Specifying Software Requirements for Complex Systems: New Techniques and Their Application

KATHRYN L. HENINGER

Abstract—This paper concerns new techniques for making requirements specifications precise, concise, unambiguous, and easy to check for completeness and consistency. The techniques are well-suited for complex real-time software systems; they were developed to document the requirements of existing flight software for the Navy's A-7 aircraft. The paper outlines the information that belongs in a requirements document and discusses the objectives behind the techniques. Each technique is described and illustrated with examples from the A-7 document. The purpose of the paper is to introduce the A-7 document as a model of a disciplined approach to requirements specification; the document is available to anyone who wishes to see a fully worked-out example of the approach.

Index Terms—Documentation techniques, functional specifications, real-time software, requirements, requirements definition, software requirements, specifications.

I. Introduction

MUCH software is difficult to understand, change, and maintain. Several software engineering techniques have been suggested to ameliorate this situation, among them modularity and information hiding [11], [12], formal specifications [4], [9], [10], [13], [16], [20], abstract interfaces [15], cooperating sequential processes [2], [18], [21], process synchronization routines [2], [8], and resource monitors [1], [6], [7]. System developers are reluctant to use these techniques both because their usefulness has not been proven for programs with stringent resource limitations and because there are no fully worked-out examples of some of them. In order to demonstrate feasibility and to provide a useful model, the Naval Research Laboratory and the Naval Weapons Center are using the techniques listed above to redesign and rebuild the operational flight program for the A-7 aircraft. The new program will undergo the acceptance tests established for the current program, and the two programs will be compared both for resource utilization and for ease of change.

The new program must be functionally identical to the existing program. That is to say, the new program must meet the same requirements as the old program. Unfortunately, when the project started there existed no requirements documentation for the old program; procurement specifications, which were originally sketchy, are now out-of-date. Our first step was to produce a complete description of the A-7 program requirements in a form that would facilitate the development of the new program and that could be updated easily as the requirements continue to change.

Manuscript received February 12, 1979; revised March 12, 1979.

The author is with the Naval Research Laboratory, Washington, DC 20375.

Writing down the requirements turned out to be surprisingly difficult in spite of the availability of a working program and experienced maintenance personnel. None of the available documents were entirely accurate; no single person knew the answers to all our questions; some questions were answered differently by different people; and some questions could not be answered without experimentation with the existing system. We found it necessary to develop new techniques based on the same principles as the software design techniques listed above to organize and document software requirements. The techniques suggested questions, uncovered ambiguities, and supported crosschecking for completeness and consistency. The techniques allowed us to present the information relatively concisely, condensing several shelves of documentation into a single, 500-page document.

This paper shares some of the insights we gained from developing and applying these techniques. Our approach can be useful for other projects, both to document unrecorded requirements for existing systems and to guide software procurers as they define requirements for new systems. This paper introduces the techniques and illustrates them with simple examples. We invite anyone interested in more detail to look at the requirements document itself as a complete example of the way the techniques work for a substantial system [5].

First this paper addresses the objectives a requirements document ought to meet. Second it outlines the general design principles that guided us as we developed techniques; the principles helped us achieve the objectives. Finally it presents the specific techniques, showing how they allowed us to achieve completeness, precision, and clarity.

II. A-7 Program Characteristics

The A-7 flight program is an operational Navy program with tight memory and time constraints. The code is about 12 000 assembler languge instructions and runs on an IBM System 4 PI model TC-2 computer with 16K bytes of memory. We chose this program because we wanted to demonstrate that the run-time overhead incurred by using software engineering principles is not prohibitive for real-time programs and because

U.S. Government work not protected by U.S. copyright

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

the maintenance personnel feel that the current program is difficult to change.

The A-7 flight program is part of the Navigation/Weapon Delivery System on the A-7 aircraft. It receives input data from sensors, cockpit switches, and a panel with which the pilot keys in data. It controls several display devices in the cockpit and positions several sensors. Twenty-two devices are connected to the computer; examples include an inertial measurement set providing velocity data and a head-up display device. The head-up display projects symbols into the pilot's field of view, so that he sees them overlaying the world ahead of the aircraft. The program calculates navigation information, such as present position, speed, and heading; it also controls weapon delivery, giving the pilot steering cues and calculating when to release weapons.

III. Requirements Document Objectives

For documentation to be useful and coherent, explicit decisions must be made about the purposes it should serve. Decisions about the following questions affect its scope, organization, and style: 1) What kinds of questions should it answer? 2) Who are the readers? 3) How will it be used? 4) What background knowledge does a reader need? Considering these questions, we derived the following six objectives for our requirements document.

1) *Specify external behavior only.* A requirements document should specify only the external behavior of a system, without implying a particular implementation. The user or his representative defines requirements using his knowledge of the application area, in this case aircraft navigation and weapons delivery. The software designer creates the implementation, using his knowledge of software engineering. When requirements are expressed in terms of a possible implementation, they restrict the software designer too much, sometimes preventing him from using the most effective algorithms and data structures. In our project the requirements document must be equally valid for two quite different implementations: the program we build and the current program. For our purposes it serves as a problem statement, outlining what the new program must do to pass acceptance tests. For those maintaining the current program, it fills a serious gap in their documentation: they have no other source that states exactly what the program must do. They have pilot manuals, which supply user-level documentation for the entire avionics system, of which the program is only a small part. Unfortunately, the pilot manuals make it difficult to separate the activities performed by the computer program from those performed by other devices and to distinguish between advice to the pilot and restrictions enforced by the program. The maintainers also have implementation documentation for the current program: mathematical algorithm analyses, flowcharts, and 12 000 lines of sparsely commented assembler code. But the implementation documents do not distinguish between the aspects that are dictated by the requirements and those that the software designer is free to change.

