Modeling the External Software Interface for Requirements Specification

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

Technical Report #89-41
December, 1989

Abstract

Requirements specification is an important part of the software, indeed the system, development process. It is critical that this effort be started early. This work suggests an early model for software developers to incorporate a systems viewpoint in their process. This model is an attempt to formalize an approach that will include a systematic representation of essentials of the external interface for software that is embedded within a larger system. The model is useful for early analysis of the software system and environment for such things as consistency, completeness, and safety.

1 Prologue

Recently the notion of a systems approach to software development has become popular. This can be employed during the requirements specification process to help alleviate some of the productivity problems in the frontier software development domain. There are two perspectives from which to view this. The first one is the viewpoint of the systems developer, who is outside the system and, with that focus, should look at software considerations much earlier in the development process than is the current practice. It is not enough to say that software will be there; software is not infinitely flexible!

The other perspective is the one of concern in this work. It is the perspective of the software developer, who can include a model of parts of the system external to the embedded software sub-system—include this model from the beginning of the software development effort.

The interface between embedded software and other components of the system is an important part of this model. Careful consideration of this interface can do much to enable a systems viewpoint for software developers. There is concern for specifying parts of this interface, concern for the process and for what can be specified. The military standard established by the Department of Defense for development of its software [DoD85] states that "the contractor must define and analyze the interface qualification requirements for each computer software configuration item," for example. The embedded computer system requirements workshop held a few years ago cited methods for modeling the environment of the software sub-system as one of the most important issues that should be addressed by research [WL85].

2 Models and their uses

What we are always doing in modeling is describing a system other than the actual one, with compromises between what is real and what we can describe. As figure 1 illustrates,

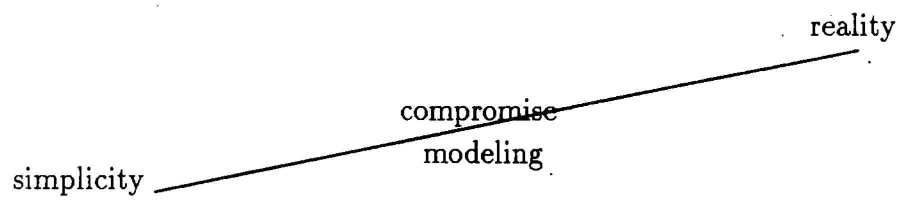


Figure 1

a system is modeled here at the left that approximates some real system on the right. The useful model is necessarily an abstraction that is simpler than the real system, else the modeling effort is not effective.

According to Cho [Cho87], "Modeling is the activity of understanding the problems under construction. A model is a representation of an existing or conceptual object, an abstraction of a real world phenomenon that will be the basis for development of a piece of software." Indeed, this is true for development of the entire system.

A model may illustrate components in a general sense only, although sometimes it may include components' most basic elements as well. It also shows the relationships (or lack of them) between the parts of the model. Each model is an example for some future effort; sometimes a model can illustrate how other systems should be described. For example, the A-7 operational flight software specification ([HKPS78]) has been used as a model of how to specify software requirements. A model is also a guideline for essential elements; it shows the essence of what is being developed. Concrete models can demonstrate the existence/non-existence of certain properties and can serve to document the existence of some features. A wind tunnel aircraft model, for example, can demonstrate the stability of the aircraft in turbulence.

The model suggested in this paper has all these motivations. It is an attempt to formalize an approach that will include a natural, systematic way to represent the essentials of the external interface of software in a useful way and then include it in analysis of the requirements specification.

The implemented, working system must have certain properties which are specified in the requirements for the system and have been delegated to the software. Sometimes the properties are dynamic and they must be ensured, checked, and maintained by monitoring, feedback loops, real-time checking, analysis, whatever. These properties can only be achieved by preparing the system under development, systematically from the onset, to satisfy them.

A further motivation, then, for the interface model is to better enable the software to control component interaction, both static and dynamic, to ensure system safety. Even for elements of the system that are not the responsibility of software, additional monitoring may be desired if their failure affects monitored or controlled elements. System reliability and safety are greatly affected by the interaction of system components. In order to analyze this interaction during the requirements specification process as far as the influence of software goes, a model of the interaction must be available. Others have called for such a model; Leveson and Harvey [LH83] have said: "[A]ssertions which involve the state of parts of the system external to the logic of the software, e.g., the environment in which the software operates such as hardware or support systems, will be necessary. Environmental failures or interfacing problems cannot be prevented by a purely software analysis of safety."

The term component is being used here to mean the smallest part capable of performing a function. One can think of each component as a black-box module that has some pre-determined functionality (from the software point of view). However, the particular pieces labeled as components may change as development continues, since the specifics of a given function may change and more details (further subdivisions) of components will be necessary to reflect these considerations. The process is iterative. Most information about other components must be provided, but software analysts may ask questions and develop a standardized approach to the information provided so that the software may be more adequately prepared for its role in the system.

3 Interface definition

The *Dictionary of Science and Engineering* [Par84] describes an interface as "some form of electronic device that enables one piece of gear to communicate with or control another; a device linking two incompatible devices, such as an editing terminal of one manufacturer to the typesetter of another; ...or a shared boundary." The model proposed here, then, is of an interface in the first and last of these senses.

This is a system level model, for a top down approach to software development. It is the viewpoint or framework within which to begin; the interface is the "context model" for the software development effort. The interface model describes the interaction of black-box components; it is a part of the environment and a part of the software. The focus, then, is on inputs and outputs at the system level between all the system components, for they are the interaction. The computer interface between the software sub-system and other components is a subset of the interface described here, which includes interaction between all components.

