the defining function) and a successor function. Processes are cyclic; i.e., their mapping functions continue to be evaluated forever. The size of the system structures is fixed: there are a bounded number of processes, no state can require unbounded store, and no step can take unbounded time.

The process evaluation steps give the language a state orientation. Each process step amounts to the evaluation of the successor function of one of the system's asynchronous processes. Process states occur before the mapping evaluation and after; the process is in a computation phase between states.

A system environment model made up of one or more asynchronous processes is included with the embedded system model. The aim is to provide an explicit model of the proposed system interacting with an explicit model of the system environment. Processes in the environment model may be people, machines, other programs, etc.

The forced discipline of a formal language reduces the number of errors that are retained in the representation. Execution of the formal specification is tolerant of incompleteness in the system description. This incompleteness may take the form of undefined mappings; the system declaration and mapping declarations must be present, however. There are three ways for an incomplete mapping to be handled. The first is by defining a default

```
                4 TYPES OF STATEMENTS

System declarations: tuples
          e.g. ( monitor-1[data],
                 monitor-2[data],
                 update-cycle-1[db],
                 update-cycle-2[db],
                 nurses-station)

Functional declarations:  mappings
          e.g.  update-cycle-1:  DATA-BASE --> DATA-BASE
                get-data-j:  --> DEVICE-DATA

Set definitions:  expressions (union, cross
                     product, enumeration)
          e.g.  DEVICE-DATA = TYPE X INPUT-DATA
                PULSE = REAL

Functional definitions:  expressions (constants,
                     composition, tuples,
                     cond. selection)
          e.g.  new-func-1 = /p1:f1, p2:f2,
                           'true':f3/
```

Figure 6.

value to be used whenever an undefined mapping is encountered. Another is for the interpreter to select a value from the function range at random. Finally, the function value may be requested and supplied interactively. In fact, the interpreter could be instructed to consult a function written in the programming language C when the undefined mapping is encountered.

Executable specification capability facilitates validation in several ways. Demonstrations may be provided for users to validate the external behavior of the system; in some cases, the executable specification may be extendible to a prototype. Furthermore, the behavior of the final system may be compared to the behavior of the executing specification during acceptance testing.

The PAISLey language is composed of applicative statements and special functions called exchange functions. Functions are used to describe relationships and are not procedural. Figure 6 shows the four types of

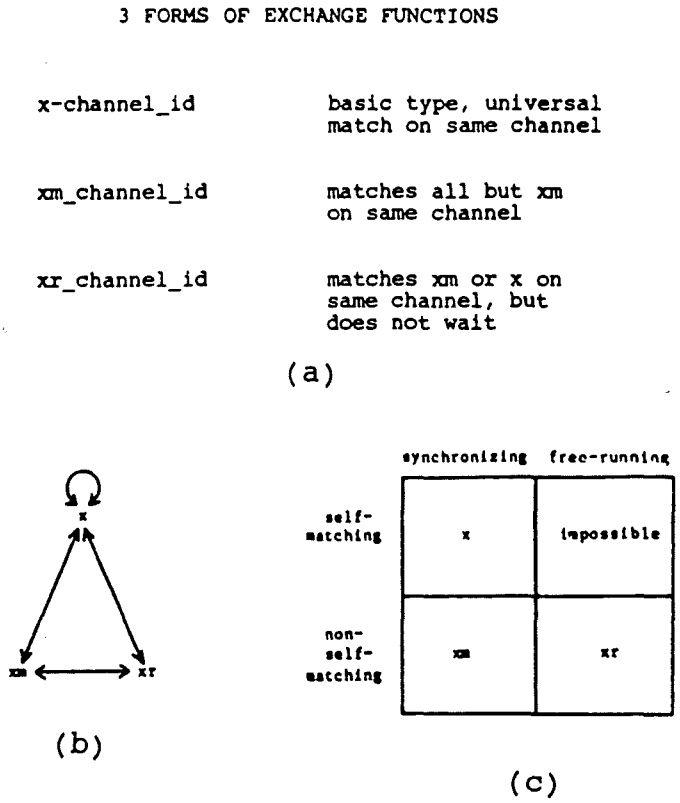


Figure 7: [Za82].

statements in the language. A functional declaration statement gives a domain (optional) and a range for a mapping. Tuples in the functional definitions are executed in parallel. Most functions are side-effect-free.

Exchange functions bind the asynchronous processes together. They may be viewed as ordinary mappings at the local level, except that they have side-effects. From the global point of view, they provide two-way mutually synchronized, interprocess communication. The three forms of exchange functions are listed in figure 7a, where channel-id is a user supplied label. Figure 7b further depicts the possible exchange function interactions,

while figure 7c points out the possible correspondence between process synchronization type and synchronization mechanism type. It is impossible to have a free-running process match with another free- running process, because they never wait and cannot both be in the same channel at the same time.

An example of PAISLey in use is given in figure 8. These requirements describe the patient monitoring program in figure 2. Note the use of the exchange functions x and xm for communication on channel nurse-needed, and x and xr on channels new-safe-ranges-j and read-data-j. Due to the bounded system requirement for PAISLey, the example assumes ten patients, each with separate monitors.

It is possible to include some performance constraints for the proposed system in PAISLey. Timing constraints may be defined for individual functions; these are for simulated time, however, not real time. Scheduling is done by the interpreter in a top-down fashion, so inherited constraints are implied by this procedure. Upper bound, lower bound, distribution mean, or combinations of these are possible timing constraints. Reliability constraints have not yet been implemented but plans do exist for constraints such as required ranges for probability of success for given functions. Current syntax is unavailable for both timing and reliability performance constraints; the periodic requirement for the monitor reading by updata-cycle-j is given as a comment in figure 8.