2) *Specify constraints on the implementation.* In addition to defining correct program behavior, the document should describe the constraints placed on the implementation, especially the details of the hardware interfaces. As is usually the case with embedded systems,[1] we are not free to define the interfaces to the system, but must accept them as given for the problem. A complete requirements description should therefore include the facts about the hardware devices that can affect the correctness of the program.

3) *Be easy to change.* Because requirements change, requirements documentation should be easy to change. If the documentation is not maintained during the system life cycle, control is lost over the software evolution; it becomes difficult to coordinate program changes introduced by maintenance personnel.

4) *Serve as a reference tool.* The primary function of the document is to answer specific questions quickly, rather than to explain in general what the program does. We expect the document to serve experienced programmers who already have a general idea about the purpose of the program. Precision and conciseness are valued. Indispensable reference aids include a glossary, detailed table of contents, and various indices. Since tutorial material has different characteristics, such as a narrative style, it should be developed separately if it is needed.

5) *Record forethought about the life cycle of the system.* During the requirements definition stage, we believe it is sensible to exercise forethought about the life cycle of the program. What types of changes are likely to occur [22]? What functions would maintainers like to be able to remove easily [17]? For any software product some changes are easier to make than others; some guidance in the requirements will help the software designer assure that the easy changes correspond to the most likely changes.

6) *Characterize acceptable responses to undesired events.* Undesired events [14], such as hardware failures and user errors, should be anticipated during requirements definition. Since the user knows the application area, he knows more than the software designer about acceptable responses. For example, a pilot knows better than a programmer whether a particular response to a sensor failure will decrease or increase his difficulties. Responses to undesired events should be stated in the requirements document; they should not be left for the programmer to invent.

IV. Requirements Document Design Principles

Our approach to requirements documentation can be summarized by the three principles discussed below. These principles form the basis of all the techniques we developed.

1) *State questions before trying to answer them.* At every stage of writing the requirements, we concentrated first on formulating the questions that should be answered. If this is not done, the available material prejudices the requirements investigation so that only the easily answered questions are asked. First we formulated the table of contents in Fig. 1 in order to characterize the general classes of questions that should be answered. We wrote it before we looked at the A-7

[1]An embedded system functions as a component of a significantly larger system. Parnas [15] has a discussion of embedded system characteristics.

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

Chapter	Contents
0 Introduction	Organization principles; abstracts for other sections; notation guide
1 Computer Characteristics	If the computer is predetermined, a general description with particular attention to its idiosyncrasies; otherwise a summary of its required characteristics
2 Hardware Interfaces	Concise description of information received or transmitted by the computer
3 Software Functions	What the software must do to meet its requirements, in various situations and in response to various events
4 Timing Constraints	How often and how fast each function must be performed. This section is separate from section 3 since "what" and "when" can change independently.
5 Accuracy Constraints	How close output values must be to ideal values to be acceptable
6 Response to Undesired Events	What the software must do if sensors go down, the pilot keys in invalid data, etc.
7 Subsets	What parts of the program should be easy to remove
8 Fundamental Assumptions	The characteristics of the program that will stay the same, no matter what changes are made
9 Changes	The types of changes that have been made or are expected
10 Glossary	Most documentation is fraught with acronyms and technical terms. At first we prepared this guide for ourselves; as we learned the language, we retained it for newcomers.
11 Sources	Annotated list of documentation and personnel, indicating the types of questions each can answer

Fig. 1. A-7 Requirements table of contents.

at all, basing it on our experience with other software. Then we generated questions for the individual sections. Like any design effort, formulating questions requires iteration: we generated questions from common sense, organized them into forms, generated more questions by trying to fill in the blanks, and revised the forms.

2) *Separate concerns.* We used the principle of "separation of concerns" [3] to organize the document so that each project member could concentrate on a well-defined set of questions. This principle also serves the objective of making the document easy to change, since it causes changes to be well-confined. For example, hardware interfaces are described without making any assumptions about the purpose of the program; the hardware section would remain unchanged if the behavior of the program changed. The software behavior is described without any references to the details of the hardware devices; the software section would remain unchanged if data were received in different formats or over different channels.

3) *Be as formal as possible.* We avoided prose and developed formal ways to present information in order to be precise, concise, consistent, and complete.

The next two sections of the paper show how these principles are applied to describe the hardware interfaces and the software behavior.

V. Techniques for Describing Hardware Interfaces

Organization by Data Item

To organize the hardware interfaces description, we have a separate unit, called a *data item*, for each input or output that changes value independently of other inputs or outputs. Examples of input data items include barometric altitude, radar-measured distance to a point on the ground, the setting of the inertial platform mode switch, and the inertial platform ready signal. Examples of output data items include coordinates for the flight path marker on the head-up display, radar antenna steering commands, and the signal that turns on and off the computer-failed light. The A-7 computer receives 70 input data items and transmits 95 output data items.

In order to have a consistent approach, we designed a form to be completed for each data item. We started with an initial set of questions that occurred to us as we read about the interfaces. How does the program read or write these data? What is the bit representation of the value? Can the computer tell whether a sensor value is valid? As we worked on specific data items, new questions occurred to us. We added these questions to the form, so that they would be addressed for all data items. The form is illustrated in Figs. 2 and 3 at the end of this section.

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

Symbolic Names for Data Items and Values

The hardware section captures two kinds of information about data items: *arbitrary details* that might change if a device were replaced with a similar device, and *essential characteristics* that would be shared by similar devices. The bit representation of a value is an arbitrary detail; the semantics of the value is an essential characteristic. For example, any barometric altitude sensor provides a reading from which barometric altitude can be calculated—this information is essential. But the resolution, representation, accuracy, and timing might differ between two types of barometric altitude sensors—this information is arbitrary.

Essential information must be expressed in such a way that the rest of the document can use it without referencing the arbitrary details. For example, each data item is given a mnemonic name, so that it can be identified unambiguously in the rest of the document without reference to instruction sequences or channel numbers. If a data item is not numerical and takes on a fixed set of possible values, the values are given mnemonic names so that they can be used without reference to bit encodings. For example, a switch might be able to take the values "on" and "off." The physical representation of the two values is arbitrary information that is not mentioned in the rest of the document in case it changes. The names allow the readers and writers of the rest of the document to ignore the physical details of input and output, and are more visually meaningful than the details they represent.