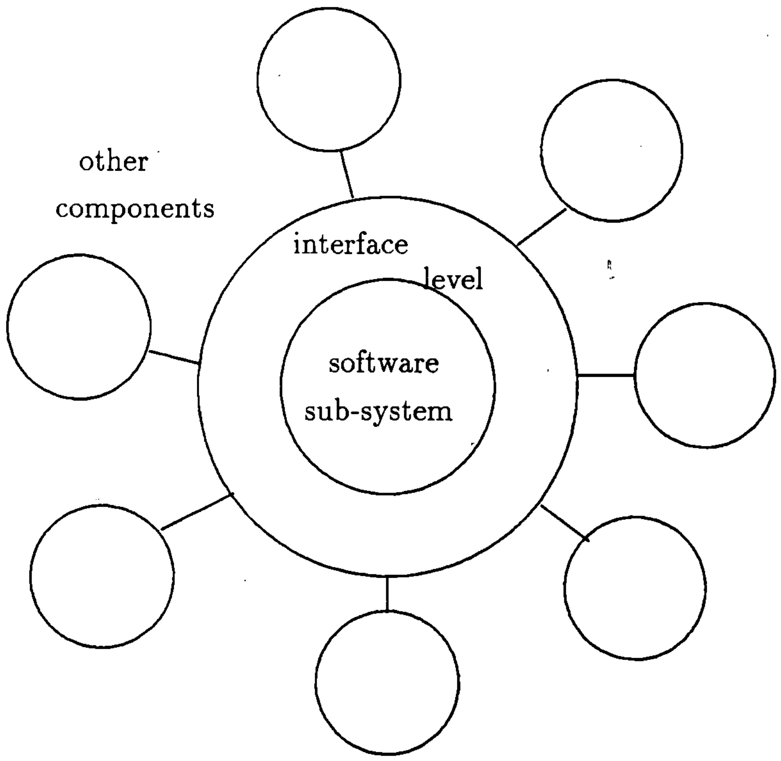


Figure 2

3.1 Views of Interaction

The relationship of this element to the rest of the system model can be viewed in multiple ways. Figure 2 depicts one view. It portrays the software as the "brain" of the system; all interactions are controlled by the software sub-system. A process control system would probably be of this type. The other picture, figure 3, takes the view that the software is just one of the components. Some manufacturing plants are of this type. It may have control over certain interaction, but some of the interface (that may/may not be included in the model) is totally out of the control of the software. The first view is that of total control; i.e., mappings between inputs and outpus will have software as a source or destination for each match. The other is for partial control of interface functions. The model and ensuing analysis can work for either intent, however the greater control results in a more thorough analysis for the first view.

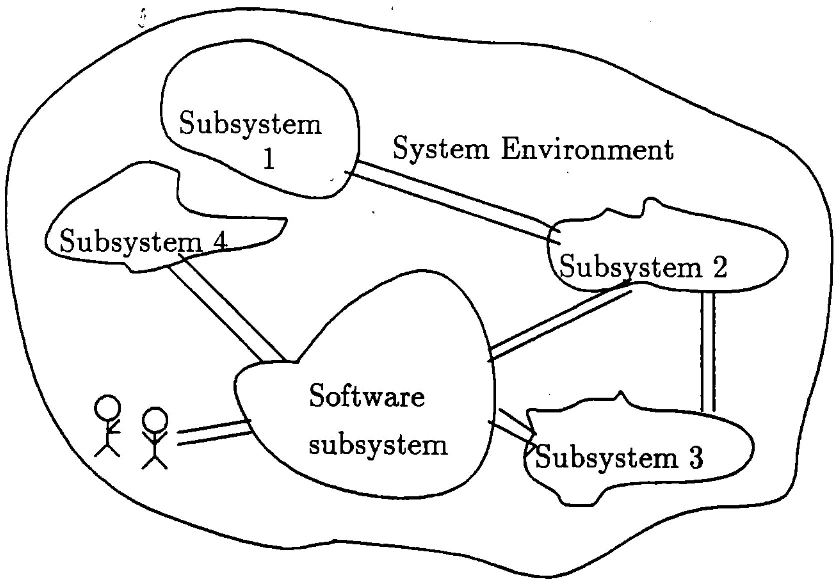


Figure 3

Interface deals with inputs and outputs; it describes a layer "between" the environment components and the embedded software. As this is a model for use during the earliest phase of development, it is appropriate to include only the form of the data exchange, not how it is exchanged.

The computer will need to acquire data (status, measurements) from, as well as provide data (instructions, commands) to, other system elements. The interface model is a description of the essentials of that acquisition process—both ways. The actual computer interface (i.e., the exchanges between software and other components) is part of the software requirements interface (a subset if you will). The software requirements interface also contains information about how other components interface; this information is necessary if the software is to have any (or can have any) monitoring functionality for system safety.

Most of the environment model is not at the discretion of the software developers. Rather this model is for the purpose of analyzing the appropriateness of the software requirements to meet the system requirements and to see how changes there (in the system requirements) may affect the software being developed and to minimize these effects, within the realm of the software control. Again, this demonstrates the need for a system view from both perspectives. Software is not infinitely flexible, and it is not operating in a vacuum. Changes will, of course, be made throughout development as an iterative process will develop the model—and the eventual system.

3.2 Model concerns

The interface model is a place to deal with various concerns. Lamport [Lam85] has stated that "misunderstandings in these [(interface)] low-level concerns is likely to be a lot more disastrous than the failure to correctly implement some subtle aspects of [communication] protocol."

Too often, issues of timing and value accuracy have been avoided during requirements specification. This is because there is no place or discipline to address these in any current techniques or languages for requirements specification. With the interface model, there is a place to specify these, and the discipline that it provides will increase the precision and correctness of the software specification.

The interface describes a mapping between outputs of the system components and inputs of system components. It is a model to show the "relationship" between the parts of the system. It can be used to demonstrate the existence/non-existence of certain safety properties and may even document the existence of some features.

A subset of these inputs and outputs is directly controlled or monitored by the software. For these the mapping will be between software I/O and another component O/I. Other mappings besides these may be interesting for their safety implication, however. Developing the interface model is not just scaling up of interface problems encountered within software; interaction problems of "unknown" components, those whose control is from outside the system and perhaps not according to any particular function, does not exist at lower levels.

Inputs and outputs have certain characteristics that may be assumptions. Some may be likely to change; this model attempts to allow isolation of these likely to change features which include capacity, capability, mode, undesired event requirements. A place is needed in the requirements specification model to document assumptions about components. This is not easy as these assumptions may be implicit instead of explicitly stated in previous documentation. Some analysis tools can draw out hidden requirements; others can uncover the need for omitted requirements. Requirements specification is the appropriate time to acknowledge these and to determine the desired responses if they are violated. It makes sense to isolate these assumptions and the checks for their violation in a separate level, away from component functionality, as they are likely to change over the lifetime of the system. Fussell has said "A potential problem is that the analyst is seldom aware of all the assumptions he has made."[FA79]

Constraints will need to be included in this model. These may be forced on the model by the system constraints, or are the results of analysis on the model that shows certain constraints to be necessary here. They are expressed as rules for the mapping of inputs to outputs (or vice versa) to obey.

Also of concern are failure events that include failure to follow specified I/O maps or failure to meet constraints placed on them, and failures of individual components in the system. These two sources represent failure in the interface and failure in the environment. They usually will be handled differently: Failure in the environment is something to react to; failure in the interface indicates another source of fault.

3.3 Use of the Model

The interface model is an organization that can allow for recovery from undesired events; can isolate changeable parts; and can have safety concerns and considerations built-in to facilitate analysis. The recovery from undesired events or exceptions belongs here, since this recovery is not a absolute part of the software functionality. Analysis should be included that augments this model so that undesired events are dealt with or prevented altogether. There is a need to "plan for the unexpected"; failures cannot be prevented altogether, especially when failure occurs due to disturbances in the environment.

