Published in the Proceedings of the 20th Digital Avionics Systems Conferene (DASC01), Daytona Beach, Florida, Oct. 14-18, 2001. Selected as best paper of the Systems Engineering Track.

EXTENDING THE FOUR-VARIABLE MODEL TO BRIDGE THE SYSTEM-SOFTWARE GAP

Steven P. Miller, Alan C. Tribble

Rockwell Collins, Cedar Rapids, Iowa 52498

Abstract

System and software requirements are inextricably intertwined, yet the challenge of extracting software requirements from system requirements is often an exercise in frustration. We propose an extension of the four-variable model originally developed by Parnas and Madey that clarifies how system requirements can be allocated between hardware and software. This extension organizes the software so that it traces clearly and directly to both the system and hardware requirements. An attractive aspect of this paradigm is that it is consistent with object-oriented approaches and allows the system requirements to be organized to facilitate object-oriented software development.

Introduction

Most complex systems consist of several subsystems that work closely together to achieve the overall objectives. Since each subsystem is itself a complex system built from hardware and software components, the process of collecting and managing requirements is usually done in phases. In a typical development, requirements are defined for the overall system, these system requirements are allocated to the subsystems, and the subsystem requirements are allocated to hardware and software components. In a large project, the subsystem requirements may be recursively allocated to smaller subsystems before being finally allocated to hardware and software.

Unfortunately, the step of allocating the subsystem requirements to the hardware and software frequently becomes an exercise in frustration. Comments one often hears include "almost all my subsystem requirements are allocated to software and almost none to hardware" or "all my subsystem requirements trace indirectly to every hardware component". This reflects the fact that hardware and software components are not parts of the subsystem in the same way that subsystems are parts of the overall system. Rather, software is used to build new virtual machines over the basic functionality provided by hardware. As a result, most subsystem requirements *do* trace almost entirely to the software requirements, since it is the software that extends the hardware to implement these requirements. At the same time, the system cannot be built without hardware, and almost every subsystem requirement is indirectly dependent on several hardware components.

In this paper, we propose a extension of a model of embedded systems originally developed by Parnas and Madey [1] that addresses many of these concerns and clarifies how system requirements can be allocated between hardware and software. This extension organizes the software requirements so that some trace directly to the subsystem requirements and the remainder to the hardware requirements. A particularly attractive aspect of this model is that it is consistent with current object-oriented paradigms and provides a natural way of organizing subsystem requirements to facilitate object-oriented development.

The Four-Variable Model

The original four-variable model proposed by Parnas and Madey evolved out of early efforts to specify the requirements for the A-7 aircraft in SCR [2]. This paradigm was extended by the Software Productivity Consortium in the Consortium Requirements Engineering (CoRE) methodology to include object-oriented concepts to make the specification robust in the face of change and to support the development of product families [3], [4]. The CoRE methodology was later used to specify the avionics system of the C-130J aircraft [5] and portions of the mode logic of the flight guidance system of a general aviation aircraft [6], [7]. Much additional work has been done by the

Naval Research Lab to formalize SCR and to provide supporting tools [8].

An overview of the four-variable model is shown in Figure 1. The variables in this model are continuous functions of time and consist of:

- *Monitored variables* (MON) in the environment that the system observes and responds to;
- *Controlled variables* (CON) in the environment that the system is to control;
- *Input variables* (INPUT) through which the software senses the monitored variables; and
- *Output variables* (OUTPUT) through which the software changes the controlled variables.

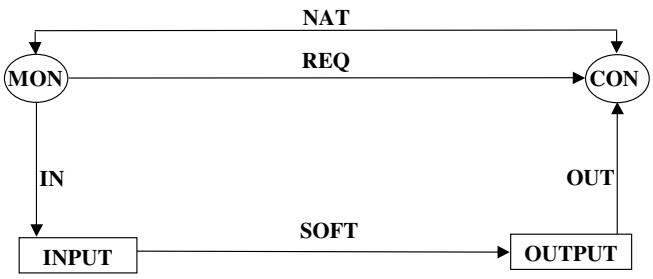

Figure 1 – Original Four-Variable Model

For example, monitored values might be the actual altitude of an aircraft and its airspeed while controlled variables might be the position of a control surface such as an aileron or the displayed value of the altitude on the primary flight display. The corresponding input and output values would be the ARINC-429 bus words that the software reads, or writes, to sense these quantities.