We bracket every mnemonic name in symbols indicating the item type, for example /input-data-items/, //output-data-items//, and $nonnumeric-values$. These brackets reduce confusion by identifying the item type unambiguously, so that the reader knows where to find the precise definition. Moreover, the brackets facilitate systematic cross referencing, either by people or computers.

Templates for Value Descriptions

The values of the numerical data items belong to a small set of value types, such as angles and distances. At first we described each data item in an ad hoc fashion, usually imitating the descriptions in the documents we referenced. But these documents were not consistent with each other and the descriptions were not always complete. We made great progress when we developed informal templates for the value descriptions, with blanks to be completed for specific data items. For example, the template for angles might read:

> angle (?) is measured from line (?) to line (?) in the (?) direction, looking (?)

For example, <u>magnetic heading</u> is measured from <u>the line from the aircraft to magnetic north</u> to <u>the horizontal component of the aircraft X axis</u>, in the <u>clockwise</u> direction looking <u>down</u>.

Although templates were not used as hard-and-fast rules, their existence made values easier to describe, made the descriptions consistent with each other, and helped us apply the same standards of completeness to all items of the same type.

Input Data Items Described as Resources, Independent of Software Use

When describing input data items, we refrain from mentioning how or when the data is used by the software, to avoid making any assumptions about the software function. Instead, we describe the input data items as if taking inventory of the resources available to solve a problem. We define numerical values in terms of what they measure. For example, the value of the input data item called /RADALT/ is defined as the distance above local terrain as determined by the radar altimeter. Many nonnumerical inputs indicate switch positions; these are described without reference to the response the pilot expects when he changes the switch, since the response is accomplished by the software. For example, when the pilot changes the scale switch on the projected map display, he expects the map scale to change. Since the response is achieved by the software, it is not mentioned in the input data item description, which reads, *"/PMSCAL/ indicates the position of a two-position toggle switch on the projected map panel. This switch has no hardware effect on the projected map display."*

Example of an Input Data Item Description

Fig. 2 shows the completed form for a nonnumerical input data item. The underlined words are the form headings. <u>Value encoding</u> shows how the mnemonic value names used in the rest of the document are mapped into specific bit representations. "Switch nomenclature" indicates the names of the switch positions as seen by the pilot in the cockpit. <u>Instruction sequence</u> gives the TC-2 assembler language instructions that cause the data to be transmitted to or from the computer. We are not usurping the programmer's job by including the instruction sequence because there is no other way to read in this data item—the instruction sequence is not an implementation decision for the programmer. The channel number is a cross reference to the computer chapter where the general characteristics of the eight channels are described. <u>Data representation</u> shows the location of the value in the 16-bit input word. Notice how the <u>Comments</u> section defines the value assumed by the switch while the pilot is changing it. This is an example of a question we asked about all switches, once it had occurred to us about this one.

Output Data Items Described in Terms of Effects on External Hardware

Most output data items are described in terms of their effects on the associated devices. For example, the description of the output data items called //STEERAZ// and //STEEREL// shows how they are used to communicate the direction to point the antenna of the radar. This section does not explain how the software chooses the direction. For other output data items we define the value the peripheral device must receive in order to function correctly. For example, the description of the output data item called //FPANGL// shows that the radar assumes the value will be a certain angle which

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

Input Data Item: IMS Mode Switch

Acronym: /IMSMODE/

Hardware: Inertial Measurement Set

Description: /IMSMODE/ indicates the position of a six-position rotary switch on the IMS control panel.

Switch nomenclature: OFF; GND ALIGN; NORM; INERTIAL; MAG SL; GRID

Characteristics of Values

Value Encoding:		
	\$Offnone\$	(00000)
	\$Gndal\$	(10000)
	\$Norm\$	(01000)
	\$Iner\$	(00100)
	\$Grid\$	(00010)
	\$Magsl\$	(00001)

Instruction Sequence: READ 24 (Channel 0)

Data Representation: Bits 3-7

Comments: /IMSMODE/ = \$Offnone\$ when the switch is between two positions.

Fig. 2. Completed input data item form.

Output Data Item: Steering Error

Acronym: //STERROR//

Hardware: Attitude Direction Indicator (ADI)

Description: //STERROR// controls the position of the vertical needle on the ADI. A positive value moves the pointer to the right when looking at the display. A value of zero centers the needle.

Characteristics of Values

Unit: Degrees

Range: -2.5 to +2.5

Accuracy: ± .1

Resolution: .00122

Instruction Sequence: WRITE 229 (Channel 7)
Test Carry Bit = 0 for request acknowledged
If not, restart

Data Representation: 11-bit two's complement number, bit 0 and bits 3-12
scale = 512/1.25 = 409.6
offset = 0

```
( )          (          INDICATED VALUE                         )
___  Not used ___ ___ ___ ___ ___ ___ ,___ ___ ___ ___  _0_  _0_  _0_
0    1    2   3   4   5   6   7   8   9   10  11  12  13   14   15
BIT
```

Timing Characteristics: Digital to DC voltage conversion. See Section 1.5.7.

Comments: The pointer hits a mechanical stop at ± 2.5 degrees.

Fig. 3. Completed output data item form.

it uses to determine the climb or dive angle the aircraft should use during terrain following. We avoid giving any meaning to an output value that is not a characteristic of the hardware.