In some systems, it may be desirable to have the interface be a watchdog for overall system performance, since it can have access to the necessary information to do this. This could be thought of as adding to the mission of the software subsystem, or it may already be included as part of the software interface.

Example components that might be included in the interface model are sensors, displays, the software sub-system, controlled components, and even humans. Example concerns for the model are accuracy, scheduling, volume, timing, proper and improper modes of operation.

4 Notation and Definitions

States, events, and conditions of the finite state machines are suitable for describing the software sub-system. The language of Statecharts [Har87] is an extension of finite state machines that includes such features as modularity, hierarchy, structuring, orthogonality, and generalized transitions. [Mel88] contains a more thorough description of the language and a comparison of its features to other state machine based languages as well as a discussion of some experience with its use. Statecharts provides a graphical language that may be suitable to describe the aspects of system components that are relevant to the interface model. However, it does not provide a straightforward notation for assigning attributes to the inputs and outputs to and from the components without introducing considerable clutter to the description. So, these will be described separately, with input and output declarations charts to augment the Statecharts. Properties will be declared for various inputs/outputs and mappings between them allowed/disallowed according to given, implied, or derived constraints.

The model is made up of a set of Statecharts showing the relevant states of components, input and output declarations for each data exchange, and a mapping between these. These mappings make explicit the assumptions that are made at this level and time. Analysis on the model may then be performed to examine the mappings and declarations for safety, consistency, etc. Descriptions of the normal (desired) behavior and defective (undesired) behavior are included. Required behaviors should undesired events occur can then be specified.

The model is developed along with the charts. When an exchange is called for, the I/O details are outlined in an exchange chart. This will show the need for a matching (under the mapping M) exchange to supply or receive the data and what its characteristics must be.

An input declaration, for example, must include declaration of the assumed acceptable range(s) for its value. The declaration must indicate what is to be done if this range is violated. Safety-critical variables may include a warning range or boundary, that does not violate safety constraints but is close to unacceptable, and an acceptable range.

The specification of ranges for values of inputs and outputs constitutes the specification of certain constraints. Some constraints are inviolable; others, like safety, are violable, but with undesired consequences. The process specifies constraints which the functioning system must not violate. If it does, then it will cause an exception.

When the constraint is violated, whether it be an exception to a given input or output range, an exception condition will be in force. This may occur for example when an outside disturbance such as wind causes an airborne system to encounter undesired turbulence. Exception correction may then call for operator assistance, backups, or fail safe procedures.

Some definitions will be helpful in the specific description of the model. Most of these relate to specifying timing properties.

According to Lowe, "The qualifications 'continuous' and 'discontinuous' as applied to control systems and elements refer to the magnitudes of certain signals in the system."[Low71]. Discontinuous here implies that the values jump. These may also be discontinuous in time, implying that monitoring, controlling, whatever has gaps in it timewise. It is called a sampling control system if it is not continuous in time. That is, it must rely on sampling data at intervals. Note that both manual and computer control are of necessity sampling systems. We cannot really read or do two things at once; computers cannot provide either continuous or continual output. If continuous monitoring (information) is required, then the intervals have to be made very small to *simulate* continuous control. To distinguish between these two continuity concepts and to emphasize the continual availability of certain values, continuous in time will be referred to as continual. That is, continual $\equiv$ time continuous and continuous $\equiv$ magnitude continuous.

On the other hand, components in the environment of the software are not necessarily, indeed not usually, discrete or sampling. It is often difficult to adequately represent the continuous nature of these with finite state machines.

From the operating systems domain (see, for example, Deitel [Dei84]), we can surmise definitions for the capacity or load of a certain declared input. Capacity is the measure of the maximum *work* per unit of time that a system or data line may possibly accomplish. That is, it is the rate at which the load from all can be handled by this one. Load is the measure of the actual amount of *work* that has been submitted and must be processed in order to be functionally acceptable. For our purposes, work is the acceptance or submittal of information, i.e., the only work we are considering is an exchange.

Note that load here is different from the pure systems engineering sense. Load will not cause the software to "wear out early", as with physical, mechanical components. Rather load may cause the software component to react in an unsafe way when it is violated (overloaded).

A process control system is defined by Lowe as an arrangement of elements that are interconnected to maintain, or to affect in some specified manner, some physical quantity or condition of the process which forms part of the system [Low71]. The product or condition that results may be thought of as attaining a certain state in this model. Then any product is just an output; both inputs and outputs may be physical quantities. To paraphrase Johnson, "Control events occur in a sequence for which an output state produces a change in the input state, which than causes a change in the output state. The process continues until some overall objective has been met. The present output is dependent on the sequence of previous states of the system." [Joh84]. For the builders of software systems, this objective may be a certain state; for some components it may be a physical output, like a certain flow rate.

Disturbances and time lags are features of the process and control elements or components and defined as follows: **Disturbances** are the effects of influences on the process, besides the controller. (If serious enough, they may require an exception procedure. **Time lags** are delays in the response (output) to a certain change or input. Note that these delays are not always unwelcome, but might be beneficial in that they restrict the speed with which disturbances can affect the system.

As a required response time is specified, that input must match with other exchanges, and the system level requirement is then affirmed throughout the interface model.

5 Interface Model

There are common, essential elements to the external software interface, no matter what the functionality of the system being developed. It is possible to discuss a "generic" interface model because, no matter what the specific requirements for the black-box software component (or any other component, for that matter), the interface basically has the same functionality: data exchange through inputs and outputs of system components. Anything that deals with passing information from one component to another, including exceptions to the normal, is a concern of the interface model.

5.1 Exchanges

A basic Statechart description of the system is derived from the system specification. These charts describe the states and sub-states of the system and how transition is accomplished from one to the other. The actual state of the system will be many orthogonal (or parallel) states. A particular component will probably be described as well by several states. The term exchange will be used here to indicate either an input or an output. The declarations for characteristics of exchanges are similar to abstract data types for interface components. An input or output is declared in relation to a particular condition, event, occurrence, or component state. The event/condition/state will be said to result in the output oy or input ix, for example.