4.3 Experience

Reference has been made to three different early experiences with PAISLey. A finite-element system for partial differential equations was specified using the language [ZS86]. At the time of the experiment, the interpreter tool was

```
"Definitions"
        SAFE-RANGES = LOW-PULSE X HI-PULSE X LOW-PRESSURE X HI-PRESSURE X
                      LOW-TEMPERATURE X HI-TEMPERATURE X LOW-SKIN-RESISTANCE
                      X HI-SKIN RESISTANCE;
        DEVICE-DATA = TYPE X INPUT-DATA;
        TYPE = ('failure','success');
        INPUT-DATA = PULSE X TEMPERATURE X BLOOD-PRESSURE X SKIN-RESISTANCE;
        PULSE, TEMPERATURE = REAL;
        SKIN-RESISTANCE = INTEGER;
        BLOOD-PRESSURE = INTEGER X INTEGER;
        RANGE = ('safe','unsafe');

"System declaration"
        (j#1..10 <, monitor-j[data],
                    update-cycle-j[db] >,
                    nurses-station);

"Environment"
        nurses-station:  --> FILLER;
        nurses-station = "Contains x-nurse-needed, and xm-new-safe-ranges-j
                         for j=1..10";
        j#1..10 <; monitor-j:  DEVICE-DATA --> DEVICE-DATA;
                   monitor-j = "Contains xr-read-data-j[data].
                        Data is continuously available; however, for simulation
                        purposes will have to model as a discrete process." >;

"Functional declarations and definitions"
        j#1..10 <; update-cycle-j:  DATA-BASE --> DATA-BASE;
                   update-cycle-j:  ! --> "Must repeat every patient-j-time
                                                        seconds";
                   updata-cycle-j[db] =
                        proj[1,(process-msg-j[db,get-data-j])];
                   get-data-j:  --> DEVICE-DATA;
                   get-data-j = xm-read-data-j[from-monitor];
                   process-msg-j:  DATA-BASE X DEVICE-DATA -->
                                                   DATA-BASE X DEVICE-DATA;
                   process-msg-j[db,d] =
                        /proj[1,(d)]='failure':  send-warning-j,
                         'true':  process-data-j[db,d]/;
                   send-warning-j:  --> FILLER;
                   send-warning-j = xm-nurse-needed['failure', j];
                   process-data-j:  DATA-BASE X DEVICE-DATA -->
                                                   DATA-BASE X DEVICE-DATA;
                   process-data-j[db,d] = proj[2,(store-data-j[db,d],
                                examine-data-j
                                     [compare[ck-new-safe-ranges[db],d]])];
                   ck-new-safe-ranges:  DATA-BASE --> DATA-BASE;
                   ck-new-safe-ranges[db] = /xr-new-safe-ranges-j
                                                [new-range]<>'null':
                                update-ranges[db,new-range], 'true': db/;
                   compare-j:  DATA-BASE X DEVICE-DATA -->
                                RANGE X DATA-BASE X DEVICE-DATA;
                   compare-j[db,d] = "Compare input to stored information
                                and determine if range is safe or not.";
                   examine-data-j:  RANGE X DATA-BASE X DEVICE-DATA -->
                                        DATA-BASE X DEVICE-DATA;
                   examine-data-j[r,db,d] = /r='unsafe':
                        xm-nurse-needed['unsafe',d,j], 'true':  [db,d]/;
                   store-data-j:  DATA-BASE X DEVICE-DATA --> FILLER;
                   store-data-j[db,d] = "Store device data in current
                                                data base."       >;
                   update-ranges:  DATA-BASE X SAFE-RANGES --> DATA-BASE.
```

Figure 8.

not completed so the specification was translated into Fortran for execution. Reportedly the experience turned out well. Workshops in using PAISLey have been conducted at AT&T to gain perception into and experience with the use of the language. Some participants were enthusiastic about the language, however a major complaint was that the functional notation is difficult to learn and difficult to use. Work is in progress on an experience with robot-based factory stations to automate the testing of lightwave diode chips. The goal for this work is to maximize station throughput.

Insight was gained from the experience of specifying a portion of the user interface of SALT at AT&T [BZ87]. There were two stated purposes for the experiment. The goal of those working on the SALT system was to capture the requirements of the system so that the subsystem interfaces and the user interface were clarified, documented, and validated. The PAISLey purpose was to test the success of doing a specification in PAISLey, to measure the success professionals have with learning to read and write the language, and to judge the quality and productivity results with PAISLey. The project selected is a part of the Undersea Lightwave Cable System. The communication system controlling transmission for the cable consists of hardware and a computer system called SCOUT which has 17 processes, one of which is the user interface. SALT is the computer system used when SL is off line; it therefore has much overlap with SCOUT. Part of the user interface for SALT was specified in PAISLey. The experiment included a workshop, prototype specification (to explore the suitability of project), informal training, formal specification, review, and demonstration.

This experiment provided information about PAISLey that is both negative and positive. On the negative side, there was a poor language-to-problem match in that the user interface is basically a sequential appli-

cation. There is a high cost of adoption for PAISLey; much time and frustration was spent learning to use the language. Exceptions were poorly handled and had to be specified informally. No modularity is available in PAISLey, and this amplified the complexity of doing the specification. Since all names are global, a naming convention had to be used to refer to common subfunctions in the system.

On the positive side, the demonstration capability worked out very well. Also, the formality of the language was important for reducing errors. There was a substantial "fear of the new" when the experiment was begun, and this was overcome with management commitment, hard work, earned credibility, and eventually an enthusiasm for state-of-the-art involvement.

4.4 Assessment

The assessment of the work done with PAISLey has three parts: future plans, strengths, and weaknesses.

There are several additions and modifications planned for PAISLey. Zave would like to make the system more interactive, perhaps adding graphics editing and animated execution. Support for the transformational implementation has yet to be developed; the reliability performance constraint is not functional. These are current concerns. Support for modularity is planned, possibly in the form of abstract data types for the set definitions. Plans also exist for a report generation capability from a database representation of the specification. It is recognized that extension for exception mechanisms is necessary, probably allowing the language to leave the functional notation temporarily for exception handling specification.

Strengths of the language are many. PAISLey is a formal language that produces an executable specification. Tools include a consistency checker

for static consistency. Inclusion of the environmental model is a big plus, and this systems approach can lead to increased reliability for applications in the targeted domain. The environment is often a source of change, so its inclusion in the model allows incorporation of this change. This feature, however, does not seem to be the current emphasis. Communication with the user is facilitated since the specification is a problem model. This orientation may also facilitate traceability as the requirements are closely related to the corresponding problem statement. The tolerance of incompleteness in the executable specification is an unquestionable advantage.

Criticism of PAISLey includes various aspects. The approach incorporates design level decisions with the explanation that they are necessary for the transformation approach. However, the requisite transformational methods are not available yet. The approach is a poor predictor of solution complexity, since it does not model the solution. An embedded system description in PAISLey is not extensible. It is changeable in the sense that incomplete descriptions can be further defined from within or outside the system, but processes cannot be added to the system. While it leads to a representation that is machine processable, the functional notation is very difficult to use, for requirements engineers as well as end users of the specification documents. Suggestions have been made to augment the specification with diagrams and to augment the language with application dependent notations, similar to macros. More performance analysis capability is needed. The urgent performance requirements of the target application require attention here. Reliability is still a future plan; PAISLey lacks the capability to express testable timing and fault tolerant properties.

5 Statecharts

Another specification language, one that is a relative newcomer to the scene, is the language of Statecharts, being developed at the Weizmann Institute in Israel by David Harel. Primary references for the language are [Har87], which describes the syntax and intended use, and [HPSS87], an extended abstract of the semantics of the language. Other sources include a technology assessment by Microelectronics and Computer Technology Corporation (MCC) in [BGFG86] and information about STATEMATE, a development environment that incorporates the Statecharts specification language, in [iLo87] and [HLN*88]. Statecharts are discussed by first considering the intended context for the language, followed by an overview of the language itself. Experience with the technique is related, and finally, an assessment of its current status and future plans is given.

5.1 Context

Statecharts are intended to deal with the specification problems of reactive systems. Reactive systems are those that are essentially event-driven systems, reacting continuously to external and internal stimuli [Har87]. Statecharts could also be used for applications such as data transformation systems, but their special features would not be exploited in that realm. The targeted systems involve real-time, embedded, control and communication, and interactive applications; all of these require maintenance of a relationship with the software system's environment rather than computation of input/output functions.