Example of an Output Data Item Description

Fig. 3 shows the completed form for a numerical output data item. Notice how the value is described in terms of its effect on a needle in a display, rather than in terms of what the needle is supposed to communicate to the pilot. The value is characterized by a standard set of parameters, such as range and resolution, which are used for all numerical data items. For Data representation, we show how the 16-bit output word is constructed, including which bits must be zero, which bits are ignored by the device, and which bits encode the output value. Since the actual output value is not in any standard units of measurement, we also show how it can be derived from a value in standard units, in this case degrees. The relation between output values and values in standard units is given by the equation

$$\text{output value} = \text{scale} \times (\text{standard value} + \text{offset})$$

Since the same equation is used for all numerical data items, we need only provide the scale and offset values for a particular data item. Thus the output value for the data item //STERROR// in Fig. 3 is derived from a value in degrees by the following expression:

$$\text{output value} = 409.6 \times (\text{standard value} + 0)$$

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

The Timing considerations section contains a pointer to another section; since many output data items have the same timing characteristics, we describe them once, and include cross references. The comment shows a physical limit of the device.

VI. Techniques for Describing Software Functions

Organization by Functions

We describe the software as a set of functions associated with output data items: each function determines the values for one or more output data items and each output data item is given values by exactly one function. Thus every function can be described in terms of externally visible effects. For example, the function calculating values for the output data item //STERROR// is described in terms of its effects on a needle in a display. The meaning conveyed to the pilot by the needle is expressed here.

This approach, identifying functions by working backward from output data items, works well because most A-7 outputs are specialized; most output data items are used for only a small set of purposes. The approach breaks down somewhat for a general-purpose device, such as a terminal, where the same data items are used to express many different types of information. We have one general-purpose device, the computer panel, where the same set of thirteen seven-segment displays can display many types of information, including present position, wind speed, and sensor status. We handled this situation by acting as if each type of information had its own panel, each controlled by a separate function. Thus, we have forty-eight panel functions, each described as if it always controlled a panel, and a set of rules to determine which function controls the real panel at any given moment. This approach, creating *virtual panels*, allows us to separate decisions about what the values are from decisions about when they are displayed. It also causes the description to be less dependent on the characteristics of the particular panel device than it otherwise would be.

Software functions are classified as either demand or periodic. A *demand function* must be requested by the occurrence of some event every time it is performed. For example, the computer-failed light is turned on by a demand function when a computer malfunction is detected. A *periodic function* is performed repeatedly without being requested each time. For example, the coordinates of symbols on the head-up display are updated by periodic functions. If a periodic function need not be performed all the time, it is started and stopped by specific events. For example, a symbol may be removed from the head-up display when a certain event occurs.

This distinction is useful because different performance and timing information is required for demand and periodic functions. To describe a demand function one must give the events that cause it to occur; an appropriate timing question is *"What is the maximum delay that can be tolerated between request and action?"* To describe a periodic function, one must give the events that cause it to start and stop and the conditions that affect how it is performed after it is started; an appropriate timing question is *"What are the minimum and maximum repetition rates for this function?"*

Output Values as Functions of Conditions and Events

Originally we thought we would describe each output as a mathematical function of input values. This turned out to be a naive approach. We found we could seldom describe output values directly in terms of input values; instead we had to define intermediate values that the current program calculated, but that did not correspond to any output values. These in turn had to be described in terms of other intermediate values. By the time we reached input values, we would have described an implementation.

Instead, we expressed requirements by giving output values as functions of aircraft operating conditions. For example, the output data item named //LATGT70// should change value when the aircraft crosses 70° latitude; how the program detects this event is left to the implementation. In order to describe outputs in terms of aircraft operating conditions, we defined a simple language of conditions and events. *Conditions* are predicates that characterize some aspect of the system for a measurable period of time. For example, /IMSMODE/ = $Gndal$ is a condition that is true when the IMS mode switch in the cockpit is set to the GND ALIGN position (see Fig. 2). If a pilot expects a certain display whenever the switch is in this position, the function controlling the display is affected by the value of /IMSMODE/. An *event* occurs when the value of a condition changes from true to false or vice versa. Events therefore specify instants of time, whereas conditions specify intervals of time. Events start and stop periodic functions, and they trigger demand functions. Events provide a convenient way to describe functions where something is done when a button is first pushed, but not if the pilot continues to hold it down. Before we distinguished clearly between events and conditions, situations of this sort were very difficult to describe simply.

Consistent Notation for Aircraft Operating Conditions

Text Macros: To keep the function descriptions concise, we introduced over two hundred terms that serve as text macros. The terms are bracketed in exclamation points and defined in an alphabetical dictionary. A text macro can define a quantity that affects an output value, but that cannot be directly obtained from an input. An example is "!ground track angle!", defined as "the angle measured from the line from the aircraft to true north to !ground track!, measured clockwise looking down." Although the derivation of such values is left to the implementation, text macros provide a consistent, encapsulated means to refer to them while specifying function values.

Text macros also serve as abbreviations for compound conditions that are frequently used or very detailed. For example, !Desig! is a condition that is true when the pilot has performed a sequence of actions that designates a target to the computer. The list of events defining !Desig! appears only in the dictionary; while writing or reading the rest of the docu-

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

ment, these events need not be considered. If designation procedures change, only the definition in the dictionary changes. Another example of a text macro for a compound condition is !IMS Reasonable!,[2] which represents the following bulky, specific condition:

!IMS total velocity! $\leqslant$ 1440 fps AND
change of !IMS total velocity! from .2 seconds
ago $\leqslant$ 50 fps

Even though this term is used many times in the function descriptions, only one place in the document need be changed if the reasonableness criteria change for the sensor.

The use of text macros is an application of stepwise refinement: while describing functions, we give names to complicated operating conditions or values, postponing the precise definitions. As the examples above show, we continue introducing new terms in the definitions themselves. This allows us to limit the amount of detail we deal with at one time. Furthermore, like the use of /, //, and $ brackets in the hardware descriptions, the use of ! brackets for text macros indicates to the reader that reference is being made to something that is defined precisely elsewhere. This reduces the risk of ambiguity that usually accompanies prose descriptions (e.g., !Desig! versus designated).