A declaration exchange chart will include the following for input:

$value(ix) \in$ ValuRange
$time(ix) \in$ TimeRange
$timetype(ix) \in TT$
$source(ix) \in C$
capacity
exceptions

An output declaration will include:

$value(oy) \in$ ValuRange
$time(oy) \in$ TimeRange
$timetype(oy) \in TT$
$destination(oy) \in C$
load

ValuRange and TimeRange are sets. Their elements declare a range of acceptable values for input (output) and acceptable times for input (output) occurrence, respectively. Note that these do not prevent other values not in range from being generated, but rather notices that they have occurred. These ranges may be absolute, i.e., they may be definite numerical limits, or they may depend on some previous event or value. The example included in the next section illustrates both cases. These limiting ranges are appropriately documented in this model. Actually, there may need to be several layers of limits for a particular time or value; needs watching, getting closer, warning, danger is imminent, boundary line violated, or alarm, for example.

match

environment output	to	software input
continual		periodic S-R (on demand)
S-R		S-R
periodic		S-R(with appropriate R-time limits) periodic

software output	to	environment input
continual		continual periodic S-R
S-R		S-R
periodic		S-R periodic

Figure 4

The set of all *timetypes* (TT) is an ordered set.

$$TT = \{\text{continual, periodic, S-R}\}$$

Figure 4 shows the relationship between the various elements. Mappings between intended outputs and inputs will not be allowed to violate the chart. Often "details" of what kinds of interface will be required are omitted from the software specification because they are not known yet. Sometimes, however, there is some assumption made about how this will occur, such as it will be periodic or it will be done strictly by interrupt, etc. There must be some view or forum for documenting and expressing these "details."

C is the set of all components included in the model. These will generally be system components, although for some particular applications it may be appropriate to break up the system components at this time and represent each sub-component in the set C.

There are some differences between the two kinds of declarations. An output will not have a source; rather it will have a destination. An input source may be the component that is accepting this input, and, similarly, the destination may be the same component that results in the output. Exceptions are included with input declarations only and are discussed in section 5.3.

5.2 Mappings and Rules

According to Parnas [PvSK88], there is some type of mapping from the environment (env) to itself: $f(env) \rightarrow env$. Then within that is a mapping (function) $f(I) \rightarrow O$ where I and O are input and output to and from the software respectively. The interface model also describes a mapping: $M(O) \rightarrow I$, but in a very different sense from the previous one, where f represents the internal function of the software. Here the mapping represents an intended exchange of information; i.e., O is the "supplier" for I.

To model the interface requires an awareness of incompatibilities between the exchange declarations. To provide compatibility, inputs and outputs are checked according to rules. Some rules for checking are generic, such as:

capacity: Every input has a *capacity* rate which is

$$\frac{\text{max \# of inputs that can be handled}}{\text{(per) time unit}}$$

load: Every non-continual output has a max *load* declared as a rate which is

$$\frac{\text{max \# of outputs that may be sent}}{\text{(per) time unit}}$$

To match these, the following rule is used:

$$\exists \text{ some mapping } M \ni \forall i \epsilon I \;\; \exists \text{ some output } o \mid M(o) \rightarrow i$$

$$\sum_{\{oy|M(oy)\rightarrow ix\}} load(oy) \;\; \leq \;\; capacity(ix)$$

(Note that this constraint is conservative.)

compatible timetypes: These help define a certain consistency for the declared assumptions under the mapping M:

$$timetype(M(y)) \leq timetype(y)$$

Then the following must hold true:

$$\forall input(ix)|(source(ix) = B,$$

$$\exists(oy)|destination(oy) = D \wedge M(oy) = ix$$

It is clear that ix must be generated within component D, and likewise, oy within component B.

Other generic rules are possible for time and value ranges.

Other specific rules are derived from the particular application systems being developed. These are actually statements of the requirements constraints that may be assumed with the information provided at this time. For example: $\forall input(ix)$ generated within component B, $time(ix) > 50$; or $\exists input(ix) | time(ix) < 10$.

External completeness in this model means that the closure property discussed in [JL89] holds for all system inputs and outputs included in the model. The attempt to define a "complete" system is an attempt to cover all possible situations that may arise in the environment components and in the software controller, even if these situations are highly unlikely. Of course, external completeness also implies the specified software system meets the allocated system specification.

Within the exchange descriptions is the information to establish this closure property. There is also more information. In particular there is information to perform certain safety analyses. These will be discussed in a forthcoming report.

5.3 Exceptions

This modeling activity is probably just the first in many layers towards a robust design for software that can handle undesired activity in the system. Component outputs have declared value and time ranges, and exception transitions should be defined for their violation within the component.

The software component knows exactly when output from it will occur, i.e., it has a way to directly control it within a range, but not so for input to the software component. Software can check for adherence to certain declared (assumed) ranges, but it cannot force input to software to occur as specified. These relate specifically to time and value exceptions.

Exceptions in the model are declared only for the inputs, i.e., exceptions are a characteristic of software inputs. Exception handlers of several types are needed to handle them, based on the severity. Examples are correct, alert human, replace, or ignore. Note that the middle two, alert human and replace, are from errors.

This work does not discuss all the various possibilities for exception handling. The interested reader is referred to [Goo75] for a thorough treatment. Exceptions are caused by some error in the output that is supplying this input through mapping M. It may denote a failure of the source component for this input. The following relation **AF**, affects, can be defined:

$$\textit{if}\ [source(ix) = c] \wedge [M(oy) = ix] \wedge [destination(oy) = d] \wedge [failure(c) \epsilon \text{ Error}],$$
$$\text{then } c\mathbf{AF}d.$$

Some variables (inputs) are noninteractive; i.e., their measurement, evaluation, and feedback can go on for each without worrying about affecting values of the other variables. When components and their outputs are noninteractive, they would be seen to have no relationship in the **AF** relation.

So then component descriptions including the software are black-box descriptions as far as possible. There are also input and output descriptions that belong to each component because they result from one of its states, events, or conditions. Most components have failure states that result in incorrect, or perhaps lack of, output. Those states can affect other inputs (i.e., the ones that match the outputs in the above mapping M). In order to limit the amount of analysis necessary, it is sufficient to analyze for failure consequences on affected subsets where a $failure(x) = f$ **AF** component B if $f \Rightarrow$ (B fails) or (B degrades) Perhaps one can say that:

$$source(ix) = B \wedge failure(B) \epsilon (Unsafe)$$

Analysis for safety in the interface model is being developed using this affects relation.

5.4 Example

An example of this interface model is given for the following system:

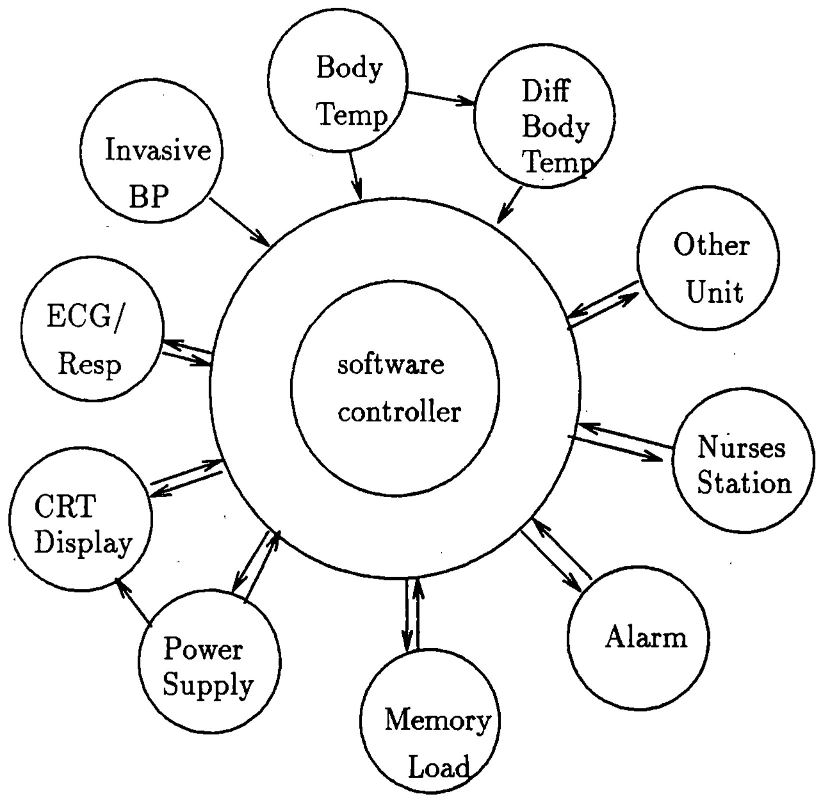
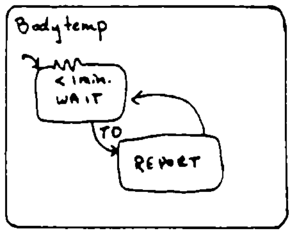


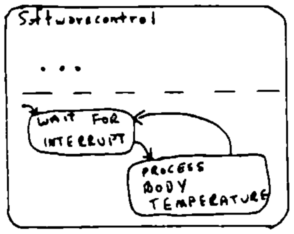
Figure 5

A patient monitoring system is being developed for a hospital ward. Each patient will be monitored by a separate unit with collectors for such factors as body temperature, differential body temperature, invasive blood pressure, and heart and breathing rate (ECG and Resp). The software updates these factors on a periodic basis and stores them in the data base for the patient being monitored. Invasive blood pressure will be reported whenever there is a significant change in the pressure; temperature and differential will interrupt with an update every minute; and ECG and Resp will be read continuously. Particular safe ranges for each factor will be specified individually for each patient as part of the patient data provided before start-up. If a factor falls outside of the safe range for the patient being monitored, a patient alarm will be sounded on the unit and in the nurses' station. This alarm, as well as an alarm for possible system malfunction, may be disabled with a manual button push if need be, for example, during patient transfer. This unit may accept data from and send data to a central data base, or it may receive updated information from the nurses' station. The monitor display will indicate whether or not the unit is on AC or batteries, and if on batteries, the amount of power left. There is enough memory in the unit to monitor the patient for 8 hours only. After that time the data base should be downloaded to central and the unit restarted.

Figure 5 shows a first view of how this system must interact.

There should be some notion of building the system. Begin with a null system and add components one at a time. If it is known what kind of analysis will be desired, it can then be done in stages, provided the properties are hierarchical, i.e., provided the properties are not lost once they have been proven for separate components which are now joined together. Consistency is this type of property. It is unfortunate that safety is not a hierarchical property. Figure 6 shows how the specification process begins for the output and input exchanges required for BT, body temperature. All times are in seconds. [Ranges] denote inclusive real ranges. Inputs will be named with "i" appended, outputs with "o" appended; "VAR′" will indicate the previous occurrence of VAR. Other abbreviations used are straightforward. The charts (state and exchange) will evolve as the interface is further specified. Each iteration requires the developer to check for adherence to the given rules or constraints. Note that some consistency checking is done *on the fly* as the inputs and outputs can be matched as they are defined and the corresponding elements seen to be consistent.





```
outputexchange(Bodytemp): BTo
   value(BTo) = [60,120]
   time(BTo) = [time(BTo')+59,time(BTo')+61]
   timetype(BTo) = periodic
   destination(BTo) = Swcontrol
   load = 1/60
```

```
inputexchange(Swcontrol): BTi
   value(BTi) = [60,120]
   time(BTi) = [time(BTi')+50,time(BTi')+70]
   timetype(BTi) = periodic
   source(BTi) = Bodytemp
   capacity(BTi) is 100/60
   exceptions:  If value or time is out of range, system
        malfunction alarm is to be set and nurses station
        notified.  Operation should continue.
        If capacity is exceeded, data will be lost.
```

Figure 6

A system overview Statechart is provided in figure 7. Additional exchanges and detailed Statecharts are given in figures 9 and 8, respectively. The mapping of inputs and outputs in the system is shown in figure 10. In order to provide a simple example, no startup elements have been included. To include the details of startup (or shutdown) would add more states, but would not add to the relevant information in the model. One particular element would be harder with startup included. That is, there is no previous time of a previous such event and no value of same. The first iteration could either be set to a physical time, or allowable default values could be automatically used for startup states. In this particular example, only the previous values and times are used. The problem would be handled similarly if more than these were needed.

Software functionality for the controller is not described in detail in this example. The requirements specification model would have less abstraction, but this example does not need that information. A first look at the interface for a developing system may not need to focus on software functionality at all either. Details of startups and internals of software control with a Statecharts model of a similar patient monitor are in [Mel88]).