States, events, and conditions of finite state machines are suitable for describing real-time reactive systems. This is not a new idea; for example,

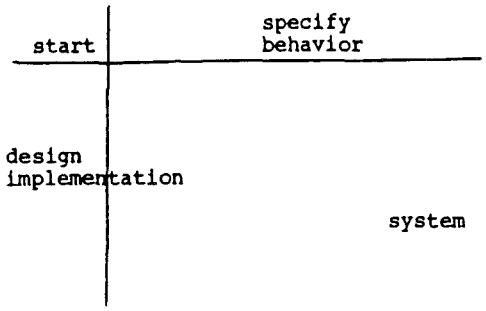


Figure 9.

Petri Nets provide a state-event description. Problems with state diagram descriptions in the past have stemmed from the exponential explosion in the number of possible states in a complex system, the single dimension of the flat state diagram, and the lack of communication description capabilities. Harel states that to overcome these problems an effective state event language requires modularity, hierarchy, structuring, a way of reducing the number of states considered, orthogonality, and generalized transitions [Har87]. Statecharts is the implementation of his ideas for extending finite state machines to include these features.

Typically, reactive systems require complex behavior that does not decompose recursively into simpler functions; modules of the behavioral description are not apt to correspond to modules of an implementation. Yet the implementation process must break a complex system down into smaller physical parts. Harel and Pnueli have presented a *magic square* for development of reactive systems that combines the development of the behavioral specification with the activity decomposition necessary for design and addresses interconnections among the two representations [HP85].

As previously indicated, problem structure does not necessarily reflect

the required system structure in complex reactive systems. The implementation may be constrained by existing interconnections, distribution, or other structure. Magic square development is a two-dimensional process with the design of an implementation and the specification of the behavior for that system progressing at once on some path through the two-dimensional space (cf. figure 9). Progress along the horizontal dimension is achieved by the refinement of a statecharts representation, while progress along the vertical is from decomposition of system activities into a tree-like structure.

The ideal path through the space is not determined. However, it should have some horizontal progress prior to each significant vertical advance. The behavioral representation at a point on the path corresponds to the implementation description at that point. Consistency between the multiple behavioral representations is very important (and non-trivial), where consistency is taken to imply that the external behavior of a system module on one vertical level must be equivalent to the external behavior of the next level of system modules that implement it.

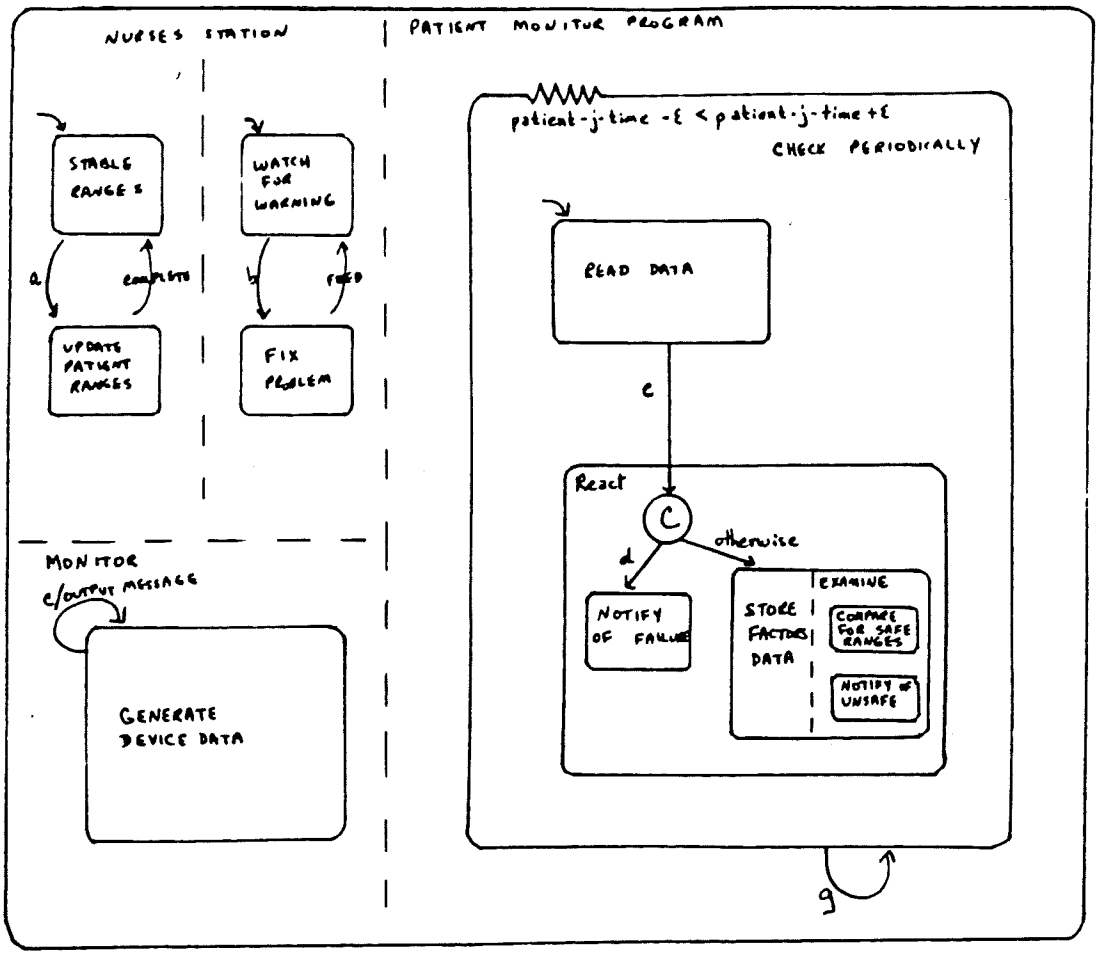
Major current employment of statecharts is within the evolving development environment by i-Logix, Inc. and Ad Cad Ltd. STATEMATE is a graphical working environment for use by system developers not only for requirements definition, but also for design, for analysis, and for documentation. The expectation is that a reactive system is better developed by considering three views—functional, structural, and behavioral—to create a visual formalism of the developing system. Four graphical languages are provided in STATEMATE, one for each of these views plus a forms language for nongraphical information and for specifying relationships between the different views.

The functional view uses activity charts to describe activities, data stores, and the data-items (including control) flowing between them. The system being developed is assumed as the highest level activity, and it is decomposed recursively into sub-activities, with base activities at the lowest level. Each non-base activity has a controlling statechart in the behavioral view. The structural view accomplishes high level design as the physical components and data structures of the system's implementation are specified. The language of STATEMATE for the structural view is module charts. The behavioral view specifies control with statecharts.

Besides the four languages, STATEMATE provides definitions of primitives for all four representations that are useful for statecharts such as time-out, true, false, exit, history-clear, deep-clear. STATEMATE has tools to check for incompleteness, inconsistency, and redundancy between and within the views and their hierarchical levels. Prototyping and coding packages are also available.