Conditions: We represent these predicates as expressions on input data items, for example, /IMSMODE/=$Gndal$, or expressions on quantities represented by text macros, for example, !ground track angle! = 30°. A condition can also be represented by a text macro, such as !IMS Reasonable!. Compound conditions can be composed by connecting simple conditions with the logical operators AND, OR, and NOT. For example, (!IMS Reasonable! AND /IMSMODE/=$Gndal$) is true only when both the component conditions are true.

Events: We use the notation @T(condition 1) to denote the occurrence of condition 1 becoming *true* and @F(condition 2) to denote the occurrence of condition 2 becoming *false.*

For example, the event @T(!ground track angle! < 30°) occurs when the !ground track angle! value crosses the 30° threshold from a larger value. The event @T(!ground track angle!=30°) occurs when the value reaches the 30° threshold from either direction. The event @T(/IMSMODE/ = $Gndal$) occurs when the pilot moves the switch to the GND ALIGN position. In some cases, an event only occurs if one condition changes when another condition is true, denoted by

@T(condition 3) WHEN (condition 4).

Thus, @T(/ACAIRB/=Yes) WHEN (/IMSMODE/=$Gndal$) refers to the event of the aircraft becoming airborne while the IMS mode switch is in the GND ALIGN position, while @T(/IMSMODE/=$Gndal$) WHEN (/ACAIRB/=Yes) refers to the event of the IMS mode being switched to GND ALIGN while the airplane is airborne.

[2]This text macro represents the condition that the values read from the inertial measurement set are reasonable; i.e., the magnitude of the aircraft velocity vector, calculated from inertial measurement set inputs, is less than or equal to 1440 feet per second and has changed less than 50 feet per second from the magnitude 0.2 seconds ago.

Using Modes to Organize and Simplify

Although each function is affected by only a small subset of the total set of conditions, we still need to organize conditions into groups in order to keep the function descriptions simple. To do this, we define *modes* or classes of system states. Because the functions differ more between modes than they do within a single mode, a mode-by-mode description is simpler than a general description. For example, by setting three switches, deselecting guns, and keying a single digit on the panel, the pilot can enter what is called the visual navigation update mode. In this mode, several displays and the radar are dedicated to helping him get a new position estimate by sighting off a local landmark. Thus the mode affects the correct behavior of the functions associated with these displays. The use of modes has an additional advantage: if something goes wrong during a flight, the pilot is much more likely when he makes the trouble report to remember the mode than the values of various conditions.

Each mode is given a short mnemonic name enclosed in asterisks, for example, *DIG* for Doppler-inertial-gyrocompassing navigation mode. The mode name is used in the rest of the document as an abbreviation for the conditions that are true whenever the system is in that mode.

The current mode is defined by the history of events that have occurred in the program. The document shows this by giving the initial mode and the set of events that cause transitions between any pair of modes. For example, the transition list includes the entry

```
*DIG* TO *DI*
   @T(!latitude! > 70°)
   @(/IMSMODE/=$Iner$) WHEN (!Doppler coupled!)
```

Thus the system will move from *DIG* mode to Doppler-inertial (*DI*) mode either if the aircraft goes above 70° latitude or if the inertial platform mode switch is changed to INERTIAL while the Doppler Radar is in use.

The table in Fig. 4 summarizes conditions that are true whenever the system is in a particular navigation mode. Thus in *DIG* mode the inertial platform mode switch is set to NORM, the aircraft is airborne, the latitude is less than 70°, and both the Doppler Radar and the inertial platform are functioning correctly. "X" table entries mean the value of that condition does not matter in that mode.

The mode condition tables are redundant because the information can be derived from the mode transition lists. However, the mode condition tables present the information in a more convenient form. Since the mode condition tables do not contain all the mode transition information, they do not uniquely define the current mode.

Special Tables for Precision and Completeness

In an early version of the document, function characteristics were described in prose; this was unsatisfactory because it was difficult to find answers to specific questions and because gaps and inconsistencies did not show up. We invented two types of tables that helped us express information precisely and completely.

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

MODE	/IMSMODE/	/ACAIRB/	!latitude!	Other
DIG	$Norm$	$Yes$	<70°	!IMS Up! AND !Doppler Up!
DI	$Norm$ OR $Iner$	$Yes$	<80°	!IMS Up! AND !Doppler Up! AND !Doppler Coupled!
I	$Iner$	X	<80°	!IMS Up!
IMS fail	X	X	X	!IMS Down!

Fig. 4. Section from the navigation mode condition table.

Condition Table: Magnetic heading (//MAGHDGH//) output values

MODES	CONDITIONS	
DIG, *DI*, *I* *Mag sl*,*Grid*	Always	X
IMS fail	(NOT /IMSMODE/=$Offnone$)	/IMSMODE/=$Offnone$
//MAGHDGH// value	angle defined by /MAGHCOS/ and /MAGHSIN/	0 (North)

Fig. 5. Example of a condition table.

Condition tables are used to define some aspect of an output value that is determined by an active mode and a condition that occurs within that mode. Fig. 5 gives an example of a condition table. Each row corresponds to a group of one or more modes in which this function acts alike. The rows are mutually exclusive; only one mode affects the function at a time. In each row are a set of mutually exclusive conditions; exactly one should be true whenever the program is in the modes denoted by the row. At the bottom of the column is the information appropriate for the interval identified by the mode-condition intersection. Thus to find the information appropriate for a given mode and given condition, first find the row corresponding to the mode, find the condition within the row, and follow that column to the bottom of the table. An "X" instead of a condition indicates that information at the bottom of the column is never appropriate for that mode.

In Fig. 5, the magnetic heading value is 0 when the system is in mode *IMS fail* and the condition (/IMSMODE/=$Offnone$) is true. Whenever the system is in *IMS fail* mode, the following condition is true, showing that the row is complete,

(/IMSMODE/=$Offnone$ OR(NOT /IMSMODE/=$Offnone$))

and the following statement is false, showing the row entries are mutually exclusive.