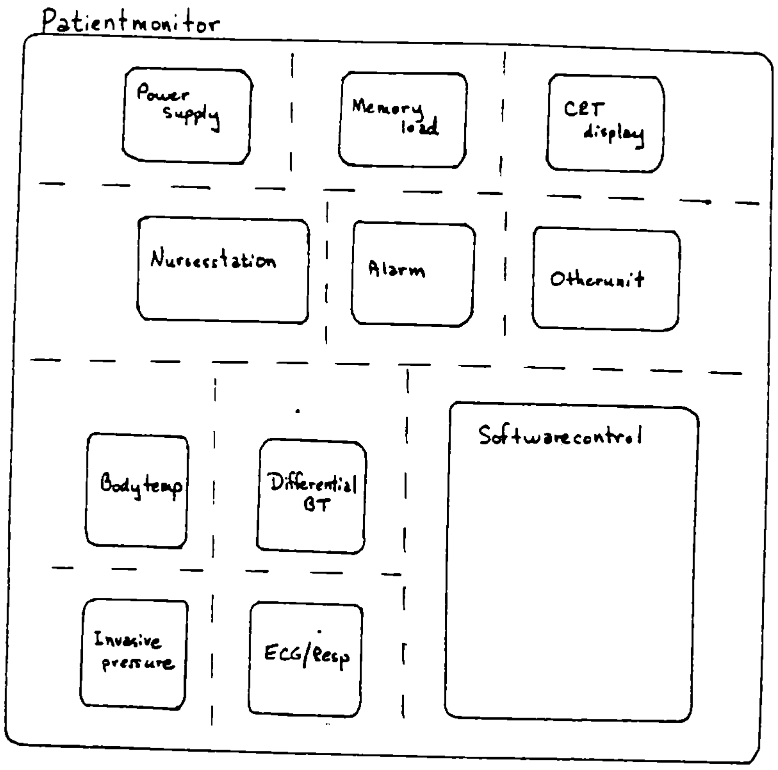


Figure 7

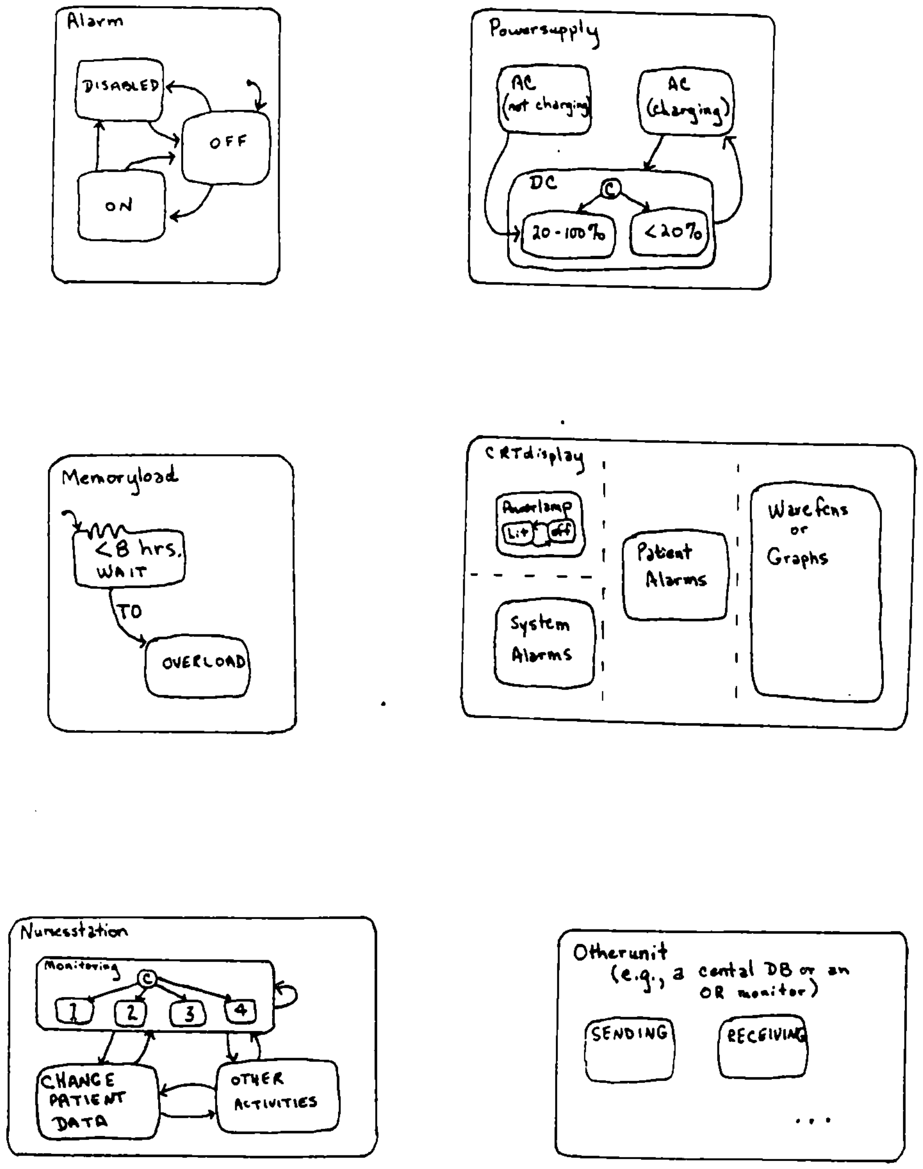

Alarm
DISABLED
OFF
ON
Powersupply
AC (not charging)
AC (charging)
DC
C
20-100%
<20%
Memoryload
<8 hrs.
WAIT
TO
OVERLOAD
CRTdisplay
Powerlamp
Lit
off
System Alarms
Patient Alarms
Waveforms or Graphs
Nursestation
Monitoring
C
1
2
3
4
CHANGE PATIENT DATA
OTHER ACTIVITIES
Otherunit
(e.g., a central DB or an OR monitor)
SENDING
RECEIVING
...