5.2 Overview

The finite state machine approach employs states, events, and transitions to describe the dynamic behavior of a system. There can be a finite number of independent states, and all are on the same level. Missing from this approach is any method for structuring the system, which is a serious problem as the number of states grows and with it the number of transition possibilities. To correct this, statecharts begins with the finite state machine approach and adds modularity, hierarchy, and broadcast communication across levels. In so doing, the complexity of the system description is reduced for users of the description as well as for engineers establishing the description.



Note: While monitor is an analog device, it can only be modeled here as discrete.

Events and conditions:
- a--safe ranges require changing
- b--notification of failure or unsafe ranges
- c--device-data available
- d--type="FAILURE"
- e--data read
- f--factors outside safe range
- g--[data stored $\wedge$ (range is safe $\vee$ nurses station so notified)] $\vee$ nurses station notified of monitor failure

Variables:
- device-data = pulse,temperature,blood-pressure,skin-resistance
- message = type,device-data

Figure 10.

Statecharts begin with conventional state diagrams, add AND/OR decomposition of states, and allow interlevel transitions between states plus broadcast communication. States may be composed in a bottom-up fashion by clustering states with common features or decomposed top-down by refining states into substates with more details. Note that clustering reduces the number of states to be dealt with at the higher level. If substates are to run in parallel, the decomposition is said to be AND, and the independent states are called orthogonal; if only one state or the other is to execute, the decomposition is exclusive OR. Parallel states may know the status of one another if that is desirable. Figure 10 describes requirements for the patient monitoring program (cf. figure 2) in statecharts. STORE FACTORS DATA and EXAMINE are orthogonal states, indicated by a dashed line dividing the superstate in which they are contained.

The notation for the language is a formal, graphical one. Rectangles with rounded corners represent states, while directed arcs represent events. Sub-statehood is depicted by containment. Events are labeled as event(condition)/action where the condition must be true at the instant the event occurs if the transition is to take place. If the transition does occur, the action is taken and the arc is traversed to the next state. Events can be junctions of events, such as $e \wedge f$, but the junction must be instantaneous. Actions can include such things as start(activity A), stop(activity A), or schedule(activity A, in x units). The latter indicates activity A will be scheduled to occur in x time units from now. Actions may also generate events which in turn trigger other state transitions.

As timing is an essential consideration for reactive systems, it may be necessary to represent minimum and/or maximum times that can be allowed for certain activities. This may be shown in Statecharts with a

jagged edge on a section of the state box and indication of an upper and/or lower bound. The appropriate transition to be taken in the event of timeout or delay should be so labeled. The example in figure 10 requires periodic checking of monitor outputs. This is expressed as a time range of not less than patient-j-time$-\varepsilon$ and not more than patient-j-time$+\varepsilon$ (which does allow a phase shift for the periodic checking).

Two special connectives are provided for specifying entrance to particular substates of a superstate hierarchy. On entrance into a state with no indication of which substate to select, a transition is implied to the default substate. This is indicated by a small arrow originating inside the superstate with no source and pointing to the substate. The history connective (Ⓗ) is used to indicate that the last previously occupied substate should be entered on transition to the superstate (a default entrance should be specified for the first entrance of the superstate). Ⓗ* may be used for recursive substate history entrance.

Other special notations included for convenience are transition stubs and connectives for conditional and selective state entrance. Conditional substate entrance is indicated by Ⓒ. The next substate is based on conditions that must be true at the instant the transition is taken (cf. state REACT of figure 10). Similarly, selection entries, denoted by Ⓢ, allow transition based on a selected event. The event is usually based on the value of some element. In some diagrams it may be desirable to eliminate unnecessary details from consideration, and, therefore, only the more abstract superstates are shown. This zooming-out process, as it is sometimes called, may require transition stubs, i.e. arrows whose source is a short perpendicular line within superstates. Substate entrances that do not follow the default and transitions that are not relevant for all substates are

indicated in the abstract chart with these stubs.

Some additional features have been considered for statecharts. These include arrays of states (substates) and overlapping states with common substate(s). STATEMATE has not included either of these; overlapping states are not permitted in any of its views. Semantics for a base set of elements and some interrelations have been defined in [HPSS87]. Included are states, history and default entrances, variables, expressions, conditions, events, actions, and transition labels. It is not determined as of this writing whether or not these semantics can be readily extended for such additional features as overlapping states.

5.3 Experience

Examples of the use of statecharts for problems of sample size are available. A detailed Statecharts specification of a Citizen's watch is included in [Har87]. STATEMATE has been used successfully by Israel Aircraft Industries to design the avionics system for a fighter airplane [HLN*88]; this indicates the appropriateness of STATEMATE for large, real, development projects. As automated support was not available, Statecharts were used manually for the behavioral description of the developing system. The results were encouraging; a substantial time savings was attributed to the language.

Statecharts are being used experimentally in the software, electronics, and semiconductor industries. The developers of Statecharts are testing their suitability for hardware components, communications systems, and interactive software systems.

The Software Technology Program of Microelectronics and Computer Technology Corporation (MCC) undertook a 2-week study in 1986 of Stat-

echarts' capabilities for specification and analysis [BGFG86]. Statecharts experts were employed to specify the requirements for an elevator control using the method. Results indicate that use of extended features such as overlapping states and conditional entrances may complicate the specification for readers. Structuring of the control description by hierarchies and decomposition is seen as a clear advantage for Statecharts. Although those involved in the study at MCC felt the notation is in its infancy and therefore has some weak points, it is a phenomenon that bears watching and may prove useful for the development of many complex systems.

5.4 Assessment

Statecharts is an evolving technology for representation of reactive systems and is useful for embedded software systems. The developers hope to someday include parameterized and overlapping states into the language; both are a means for economizing the notation. They also want to include the use of temporal logic as an assertion language to accompany statecharts or as a scenario language providing a basis for the Statechart system representation.

There are a few drawbacks to the current Statechart formalism. As with other formal notations, it is difficult to tell when the line between specification and design has been crossed, so design may slip into the requirements specification. Furthermore, there is no mechanism provided for recording decisions made requiring tradeoffs of alternatives. This is needed for the rest of development as well as during requirements analysis. There is also a need for representation of arrays of states that is simple to understand.

Statecharts provides a visual formalism of the behavioral requirements of the system. This is a distinct advantage for communication, as a graph-

ical model is more readily understood, and for verification, as the model is backed by formal semantics. Orthogonality and hierarchy structuring augmenting the underlying finite state machine model provide the modular breakdown needed for dealing with large, complex systems. The inclusion of the environmental model and means for expressing timing constraints are important for embedded systems. In particular, time-outs allow expression of exceptions to timing assumptions for the environment. This formalism is still very new and reports of its use for other large, real problems will be eagerly awaited.

6 SCR

One of the most well-known experiences with specification of a large embedded system is the work done for the operational flight program of the A-7 aircraft by the Naval Research Laboratory (NRL). This experiment resulted in a semi-formal language technique for expressing requirements. As the stated purpose of the project was Software Cost Reduction (SCR), the work is referred to here as the SCR technique. Much of the work of David Parnas influenced the approach taken, and thus his preliminary work is discussed as the context for the SCR technique. An overview of the specification language technique is given, along with a description of the experience with its use. Finally, the value of the SCR model is discussed.