(/IMSMODE/=$Offnone$ AND(NOT/IMSMODE/=$Offnone$))

Condition tables are used in the descriptions of periodic functions. Periodic functions are performed differently in different time intervals; the appropriate time interval is determined by the prevailing mode and conditions. Each row in the table completely characterizes the intervals within a mode that are meaningful for that function. The conditions must be mutually exclusive, and together they must describe the entire time the program is within the mode. These characteristics ensure that condition tables be complete, that is, all relevant intervals are indicated. They also ensure that condition tables be unambiguous, that is, given the aircraft operating conditions, the correct interval can be determined.

Event table : When AUTOCAL Light Switched on/off

MODES	EVENTS	
Lautocal *Sautocal*	@T(In mode)	@F(In mode)
ACTION	//AUTOCAL//:=$On$	//AUTOCAL//:=$Off$

Fig. 6. Example of an event table.

Event tables show when demand functions should be performed or when periodic functions should be started or stopped. Each row in an event table corresponds to a mode or group of modes. Table entries are events that cause an action to be taken when the system is in a mode associated with the row. The action to be taken is given at the bottom of the column.

The event table in Fig. 6 specifies that the autocalibration light controlled by output data item //AUTOCAL// be turned

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

Demand Function Name: Change scale factor

Modes in which function required:
Lautocal, *Sautocal*, *Landaln*, *SINSaln*, *HUDaln*, *Airaln*

Output data item: //IMSSCAL//

Function Request and Output Description:

Event Table: When the Scale Factor Is Changed

MODES	EVENTS	
Lautocal *Landaln*	@T(In mode) WHEN (//IMSSCAL//=$Coarse$)	X
HUDaln	@T(In mode) WHEN (/IMSMODE/ = $Gndal$ AND //IMSSCAL//=$Coarse$)	@T(In mode) WHEN (NOT (/IMSMODE/=$Gndal$) AND //IMSSCAL//=$Fine$)
Sautocal *SINSaln* *Airaln*	X	@T(In mode) WHEN (//IMSSCAL//=$Fine$)
ACTION	//IMSSCAL//:=$Fine$	//IMSSCAL//:=$Coarse$

Fig. 7. Completed demand function form.

Periodic function name: Update Flight Path Marker coordinates

Modes in which function required:
DIG, *DI*, *I*, *Mag Sl*, *Grid*, *IMS fail*

Output Data Items: //FPMAZ//, //FPMEL//

Initiation and Termination Events:
Start: @T(//HUDVEL// = $On$)
Stop: @T(//HUDVEL// = $Off$)

Output description:

The Flight Path Marker (FPM) symbol on the head-up display shows the direction of the aircraft velocity vector. If the aircraft is moving straight ahead from the nose of the aircraft, the FPM is centered on the display. The horizontal displacement from display center shows the lateral velocity component and elevation displacement shows the vertical velocity component.
Although the means for deriving Flight Path Marker position varies as shown in the table below, the position is usually derived from the current !System velocities!. The velocities are first resolved into forward, lateral, and vertical components. Then FPM coordinates are derived in the following manner:

//FPMAZ// shows $\frac{\text{Lateral velocity}}{\text{Forward velocity}}$ //FPMEL// shows $\frac{\text{Vertical velocity}}{\text{Forward velocity}}$

Condition Table: Coordinates of the Flight Path Marker

MODES	CONDITIONS		
DIG, *DI*	X	Always	X
I	/ACAIRB/ = $No$	/ACAIRB/ = $Yes$	X
Mag sl, *Grid*	/ACAIRB/=$No$	!ADC Up! AND /ACAIRB/=$Yes$	!ADC Down! AND /ACAIRB/=$Yes$
IMS fail	/ACAIRB/=$No$	X	/ACAIRB/=$Yes$
FPM COORDINATES	//FPMAZ//:= 0 //FPMEL//:= 0	based on !System velocities!	//FPMAZ//:= 0 //FPMEL//:=/AOA/

Fig. 8. Completed periodic function form.

on when the two listed modes are entered and off when they are exited. We use the symbol ":=" to denote assignment. The event @T(In mode) occurs when all the conditions represented by the mode become true, i.e., when the mode is entered. @F(In mode) occurs when any one of the conditions represented by the mode becomes false, i.e., when the system changes to a different mode.

Function Description Examples

Figs. 7 and 8 illustrate the forms we created for demand and periodic functions, respectively. All function descriptions in-

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

dicate the associated output data items, thereby providing a cross reference to the hardware description. The list of modes gives the reader an overview of when the function is performed; the overview is refined in the rest of the description.

The event table in Fig. 7 shows both the events that request the function and the values output by the function at different times. For example, if the //IMSSCAL// value is $Coarse$ when the *Landaln* mode is entered, the function assigns it the value $Fine$. Notice how the table uses the symbolic names introduced in the hardware section for data items and data item values.

In Fig. 8 the initiation and termination section gives the events that cause this periodic function to start and stop. This function starts when another output data item, //HUDVEL//, is assigned the value On, and stops when //HUDVEL// is assigned the value Off. The function positions a symbol on a display device. The position of the symbol usually represents the direction of the aircraft velocity vector, but under some conditions the output data items are given other values. The output description consists of two parts: a brief prose description of the usual meaning of the symbol and a condition table that shows what will happen under different conditions. Notice that every mode in the mode list is accounted for in the table. The relevant conditions for this function are !ADC Up! or !ADC Down!, (the operating status of the air data computer sensor which provides a measurement of true airspeed) and /ACAIRB/= Yes and /ACAIRB/=No (whether the aircraft is airborne). Thus, if the system is in the inertial mode (*I*) and the aircraft is not airborne (/ACAIRB/=No is true), both coordinates of the symbol are set to zero.