Figure 8a

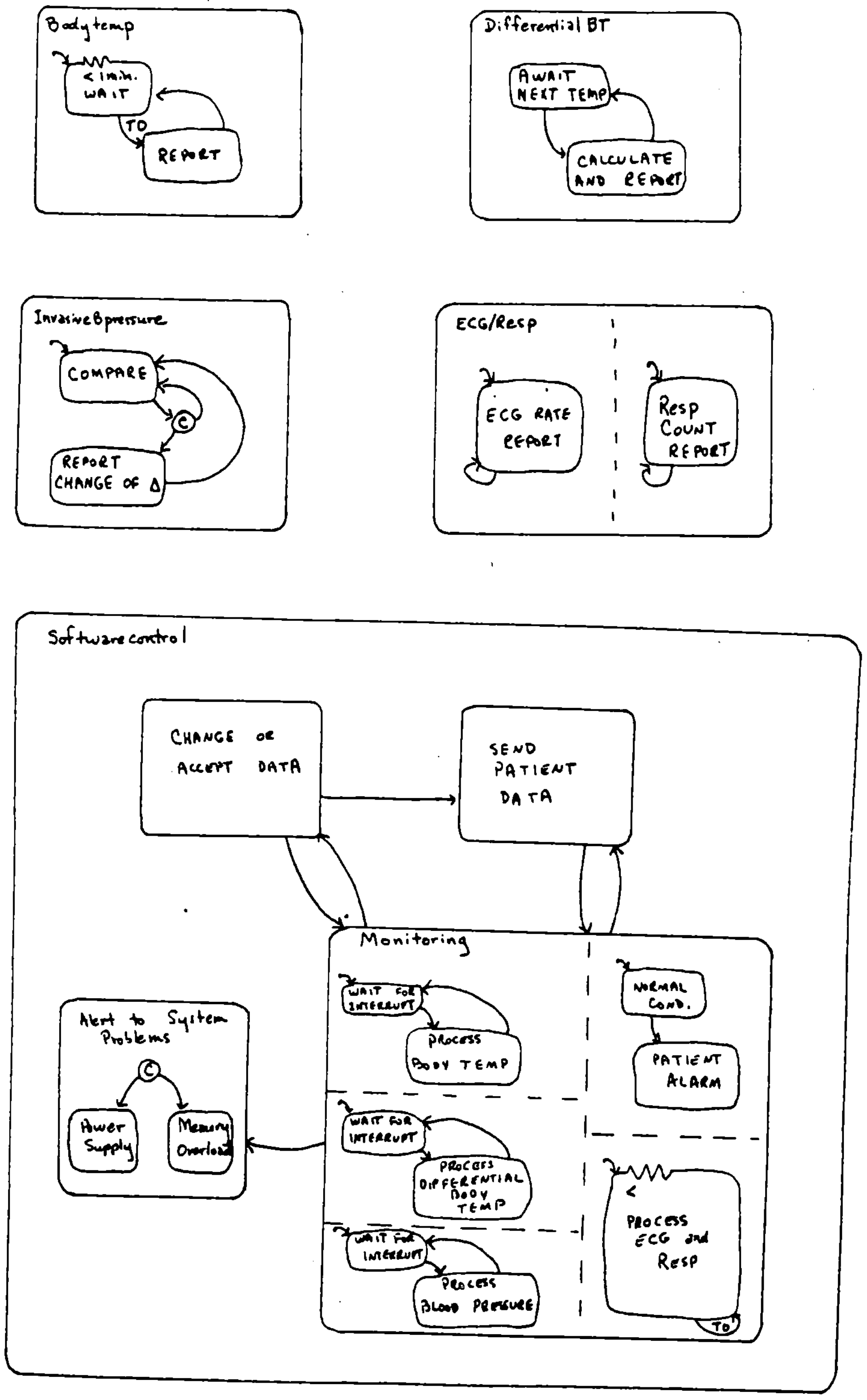

Bodytemp
< 1 min. WAIT
TO
REPORT
Differential BT
AWAIT NEXT TEMP
CALCULATE AND REPORT
InvasiveBpressure
COMPARE
C
REPORT CHANGE OF Δ
ECG/Resp
ECG RATE REPORT
Resp COUNT REPORT
Softwarecontrol
CHANGE OR ACCEPT DATA
SEND PATIENT DATA
Monitoring
WAIT FOR INTERRUPT
PROCESS BODY TEMP
NORMAL COND.
PATIENT ALARM
WAIT FOR INTERRUPT
PROCESS DIFFERENTIAL BODY TEMP
WAIT FOR INTERRUPT
PROCESS BLOOD PRESSURE
<
PROCESS ECG and Resp
TO
Alert to System Problems
C
Power Supply
Memory Overload

Figure 8b

```
outputexchange(Memoryload): OVLo
   value(OVLo) = {(memload,t) | t∈time(OVLo)}
   time(OVLo) = [28680,28860]
   timetype(OVLo) = S-R
   destination(OVLo) = Swcontrol
   load = 1/28800


inputexchange(Swcontrol): INTi
   value(INTi) = {(memload,t) | 0<t<28900} ∪ {(batlow,r) | 0≤r≤20}
   time(INTi) = [14400,28865]
   timetype(INTi) = S-R
   source(INTi) = Memoryload ∨ Powersupply
   capacity = 5/60
   exceptions:  If time is < 4hrs, indication will be of
        malfunction causing memoryoverload or malfunction
        with battery.  Recommend manual switch to battery.

outputexchange(Powersupply): PSo
   value(PSo) = {(batlow,r) | 0≤r≤20}
   time(PSo) = [14400,28800]
   timetype(PSo) = S-R
   destination(PSO) = Swcontrol
   load = 4/3600


outputexchange(Powersupply): ACo
   value(ACo) = {"onAC"}
   time(ACo) = [0,28800]
   timetype(ACo) = S-R
   destination(ACO) = CRTdisplay
   load(ACo) = 1/60


inputexchange(CRTdisplay): ACi
   value(ACi) = {"onAC","offAC"}
   time(ACi) = [0,28800]
   timetype(ACi) = S-R
   source(ACi) = Powersupply
   capacity = 1/1
   exceptions:  no exceptions


outputexchange(Nursesstation): PDUo
   value(PDUo) = [various patient data possible]
   time(PDUo) = [0,28800]
   timetype(PDUo) = S-R
   destination(PDUo) = Swcontrol
   load(PDUo) = 1/30


outputexchange(Otherunit): PDTo
   value(PDTo) = [patient data]
   time(PDTo) = [0,32400]
   timetype(PDTo) = S-R
   destination(PDTo) = Swcontrol
   load(PDTo) = 1/420


inputexchange(Swcontrol): DTi
   value(DTi) = [patient data]
   time(DTi) = [0,32400]
   timetype(DTi) = S-R
   source(DTi) = Nursesstation ∨ Otherunit
   capacity(DTi) = 3/60
   exceptions:  If capacity is exceeded, data will be lost.
        Also, Otherunit input may cause loss of data currently
        stored for patient.  Patient data out of expected
        ranges will not be accepted.


outputexchange(Swcontrol): PDo
   value(PDo) = [patient data]
   time(PDo) = [0,32400]
   timetype(PDo) = S-R
   destination(PDo) = Otherunit
   load(PDo) = 1/420


inputexchange(Otherunit): PDi
   value(PDi) = [patient data]
   time(PDi) = [0,32400]
   timetype(PDi) = S-R
   source(PDi) = Swcontrol
   capacity(PDi) = 1/60
   exceptions:  If capacity is exceeded, data will be lost.
        Patient data out of expected ranges will not be
        accepted.
```

```
outputexchange(ECG/Resp): ECGo
   value(ECGo) = [0,150]
   time(ECGo) = [time(ECGo')+.01,time(ECGo')+.1]
   timetype(ECGo) = continual
   destination(ECGo) = Swcontrol
   load = Not relevant (i.e., not interrupt driven)


inputexchange(Swcontrol): ECGi
   value(ECGi) = [0,150]
   time(ECGi) = [time(ECGi')+.005,time(ECGi')+.105]
   timetype(ECGi) = periodic
   source(ECGi) = ECG/Resp
   capacity = 500/1
   exceptions:  If data is not read fast enough, readout
        will appear to have gaps.  Values out of range
        indicate possible equipment malfunction.


outputexchange(ECG/Resp): RESPo
   value(RESPo) = [(x,y) | 0≤x≤25 ∧ 0≤y≤50]
   time(RESPo) = [time(RESPo')+.01,time(RESPo')+.1]
   timetype(RESPo) = continual
   destination(ECGo) = Swcontrol
   load = Not relevant (i.e., not interrupt driven)


inputexchange(Swcontrol): RESPi
   value(RESPi) = [(x,y) | 0≤x≤25 ∧ 0≤y≤50]
   time(RESPi) = [time(RESPi')+.005,time(RESPi')+.105]
   timetype(RESPi) = periodic
   source(RESPi) = ECG/Resp
   capacity = 500/1
   exceptions:  Values out of range indicate possible
        equipment malfunction.


outputexchange(DifferentialBT): DBTo
   value(DBTo) = [60,120]
   time(DBTo) = [time(DBTo')+55,time(DBTo')+70]
   timetype(DBTo) = periodic
   destination(DBTo) = Swcontrol
   load = 1/60


inputexchange(Swcontrol): DBTi
   value(DBTi) = [60,120]
   time(DBTi) = [time(DBTi')+45,time(DBTi')+75]
   timetype(DBTi) = periodic
   source(DBTi) = DifferentialBT
   capacity(DBTi) = 10/60
   exceptions:  If capacity exceeded, data will be
        lost and the integrity of all future inputs
        will be suspect (because this implies the DBT
        unit may have missed data).


outputexchange(Invasivepressure): BPo
   value(BPo) = [(x,y) | 80≤x≤170 ∧ 50≤y≤110]
   time(BPo) = [time(BPo')+.5,time(BPo')+10]
   timetype(BPo) = S-R
   destination(BPo) = Swcontrol
   load = 2/1


inputexchange(Swcontrol): Bpi
   value(BPi) = [(x,y) | 80≤x≤170 ∧ 50≤y≤110]
   time(BPi) = [time(BPi'),time(BPi')+300]
   timetype(BPi) = S-R
   source(BPi) = Invasivepressure
   capacity = 20/1
   exceptions:  If time is out of range, it may indicate
        equipment malfunction.  Unit should be queried.


outputexchange(Swcontrol):  ALo
   value(ALo) = {"onAL","offAL"}
   time(ALo) = [0,32400]
   timetype(ALo) = S-R
   destination(ALo) = Alarm
   load(ALo) = 1/1


inputexchange(Alarm): ALi
   value(ALi) = {"onAL","offAL"}
   time(ALi) = [0,32400]
   timetype(ALo) = S-R
   source(ALo) = Swcontrol
   capacity(ALo) = 2/1
   exceptions:  If value(ALi)=value(ALi'), there may be
        an error.  This should toggle.
```