6.1 Context

From 1972 through 1978, Dave Parnas published a series of papers that set the stage for the SCR work [Par72b], [Par72a], [Par76], [Par77], [Par78]. These writings focused on his interest in ways to improve the design process. The following five (overlapping) themes are recurrent in this work.

1- The criteria for modularization of a system during design must be information hiding, not time ordering. Information hiding implies the encapsulation of a design decision (often called a secret) in one module.

2- The design is structured to accommodate change. The information hiding criteria discourages the distribution of information so that if decisions are changed, fewer modules are affected. Small components incorporating only one function are required.

3- The system structure of modules may be viewed as a partial ordering based on the relation *uses*—a module *uses* another if a correct execution of the second is essential for the first to meet its specified requirements.

4- A precise record of all decisions, including those intermediate to specification or design, must be maintained for consistency and future analysis.

5- It is important to verify as early in the development process as possible the correctness of decisions that have been made.

In light of these ideas, a methodology was suggested in [Par77] for development of software that encourages the fulfillment of Parnas' ideas. The requirements definition is stressed as a process that must anticipate change. To have flexibility in the product, it must be a concern early in development. At this time the subsets should be identified, i.e. a minimal subset of functionality, a minimal *system* if you will, that provides useful service. Then minimal increments can be determined. Not only does this encourage ease of change and extensibility, but it provides small increments of specification that can be verified. Information hiding is used to modularize the system design for changeability. The system should be viewed as layers of virtual machines providing the identified subsets through the *uses* hierarchy, a graph with no loops. This aids in verification at all levels of development as it avoids the problems encountered when nothing works until it all works.

Individual modules will be further specified during high-level design to provide such information as the set of possible values, initial values, parameters, and effects (including any error handling).

Throughout this work, the specification is viewed as "the precise statement of the requirements that a product must satisfy." [Par77]. This is in contrast to the often used description of a high-level implementation to meet informal or perhaps unstated requirements. A major goal for the requirements specification process is to establish subsets of capabilities that will work. The requirements statement need not be in a formal language, but it should not be an informal, natural language description; a precisely defined notation is preferred. Parnas stresses the importance of an abstract, but not vague, description in terms of user observables without reference to any implementation. The SCR technique incorporates most of Parnas' ideas for the specification of requirements.

6.2 Overview

The language for the Software Cost Reduction model, described in [Hen80], is a semi-formal one. The organization of the document is intended for reference, with a glossary of terms, table of contents, indices, etc. included. Standard formats are used that include value templates and allow English descriptions within a formal setting. These provide for a consistency in the specification description that leads to better understanding and a representation that is useful throughout the life cycle. The requirements are organized in tables whenever possible; these are aids to completeness and consistency checking as well as reference. Selection tables specify functions that depend only on the particular operating mode of the software, where mode indicates an abstract state. Condition tables are used to specify functions dependent on other disjoint conditions being true. Event tables specify actions required of demand functions and periodic functions upon the occurrence of certain events.

```
Environmental Interface
      Input Data Item:  Message-From-Monitor
              Name:  /Data/
              Description:  Indicates !message-type! and the
                      !device-data!.

      Input Data Item:  Safe-Factor-Ranges
              Name:  /SRanges/
              Description:  Provides new value of safe ranges
                      for the indicated patient and monitor.

      Output Data Item:  Notification of Device Failure
              Name:  //DFail//
              Description:  Alerts nurses station of device
                      failure message.

      Output Data Item:  Notification of Unsafe Ranges
              Name:  //RUnsafe//
              Description:  Identifies patient and factors that
                      are not in safe range.
```

Figure 11.

A specification in this language describes what the system must do to pass acceptance testing; it does not describe how the system is to be implemented. The specification document addresses not only the requirements at the present time, but likely changes or additions to the system requirements.

Information comprising the specified requirements is related in five categories, each of which may be further subdivided as the application and available information warrant. See figures 11, 12, and 13 for an example of specification using the SCR language.

1- Environmental Interface Requirements. Here the environment of the embedded system is described including the type of computer on which it will be required to run, if this is known. The complete interface that is required between the system and its environment is specified.

   Each independent input and output is defined in a data-item. Data-

States and Functions

Modes:

startup	New safe ranges are input or other restart conditions.
stable system	Normal operating mode. Transitions to *intervention* when //DFail// or //RUnsafe// is output.
intervention	Device failure (implies some operation(s) has ceased) or an unsafe reading has occurred.

Functions:

demand function name: New-ranges
Modes in which function required: *startup*
Initiating event: !new-ranges-avail! becomes $TRUE$
Output affect: Stores new/replacement ranges in an internal data base.

periodic function name: Read-data
Modes in which function required: *stable system*
Initiation and Termination Events: None (always done)
Output affect: Stores input patient data in an internal data base.

demand function name: Check-data
Modes in which function required: *stable system*
Initiating event: !new-data-avail! becomes $TRUE$
Output data items: //DFail// and //RUnsafe//
Output description: If no failure of device has occurred, !device-data! is checked for unsafe patient ranges for this patient.

Condition table (Check-data):

Mode	Conditions		
stable system	/message-type/=$FAILURE$	!device-data! not within /SRanges/	otherwise
ACTION	output //DFail//	output //RUnsafe//	no action

Figure 12.

```
Timing Requirements
        Read-data function:
                current rate = $patient-j-time$
                minimum allowable rate = $patient-j-time$ - ε/2
                maximum allowable rate = $patient-j-time$ + ε/2

Dictionary
        /Data/ = !message-type! and !device-data!
        !device-data! = (pulse, temperature, blood-pressure,
                skin-resistance, device-id)
        !message-type! = ($FAILURE$ or $OK$)
        !new-data-avail! = indicates new data has been read
                and stored
        !new-ranges-avail! = indicates new safe ranges should be
                input
        $patient-j-time$ = predefined rate for reading monitor
                from patient j
        /SRanges/ = (hi-pulse, low-pulse, hi-temperature, low-
                temperature, hi-blood-pressure, low-blood-pressure,
                hi-skin-resistance, low-skin-resistance, patient-id)
```

Figure 13.

items are described via templates. Relevant facts about hardware interfaces that constrain the system should be related here as well as accuracy, value range, resolution, and timing characteristics (for input). Details common to any hardware device are noted as well as those relevant to the specific device whose use is planned. Care must be taken to avoid assumptions. The templates encourage completeness without requiring a rigid syntax.

Each output data-item is associated with a unique function that must be addressed in the next information category.

2- States and Functions. Functions the system is required to compute are described in terms of externally visible effects only, no algorithms are implied. As the relevant details of hardware interface are included in data-items, the functions do not address this information and so are meant to be relevant even if the devices are changed. A function may result in one or many outputs, but each output is associated with only one function. Functions are described using tables of conditions

and events. A condition stipulates aspects that must occur for a measurable length of time, while an event describes the moment in time that conditions change value. In order to reduce the number of conditions that must be considered, a finite state machine approach is taken. Without detailing all the states that are possible, operating modes are defined as superstates of the system. These are described in terms of the conditions they require (true and/or false conditions) and events that cause transition from one mode to another.