VII. Techniques for Specifying Undesired Events

Lists of Undesired Events

In order to characterize the desired response of the system when undesired events occur, we started with a list of undesired events and interviewed pilots and maintenance programmers to find out both what they would like to have happen and what they considered feasible. The key was the list of possible undesired events. To derive this list, we used the classification scheme shown in Fig. 9 as a guide.

```
1 Resource Failure
     1.1 Temporary
     1.2 Permanent
2 Incorrect input data
     2.1 Detected by examining input only
     2.2 Detected by comparison with internal data
     2.3 Detected by user realizing he made a mistake
     2.4 Detected by user from incorrect output
3 Incorrect internal data
     3.1 Detected by internal inconsistency
     3.2 Detected by comparison with input data
     3.3 Detected by user from incorrect output
```

Fig. 9. Undesired event classification derived from Parnas [19].

For example, in the class "Resource failure–temporary," we include the malfunctioning of each sensor since the sensors tend to resume correct functioning; in the class "Resource failure–permanent," we include the loss of areas of memory.

VIII. Techniques for Characterizing Types of Changes

In order to characterize types of changes, we looked through a file of change requests and interviewed the maintainers. To define requirements for a new system, we would have looked at change requests for similar systems. We also made a long list of fundamental assumptions that we thought would always be true about the system, no matter what. In a meeting with several maintenance system engineers and programmers, all but four of the fundamental assumptions were rejected; each rejected assumption was moved to the list of possible changes! For example, the following assumption is true about the current program, but may change in the future: "The computer will perform weapon release calculations for only one target at a time. When a target is designated, the previously designated target is forgotten." By writing two complementary lists–possible changes and fundamental assumptions–we thought about the problem from two directions, and we detected many misunderstandings. Producing a list of fundamental assumptions forced us to voice some implicit assumptions, so that we discovered possible changes we would have omitted otherwise. One reason for the success of this procedure is that it is much easier for a reviewer to recognize an error than an omission.

Listed below are examples of feasible changes.

1) Assignment of devices to channels may be changed.

2) The rate of symbol movement on the display in response to joystick displacement might be changed.

3) New sensors may be added. (This has occurred already in the history of the program.)

4) Future weapons may require computer control after release.

5) Computer self-test might be required in the air (at present it is only required on the ground).

6) It may be necessary to cease certain lower priority functions to free resources for higher priority functions during stress moments. (At present the program halts if it does not have sufficient time to perform all functions, assuming a program error.)

IX. Discussion

We expect the document to be kept up-to-date as the program evolves because it is useful in many ways that are independent of our project. The maintainers of the current program plan to use it to train new maintenance personnel, since it presents the program's purpose in a consistent, systematic way. It is the only complete, up-to-date description of their hardware interfaces. One of the problems they now face when making changes is that they cannot tell easily if there are other places in the code that should be changed to preserve consistency. For example, they changed the code in one place to turn on a display when the target is twenty-two nautical miles away; in another place, the display is still turned on when the target is twenty nautical miles away. The unintended two-nautical-mile difference causes no major problems, but it adds unnecessary complexity for the pilot and the programmer. Inconsistencies such as this show up conspicuously in the function tables in our document. Besides using the document to check the implications of small

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

changes, the maintenance staff want to modify it to document the next version of the program. They expect major benefits as they prepare system tests, since the document provides a description of acceptable program behavior that is independent of the program. In the past, testers have had to infer what the program is supposed to do by looking at the code. Finally they also intend to derive test cases systematically from the tables and mode transition charts.

The usefulness of these ideas is not limited to existing programs. They could be used during the requirements definition phase for a new product in order to record decisions for easy retrieval, to check new decisions for consistency with previously made decisions, and to suggest questions that ought to be considered. However, a requirements document for a new system would not be as specific as our document. We can describe acceptable behavior exactly because all the decisions about the external interfaces have been made. For a new program a requirements document describes a set of possible behaviors, giving the characteristics that distinguish acceptable from unacceptable behavior. The system designer chooses the exact behavior for the new product. The questions are the same for a new system; the answers are less restrictive. For example, where we give a specific number for the accuracy of an input, there might be a range of acceptable accuracy values for a new program.

X. Conclusions

The requirements document for the A-7 program demonstrates that a substantial system can be described in terms of its external stimuli and its externally visible behavior. The techniques discussed in this paper guided us in obtaining information, helped us to control its complexity, and allowed us to avoid dealing with implementation details. The document gives a headstart on the design phase of our project. Many questions are answered precisely that usually would be left to programmers to decide or to discover as they build the code. Since the information is expressed systematically, we can plan for it systematically, instead of working each detail into the program in an ad hoc fashion.

All of the techniques described in this paper are based on three principles: formulate questions before trying to answer them, separate concerns, and use precise notation. From these principles we developed a disciplined approach including the following techniques:

symbolic names for data items and values
special brackets to indicate type of name
templates for value descriptions
standard forms
inputs described as resources
outputs described in terms of effects
demand versus periodic functions
output values given as functions of conditions and events
consistent notation for conditions and events
modes for describing equivalence classes of system states
special tables for consistency and completeness checking
undesired event classification
complementary lists of changes and fundamental assumptions.

This paper is only an introduction to the ideas that are illustrated in the requirements document [5]. The document is a fully worked-out example; no details have been left out to simplify the problem. Developing and applying the techniques required approximately seventeen man-months of effort. The document is available to anyone interested in pursuing the ideas. Most engineering is accomplished by emulating models. We believe that our document is a good model of requirements documentation.

Acknowledgment

The techniques described in this paper were developed by the author together with D. Parnas, J. Shore, and J. Kallander. The author thanks E. Britton, H. S. Elovitz, D. Parnas, J. Shore, and D. Weiss for their careful and constructive reviews of the manuscript.