Figure 9

Under mapping M:

$$\begin{array}{lcl}
\text{BTo} & \longrightarrow & \text{BTi} \\
\text{OVLo} & \longrightarrow & \text{INTi} \\
\text{ACo} & \longrightarrow & \text{ACi} \\
\text{PSo} & \longrightarrow & \text{INTi} \\
\text{PDUo} & \longrightarrow & \text{DTi} \\
\text{PDo} & \longrightarrow & \text{PDi} \\
\text{PDTo} & \longrightarrow & \text{DTi} \\
\text{ECGo} & \longrightarrow & \text{ECGi} \\
\text{RESPo} & \longrightarrow & \text{RESPi} \\
\text{DBTo} & \longrightarrow & \text{DBTi} \\
\text{BPo} & \longrightarrow & \text{BPi} \\
\text{ALo} & \longrightarrow & \text{ALi}
\end{array}$$

Figure 10

6 Summary

The external interface model is used to describe the overall process of inputting and outputting data in the system. This model begins at the system level and looks at interaction there—top down.

The declaration of ranges for exchanges are frequently (usually) declarations of assumptions that software developers must make early on about other components in the system. This model allows those assumptions to be documented and changed if need be as the system is developed. Even after development, certain components may be changed during maintenance. If assumptions and/or relevant information about these components is documented in an interface level and if the software is developed with this level used to hide those details from the rest of the software details, changeability will indeed be enhanced. This last was suggested and implemented, on a somewhat different scale and view, by Parker and others in [PHPS80]. Since every undertaking must begin somewhere, certain assumptions will have to be made at the beginning that may be erroneous, possibly even illogical. Without documenting the assumed traits and interactions, there is no way to reliably ensure their change at some later time.

It is frequently this beginning stage of software development that confuses those who must go from system level descriptions to high-level software design. A common desire is to have "something to throw over the wall" to the software designers that will provide a system view from the software perspective. This model actually represents a different viewpoint and approach to something that must have been talked about or accomplished before. It's just that no one has formalized the process or indeed identified it as a process that should be done. Ad hoc rules the day!

One way to describe systems at this level is with a technique such as that outlined here, augmented with simple scenarios and pictures. Contractual wording can also be available if that is wanted. The standard description for systems seems to have information scattered all over such a document, however, with the relationship between components not shown in any clear way. It is very difficult, if not impossible, to do a safety analysis on this type of description.

It is important to focus on analysis of system interaction during the software requirements specification process. This can reduce the complexity of some analyses because it eliminates non-essential details. It also makes it possible to visit some critical decisions earlier in the development process and puts decision making at the specification level, where it belongs—where decision makers still have a view of the entire system. Analysis techniques that rely on the interface model described here are currently being developed.

This interface level model includes a Statecharts model with exchange charts. It is an attempt at providing a different and simpler way of describing time, value, capacity, etc. Categorization of inputs and outputs and consistency maps between them are derived as part of the modeling process. More experience with this technique on some larger examples will be necessary before its practicality can be determined. It is likely to be practical for a rough first cut at developing embedded software for a system with a systems viewpoint.

References

[Cho87] Chin-Kuei Cho. *Quality Programming.* John Wiley & Sons, Inc., New York, 1987.

[Dei84] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1984.

[DoD85] DOD-STD-2167: *Military Standard for Defense System Software Development*, June 4, 1985.

[FA79] J. B. Fussell and J. S. Arendt. System reliability engineering methodology: A Discussion of the state of the art. *Nuclear Safety*, 20(5):541–550, September–October 1979.

[Goo75] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12), December 1975.

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

[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.

[JL89] Matthew S. Jaffe and Nancy G. Leveson. Completeness, robustness, and safety in real-time software requirements specification. Technical Report 89-01, Dept. of Information and Computer Science, University of California, Irvine, February 1989.

[Joh84] Curtis D. Johnson. *Microprocessor-based Process Control.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

[Lam85] Leslie Lamport. Problems from the workshop on the analysis of concurrent systems. In *The Analysis of Concurrent Systems*, pages 252–270. Springer-Verlag, 1985.

[LH83] Nancy G. Leveson and Peter. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[Low71] E. I. Lowe. *Computer Control in Process Industries.* Peter Peregrinus Ltd., London, 1971.

[Mel88] Bonnie E. Melhart. Specification languages for embedded systems: a Survey. Technical Report 88-17, Dept. of Information and Computer Science, University of California, Irvine, June 1988.

[Par84] Sybil P. Parker, editor. *McGraw-Hill Dictionary of Science and Engineering.* McGraw-Hill Book Company, New York, 1984.

[PHPS80] Robert A. Parker, Kathryn L. Heninger, David L. Parnas, and John E. Shore. *Abstract Interface Specifications for the A-7E Device Interface Module.* Naval Research Laboratory, Washington, D.C., November 1980.

[PvSK88] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation standards for safety critical software. Technical Report 88–220, Dept. of Computing & Information Science, Queen's University, Ontario, May 1988.

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