3- Performance Requirements. Instead of scattering timing and accuracy requirements throughout the specification, they are described separate from the data-items and functions. In an ideal situation, the maximum delay between request and response for each demand function will be specified and the minimum and/or maximum frequency for each periodic function will be specified.

4- Look Ahead. In keeping with the background work of Parnas, SCR specification techniques encourage a precautious approach. Required behavior of the system should undesired events occur is specified here. Also indicated are constraints on the design to allow for features likely to change during the system's lifetime and to allow for reduced functionality by identifying subsets of capability.

5- Aids for document use. Finally, a glossary of terms, dictionary of identifiers, and indices are included in the requirements document. A list of references that includes relevant works and people consulted is also given.

6.3 Experience

The SCR techniques were used to specify the operational flight program of the A-7 navy aircraft. The program, which interfaces to twenty-two devices, is part of the navigation/weapons delivery system. Before this project was begun, no requirements document was in existence, but there were flight manuals, mathematical algorithm analyses, flowcharts, and code (approximately 12,000 assembly language instructions) for the existing program. The goals of the project were three-fold:

1. to demonstrate the feasibility of using software engineering principles for a large, real-time, program,

2. to establish a model for future systems specifications, and

3. to provide a forum for study of additional research ideas for complex systems.

The motivation for the undertaking is well expressed in this paragraph taken from [PCW84]:

> "More than five years ago a number of people at the Naval Research Laboratory became concerned about what we perceived to be a growing gap between software engineering principles being advocated at major conferences and the practice of software engineering at many industrial and governmental laboratories. The conferences and many journals were filled with what appeared to be good ideas illustrated using examples that were either unrealistically simple fragments or complex problems that were not worked out in much detail. When we examined actual

> software projects and their documentation, few showed any use of the ideas and no successful product appeared to have been designed by consistent application of the principles touted at conferences and in journals. The ideas appeared to be easier to write about than to use."

The project resulted in a 500 page specification document [HKPS78] after some seventeen man months to establish the techniques and the requirement specifications. Included are seventy input and ninety-five output data-items. Certainly the document has become a model for future system specification [HM83]. It has also provided the desired forum for refinement of such ideas as use of the specification during design, proper content and organization, and additional documentation for design [PCW84]. The results of the undertaking indicate the feasibility of the principles demonstrated for requirements specification of a complex embedded software system.

6.4 Assessment

Since the A-7 project using the SCR specification techniques, the document produced has been studied and used as a guide for other projects. Positive features are numerous. The consistency of the description through document format and templates makes the requirements specification easier to use and to understand; such aids also encourage completeness checking. The organization as a reference tool is a positive feature throughout software design, testing, and especially maintenance. Look ahead sections that allow for future changes also encourage a maintainable specification and system. The requirements are testable because they are stated in terms of observables only. In addition, the technique avoids any reference to how the

system is to be implemented, and includes documentation of performance requirements, essential for the embedded systems domain of interest in this paper.

The A-7 project showed the feasibility of the SCR technique for a system that existed, but was undocumented. It would be interesting to apply these ideas to a "start from scratch" development effort as well. Automated tools to augment and extend the techniques would also be helpful, especially for large systems where manual checking for such things as consistency is a mammoth undertaking.

It remains to be seen how well the techniques developed explicitly for the A-7 program can be applied to other projects. Heitmeyer and McLean [HM83] report using the SCR ideas as a starting point for their work. However, they wanted a more formal approach for the specification that did not rely on externally visible factors.

7 Others

There are other languages that merit some consideration for the specification of complex embedded systems. These, however, do not have the depth of development as do the languages already discussed. Brief descriptions of special features of these others are given as appropriate.

RTRL

A finite state machine model is used to define the language called Real-Time Requirements Language (RTRL) [Dav82], [CDK85]. The basic model is extended to allow modularity, signal handling, resource synchronization, and timing expression. The latter capability is of special interest. Maximal and minimal timing constraints are representable in one of four categories: stimulus to stimulus, stimulus to response, response to stimulus, and response to response time restriction requirements. Constraints on event duration time are also representable. Assumptions about the environment providing stimuli to the system are included as requirements in the model.

There is a set of tools for consistency and completeness analysis for RTRL called the Requirements Language Processor (RLP). RLP has the capability for processing a set of languages called a family that are each special purpose languages. RTRL is appropriate for real-time systems that are dominated by sequential computations. In particular, it was developed for telephone switching programs.

Another language, Specification and Description Language (SDL) created by the Consultative Committee of International Telephone and Telegraph (CCITT) circa 1980 [RS82], is also intended for telephony applications. It has a pictorial and graphical notation, but the program represen-

tation is not yet available.

EBS

Chen and Yeh have developed a language that, while intended for distributed systems, could be used to describe some embedded systems by dividing the system into processes [CY83]. In this Event Based Specification (EBS) language, the behavioral specification is in three parts: environment, embedded distributed system, and interface.

The model consists of events and their two relationships, *precedes* and *enables*, with concurrency implied for events that are not related (implied or stated). First order predicate calculus is used to describe the required behavior.

The EBS language is also used to develop a top-level design or structural view. The behavioral specification is then used to verify the structural description and to analyze for such properties as safety and liveness. It is not clear that the language can be used to express other accuracy and timing requirements.

ESML

Recently, a language based on extended data flow diagrams has been proposed called Extended System Modeling Language (ESML) [BJKW88]. The work is founded on the transformational schema of Ward and Mellor [WM85]. The language includes notation called termination for representing the environment of the system that supplies inputs and receives outputs. There are also notations for data flows and for control flows called transformation schema. Data flow schema may be decomposed into several

lower-level schema or specified as primitives; primitive schema must then be thoroughly described in some other language or graphical form. Control transformations must be further specified with state transition diagrams or some other table form.

Analysis of the transformation schema model is possible by providing tokens that flow through the diagram in much the same way that tokens are used in Petri nets. This leads to the last set of languages that will be discussed, i.e., those based on Petri nets.

Petri Nets

A popular and powerful medium for specifying control flow of concurrent systems is Petri nets. Based also on the finite state machine model, they provide a graphical notation for system description [Pet77]. Graph nodes called transitions are used to represent the occurrence of events, while nodes called places represent conditions that make up the overall system state. Control flow is depicted by tokens passed from place to place through enabled transitions. Concurrency is easily represented by multiple enabled transitions. Various authors have worked with extensions to this model for representing data flow (using colored tokens and memory) and timing (using transition execution time). In particular, Coolahan and Roussopoulos [CR83] specify timing requirements for time-driven systems that have critical timing constraints and a master timing mechanism for the system.

The complexity of embedded systems calls for some type of hierarchical model to simplify the specification task, however. Petri nets are a flat model, without hierarchy, and can require quite a complex graph to describe a large concurrent system. The nets are difficult to analyze [Age79], though automated analysis tools are available to aid in analyses for deadlock, safety,

mutual exclusion, etc. (see e.g. [MR85]).

8 Current Status

There is no language for specifying the requirements of complex embedded software systems that embodies all the desirable characteristics described in section 2 of this paper (cf. figure 1). Nor should a language be expected to have all these features in order to be selected for use in this context. The current state of the art in requirements specification for embedded systems requires the developer to choose a language with features and accompanying methods that are appropriate to the particular task at hand.