References

[1] P. Brinch Hansen, *Operating Systems Principles.* Englewood Cliffs, NJ: Prentice-Hall, 1973.
[2] E. W. Dijkstra, "Co-operating sequential processes," in *Programming Languages*, F. Genuys, Ed. New York: Academic, 1968, pp. 43-112.
[3] —, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1977.
[4] J. V. Guttag, "Abstract data types and the development of data structures," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 396-404, June 1976.
[5] K. Heninger, J. Kallander, D. L. Parnas, and J. Shore, *Software Requirements for the A-7E Aircraft*, Naval Res. Lab., Washington, DC, Memo Rep. 3876, Nov. 27, 1978.
[6] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 549-557, Oct. 1974.
[7] J. Howard, "Proving monitors," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 273-279, May 1976.
[8] R. Lipton, *On Synchronization Primitive Systems*, Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1973.
[9] B. Liskov and S. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7-19, Mar. 1975.
[10] B. Liskov and V. Berzins, "An appraisal of program specifications," in *Proc. Conf. on Research Directions in Software Technology*, Oct. 10-12, 1977, pp. 13.1-13.24.
[11] D. L. Parnas, "Information distribution aspects of design methodology," in *Proc. Int. Fed. Inform. Processing Congr.*, Aug. 1971, vol. TA-3.
[12] —, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 1053-1058, Dec. 1972.
[13] D. L. Parnas and G. Handzel, *More on Specification Techniques for Software Modules*, Fachbereich Informatik, Technische Hochschule Darmstadt, Darmstadt, W. Germany, 1975.
[14] D. L. Parnas and H. Wurges, "Response to undesired events in software systems," in *Proc. 2nd Int. Conf. Software Eng.*, 1976, pp. 437-446.
[15] D. L. Parnas, *Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems*, Naval Res. Lab., Washington, DC, Rep. 8047, 1977.
[16] —, "The use of precise specifications in the development of software," in *Proc. Int. Fed. Inform. Processing Congr.*, 1977.
[17] —, "Designing software for ease of extension and contraction," in *Proc. 3rd Int. Conf. Software Eng.*, May 1978.
[18] D. L. Parnas and K. Heninger, "Implementing processes in HAS," in *Software Engineering Principles*, Naval Res. Lab., Washington, DC, course notes, 1978, Document HAS.9.

Authorized licensed use limited to: Konkuk University. Downloaded on September 22, 2009 at 03:32 from IEEE Xplore. Restrictions apply.

[19] D. L. Parnas, "Desired system behavior in undesired situations," in *Software Engineering Principles*, Naval Res. Lab., Washington, DC, course notes, 1978, Document UE.1.
[20] O. Roubine and L. Robinson, *SPECIAL Reference Manual*, Stanford Res. Inst., Menlo Park, CA, SRI Tech. Rep. CSL-45, SRI project 4828, 3rd ed., 1977.
[21] A. C. Shaw, *The Logical Design of Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
[22] D. M. Weiss, *The MUDD Report: A Case Study of Navy Software Development Practices*, Naval Res. Lab., Washington, DC, Rep. 7909, 1975.

Kathryn L. Heninger received the B.A. degree in English from Stanford University, Stanford, CA, in 1972, the M.S.L.S. degree in library science in 1975 and the M.S. degree in computer science in 1977, both from the University of North Carolina, Chapel Hill.

She is presently a Computer Scientist for the Information Systems Staff at the Naval Research Laboratory, Washington, DC. Her research interests include program design methodologies and parallel processing.

Notes on Type Abstraction (Version 2)

JOHN GUTTAG

Abstract—**This paper, which was initially prepared to accompany a series of lectures given at the 1978 NATO International Summer School on Program Construction, is primarily tutorial in nature. It begins by discussing in a general setting the role of type abstraction and the need for formal specifications of type abstractions. It then proceeds to examine in some detail two approaches to the construction of such specifications: that proposed by Hoare in his 1972 paper "Proofs of Correctness of Data Representations," and the author's own version of algebraic specifications. The Hoare approach is presented via a discussion of its embodiment in the programming language Euclid. The discussion of the algebraic approach includes material abstracted from earlier papers as well as some new material that has yet to appear. This new material deals with parameterized types and the specification of restrictions. The paper concludes with a brief discussion of the relative merits of the two approaches to type abstraction.**

Index Terms—**Abstract data types, abstraction, algebraic axioms, program verification, proof rules.**

Introduction

A KEY problem in the development of programs is reducing the amount of complexity or detail that must be considered at any one time. Two common and effective approaches to accomplishing this are decomposition and abstraction.

One decomposes a task by factoring it into subtasks each of which can be treated independently. Unfortunately, for many problems the smallest separable subtasks are still too complex to be mastered *in toto*. The complexity of such problems must be reduced via abstraction. By providing a mechanism for separating those attributes that are relevant in a given context from those that are not, abstraction serves to reduce the amount of detail that one need come to grips with at any one time.

One of the most significant aids to abstraction used in programming is the self-contained subroutine. It performs a specific, arbitrarily abstract, function by means of an unprescribed algorithm. Thus, at the level where it is invoked, it separates the relevant detail of "what" from the irrelevant detail of "how." Similarly, at the level of the implementation, it is usually unnecessary to complicate the "how" by considering the "why," i.e., the exact reasons for invoking a subroutine are rarely of concern to its implementor. By nesting subroutines, one may develop a hierarchy of abstractions.

Unfortunately, the nature of the abstractions that may be conveniently achieved through the use of subroutines is limited. Subroutines allow us to abstract single events. Their applicability is thus limited to problems that are conveniently decomposable into independent functional units. Type, or data, abstraction is not amenable to such an attack. To understand why, let us look briefly at the different roles played by type and procedural abstraction in programming and programming languages.

The Role of Abstraction

Imperative programming languages have components dealing with control flow, state change, and value generation. In recent years, a great deal has been written about the virtues and drawbacks of various control flow mechanisms. I will not add to that literature. Considerably less has been written about

Manuscript received February 12, 1979; revised June 4, 1979. This work was supported in part by the National Science Foundation under Grant MCS78-01798 and the Defense Advanced Research Projects Agency monitored by the Office of Naval Research under Contract N00014-75-C-0661.

The author is with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.