The languages discussed in this paper reflect many correctness features. Almost all are formal, promoting precision and minimizing ambiguity. Static internal consistency checkers have been developed for RSL, PAISLey, and Statecharts (the latter are part of STATEMATE). Dynamic consistency is usually examined by means of simulation; simulation generators are part of SREM (for RSL), while the PAISLey specification itself is executable for simulation.

Both RSL and SCR include techniques supportive of verification. Requirements are stated in terms of testables (observables) only. The database used with RSL also enhances traceability of requirements.

Most applications need flexibility for change due to omissions, errors, advancements, etc. Two different dimensions of this capability are reflected in RSL and SCR. The SCR technique considers decisions that are likely to change and requires they be handled accordingly throughout development. RSL enables the specification to be easily modified through the use of a centralized data base. PAISLey allows execution of the specification before it is complete; omitted details can be provided at a later time. RSL can be extended as a language to incorporate special features defined for a

specific application. The languages reviewed discourage the inclusion of design decisions in the requirements; SCR is entirely free of design.

The visual qualities of Statecharts promote communication among all those using the requirements. In this sense, the Statecharts formalism appears to be usable throughout development. There is indication that requirements generated using SCR are useful for reference throughout development also.

The importance of the interface with the system environment is clear for embedded software systems. PAISLey, SCR, and Statecharts examine an included model of the environment. It is possible to consider the environment modeled as a processing node in RSL, although this is not directly addressed. The expression of performance requirements is also clearly important to this area. RTRL considers maximal, minimal, and duration timing constraints. SCR and Statecharts have capabilities for expressing some timing requirements.

Enhanced capabilities are being considered for most of the languages discussed. It may be useful to incorporate temporal logic as an assertion language to accompany a requirements specification. This is being examined for Statecharts. Diagrams are important for understanding and communicating the specification. Confidence in their potential is evidenced by their use in RSL, Statecharts, ESML, and Petri Nets. Diagrams have been suggested to accompany a requirements description in PAISLey. Application dependent notations are also under consideration for PAISLey. RSL/SREM is being studied for extension to a life-cycle development system. The value of a complete, correct requirements specification is recognized for the entire software development life-cycle.

9 Conclusions

This work has examined the need for requirements specification languages for complex embedded systems. There are several desirable properties for a specification, namely, that it be complete, consistent, precise, unambiguous, formal, traceable, testable, changeable, extensible, design free, executable, communicable, and usable. In addition, for the embedded systems domain, the ability to represent the system environment, actions required upon the occurrence of undesired events and/or performance requirements is also desirable.

Research is encouraged in this area to improve languages and methods that already exist, rather than the development of new ones. In this light, four major languages were discussed. RSL, a language with graphical requirements nets and formal descriptions supported by REVS tool set; PAISLey, a formal, applicative language that is intended for an operational approach to software requirements specification; Statecharts, a graphical yet formal representation that is receiving attention; and SCR, the semi-formal language technique resulting from the specification of an embedded software system for the A-7 aircraft.

Other representation languages exist that have features desirable for the embedded systems domain. RTRL, EBS, ESML, Petri Nets, and a few others were mentioned.

As the previous discussions indicate, significant effort has been expended towards the goal of providing an appropriate representation language for the software requirements of embedded systems. However, much remains to be done.

First, studies involving use of current approaches are encouraged. Expe-

rience with specification of the requirements for genuine systems of realistic size is important for the languages we have and for newly proposed techniques. Without such documented experiences, those who have need of the technology will not be convinced of its usefulness or applicability.

Another aspect that demands further research is the type of representation used. In his taxonomy of requirements specification languages, Roman states "[T]he industrial success of a specification technique is heavily dependent upon its treatment of human factors, that is the concepts it makes available and the interface style it supports." [Rom85]. Graphical representations that can be used to represent the system at different, appropriate levels of abstraction have a lot of intuitive appeal, perhaps because they describe the system in a manner close to the mental model of analysts. It is hoped that a graphical model promotes communication among the users, analysts, and design engineers despite their diverse backgrounds. Empirical studies are called for to demonstrate the degree of validity of these conjectures. The use of a graphical language, however, may not be adequate. For example, there is no convenient way to record tradeoff decisions in a graphical notation. It is not clear that this type of representation would be best for reference in later stages of the development. Other reference documentation is probably required. The Embedded Computer System Workshop recommended an abstract model that will facilitate analysis; such a model is represented in a language of objects, relationships, and attributes [WL85]. Perhaps multiple consistent representations are required for the description of such complex systems.

There is quite a bit of emphasis on having an executable requirements specification for embedded software systems. While this is certainly a desirable feature for analysis and demonstration, it does not come cheap. Care

must be exercised so that the executable formal language is not objectionably close to á programming language. It is not practical to deal with the details of coding the solution twice, once during requirements specification, and once again in implementation.

The fourth area of concern is with the representation of non-functional requirements. These are difficult to express in a formal notation because of the lack of a theoretical foundation [Rom85]. Research is ongoing for solutions to this problem, [Jus88] and [Jaf88], but more is needed. Especially encouraged are techniques that can be used as part of available languages. Two different types of timing requirements are treated alike in most notations: those that specify assumptions about the timing of environmental stimuli and those that specify required timed behavior of the embedded system. The former usually includes a time-out requirement. Different notation options for these would encourage completeness checking and correctness without overconstraining the designer.

The importance of the inclusion of an environmental model in the specification has already been pointed out. Completeness and correctness analyses should also include this representation of the embedded system's environment. White and Lavi state: "insufficient analysis of external phenomena is trouble" [WL85], and White indicates the importance of a well defined boundary between the two [Whi85]. The interface between the environment and the embedded system is very complex and requires special consideration.

Most of the languages discussed are part of a method for requirements specification and analysis. These methods must advocate a careful, systematic, and disciplined approach to specification that leads to completeness of the representation. They also must include analysis capability to al-

low checking for various required properties before the construction of the system begins. Some completeness criteria can be checked by automated tools. Checking for other criteria may depend on the experience of the analyst in the particular application area. Development of techniques that will promote and improve completeness and other requirements specification properties are essential.

Finally, it is important that the work done on specifying requirements be integrated with the rest of the development process for embedded systems. It is difficult to draw the line between high-level design and requirements specification for this domain; however, the distinction is necessary so as not to under- or over-constrain the software designer. Plans for change included in the requirements specification may add constraints on the designer, but they are important to the overall development process. Methods that facilitate cooperation between all levels of development will surely reduce the overall cost (in both money and time) of software production.

It is probably true that no one language or technique will meet all the needs of complex embedded systems. If this is the case, integration of the good features of several languages should be considered with the possibility of multiple consistent representation views for various specification users.

Requirements specification methods for embedded systems is an important research area in software engineering. Work is encouraged in such areas as actual experience with language use, graphical representations, executability, non-functional requirements expression, environmental model inclusion, disciplined techniques for completeness, and issues affecting the entire development process.

References

[Age79] Tilak Agerwala. Putting Petri nets to work. *Computer*, 85–94, December 1979.

[Alf77] Mack W. Alford. A Requirements engineering methodology for real-time processing requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–69, January 1977.

[Alf85] Mack W. Alford. SREM at the age of eight; the Distributed computing design system. *Computer*, 18(4):36–46, April 1985.

[BBD77] Thomas E. Bell, David C. Bixler, and Margaret E. Dyer. An Extendable approach to computer-aided software requirements engineering. *IEEE Transactions on Software Engineering*, SE-3(1):49–60, January 1977.

[BCF*83] Robert M. Balzer, Donald Cohen, Martin S. Feather, Neil M. Goldman, William Swartout, and David S. Wile. Operational specification as the basis for specification validation. In D. Ferrari, M. Bolognani, and J. Goguen, editors, *Theory and Practice of Software Technology*, pages 21–49, North-Holland Publishing Company, 1983.

[BGFG86] Glenn R. Bruns, Susan L. Gerhart, Ira Forman, and Michael Graf. *Design Technology Assessment: The Statecharts Approach.* Technical Report STP-107-86, Microelectronics and Computer Technology Corporation, Austin, Texas, March 1986.

[BJKW88] William Bruyn, Randall Jensen, Dinesh Keskar, and Paul T. Ward. ESML: an Extended systems modeling language based

on the data flow diagram. *ACM SIGSOFT, Software Engineering Notes*, 13(1):58–67, January 1988.

[Boe80] W. E. Boebert. Formal verification of embedded software. *ACM SIGSOFT, Software Engineering Notes*, 5(3):41–42, July 1980.

[BZ87] Edward F. Berliner and Pamela Zave. An experiment in technology transfer: PAISLey specification of requirements for an undersea lightwave cable system. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 42–50, IEEE, Monterey, California, March 1987.

[CDK85] M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto. Requirements-based testing of real-time systems: Modeling for testability. *Computer*, 18(4):71–80, April 1985.

[CL81] C. S. Chandersekaran and R. C. Linger. Software specification using the SPECIAL language. *Journal of Systems and Software*, 2:31–38, 1981.

[CR83] James E. Coolahan, Jr. and Nicholas Roussopoulos. Timing requirements for time-driven systems using augmented Petri nets. *IEEE Transactions on Software Engineering*, SE-9(5):603–616, September 1983.

[CY83] Bo-Shoe Chen and Raymond T. Yeh. Formal specification and verification of distributed systems. *IEEE Transactions on Software Engineering*, SE-9(6):710–722, November 1983.

[Dav82] Alan M. Davis. The Design of a family of application-oriented requirements languages. *Computer*, 15(5):21–28, May 1982.

[Egg80] Paul R. Eggert. *Overview of the Ina Jo Specification Language.* Technical Report SP-4082, System Development Corporation, October 1980.

[GMT*80] Susan L. Gerhart, D. R. Mussner, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile. An Overview of AFFIRM: a Specification and verification system. In Simon H. Lavington, editor, *Proceedings of IFIP Congress 80*, pages 343–347, 1980.

[Har87] David Harel. Statecharts: a Visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hen80] Kathryn L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[HKPS78] Kathryn L. Heninger, J. Kallender, David L. Parnas, and J. E. Shore. *Software Requirements for the A-7E Aircraft.* Naval Research Lab, Washington, D.C., November 1978.

[HLN*88] David Harel, H. Lachover, A. Naamad, A. Pnueli, M.Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: a Working environment for the development of complex reactive systems. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 396–406, IEEE, Singapore, April 1988.

[HM83] Constance L. Heitmeyer and John D. McLean. Abstract requirements specification: a New approach and its application. *IEEE Transactions on Software Engineering*, SE-9(5):580–589, September 1983.

[HP85] David Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Springer-Verlag, 1985.

[HPSS87] David Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 54–64, Ithica, New York, 1987.

[iLo87] i-Logix. The Languages of STATEMATE. March 1987. STCON-00.

[Jac83] Michael Jackson. *System Development. International Series in Computer Science*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[Jaf88] Matthew S. Jaffe. *Completeness, Robustness, and Safety in the Behavioral Requirements for Embedded Command and Control Systems.* PhD thesis, University of California, Irvine, 1988.

[Jus88] Debra Sue Jusak. *Modelling the Semantics of Time Dependent Computations.* PhD thesis, University of California, Irvine, 1988.

[MR85] E. Timothy Morgan and Rami R. Razouk. Computer-aided analysis of concurrent systems. In *Proceedings of the Fifth In-*

ternational Workshop on Protocol Specification Verification and Testing, Toulouse, France, June 1985.

[Par72a] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par72b] David L. Parnas. A Technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[Par76] David L. Parnas. Response to undesired events in software systems. In *Proceedings of the Second International Conference on Software Engineering*, pages 437–446, IEEE, San Francisco, California, October 1976.

[Par77] David L. Parnas. The Use of precise specifications in the development of software. In Bruce Gilchrist, editor, *Proceedings of IFIP Congress 77*, pages 861–867, Toronto, Canada, August 1977.

[Par78] David L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the Third International Conference on Software Engineering*, pages 264–277, IEEE, Atlanta, Georgia, May 1978.

[PCW84] David L. Parnas, Paul C. Clements, and David M. Weiss. The Modular structure of complex systems. In *Proceedings of the Seventh International Conference on Software Engineering*, pages 408–417, IEEE, Orlando, Florida, March 1984.

[Pet77] James L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[Pla85] Patrick R. H. Place. Position paper for the third international workshop on software specification and design. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 184–185, London, August 1985.

[Rom85] Gruia-Catalin Roman. A Taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–22, April 1985.

[RS82] Anders Rockstrom and Roberto Saracco. SDL—CCITT specification and description language. *IEEE Transactions of Communications*, COM-30(6):1310–1318, June 1982.

[Sha85] Mary Shaw. What can we specify? Issues in the domains of software specifications. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 214–215, London, August 1985.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[SSR85] Paul A. Scheffer, Albert H. Stone, III, and William E. Rzepka. A Case study of SREM. *Computer*, 18(4):47–54, April 1985.

[TH77] Daniel Teichroew and Ernest A. Hershey, III. PSL/PSA: a Computer-aided technique for structured documentation and analysis of information processing systems. *IEEE Transactions on Software Engineering*, SE-3(1):41–48, January 1977.

[Whi85] Stephanie M. White. Requirements modeling for embedded computer systems. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 238–240, London, August 1985.

[WL85] Stephanie M. White and Jonah Z. Lavi. Embedded computer system requirements workshop. *Computer*, 18(4):67–70, April 1985.

[WM85] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems, Volume 2.* Yourdan Press, New York, 1985.

[Yue87] Kaizhi Yue. What does it mean to say that a specification is complete? In *Proceedings of the Fourth International Workshop on Software Specification and Design*, pages 42–49, Monterey, California, April 1987.

[Zav82] Pamela Zave. An Operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, May 1982.

[Zav84] Pamela Zave. The Operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, February 1984.

[ZS86] Pamela Zave and William Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, SE-12(2):312–325, February 1986.