To complete the specification, four mathematical relations are defined between the variables:

- **NAT** defines the natural constraints imposed by the environment, such as the maximum rate of climb of an aircraft ;
- **REQ** defines the system requirements, specifying how the controlled variables are to respond to changes in the monitored variables;
- **IN** defines the relationship of the monitored variables to the input variables; and
- **OUT** defines relationship of the output variables to the controlled variables.

NAT and **REQ** describe how the controlled variables should change in response to changes in the monitored variables and define the subsystem view of the specification. **NAT** describes how the environment (the monitored and controlled variables) behaves in the absence of the system to be built, while **REQ** describes how the environment (the controlled variables) is to be constrained by the system. These relationships can be specified with mathematical precision, making them ideal for specifying safety-critical systems. The hardware interfaces surrounding the software are modeled by the **IN** and **OUT** relations that define how the input and output variables the software interacts with are related to the monitored and controlled environmental variables. Specification of the **NAT**, **REQ**, **IN**, and **OUT** relations implicitly bounds the allowed behavior of the software, shown in Figure 1 as **SOFT**, without specifying its design.

One of the great advantages of this model is that it explicitly defines the subsystem boundary through the identification of the monitored and controlled variables. If *MON* and *CON* are chosen correctly, **IN** and **OUT** will change only as the underlying hardware changes. At the same time, **REQ** changes only in response to changes in the subsystem requirements. Since customer driven changes and hardware driven changes often arise for different reasons, this helps to make the system more robust in the face of change.

Figure 1 is laid out as shown to emphasize that the *INPUT* and *OUTPUT* variables are at a lower level of abstraction than are the *MON* and *CON* variables, with the **IN** and **OUT** relationships mapping between these levels of abstraction. There are variations of the four-variable model that are useful on occasion. For example, it can be helpful to layer the **IN** and **OUT** relations into levels much like the ISO Reference Model for communication protocols. Another variation is to "glue" the controlled variables of one or more models to the monitored variables of another model to create a larger system specification, or to split a large model up into several smaller models.

Of course, for an actual system, there will dozens or hundreds of such variables and the relationships between them will be very complex. In one application of this approach, the

specification of the requirements for the C-130J avionics, there were over 1600 monitored and controlled variables [5]. Effectively organizing such a large specification is a daunting task in its own right, and details of how to do this well are beyond the scope of this paper. Additional information on this can be found in [5], [6], [7], [9], and [10].

The Extended Four-Variable Model

A weakness of the four-variable model is that it does not explicitly specify the software requirements, **SOFT**, but rather bounds it by specifying **NAT**, **REQ**, **IN**, and **OUT**. In fact, **SOFT** is often deliberately left unspecified to avoid constraining the developer. However, this leaves the software developer with the practical problem of how to structure the software and relate it to **NAT**, **REQ**, **IN** and **OUT**.

In older systems, one often finds the software implemented exactly as shown in Figure 1, i.e., as a direct mapping from input variables to output variables. In more recent systems, one finds hardware and software drivers that abstract away from the details of the input and output variables. This reverses in software part of the **IN** and **OUT** mappings in order to isolate the rest of the software from changes in the underlying hardware.

The proposed extension to the four-variable model takes this concept one step further and "stretches" the **SOFT** relationship into the relations **IN'**, **REQ'**, and **OUT'** as shown in Figure 2.

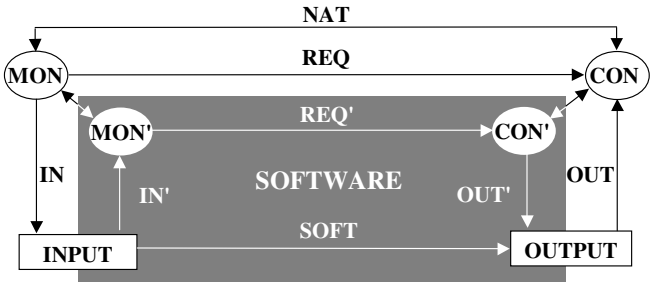

Figure 2 – Extended Four-Variable Model

Here, **IN'** and **OUT'** are nothing more than the specification of hardware drivers to be implemented in software. However, in addition to isolating the software from changes in the hardware, they also serve to recreate virtual versions of the monitored and controlled variables defined in the subsystem specification in the software, a technique often advocated in object-oriented approaches.

As we will see, one contribution of this model is that it helps to clarify the roles of **IN** and **OUT**, a common source of confusion. However, the most important contribution is that it makes the tracing of the subsystem requirements **REQ** to the software direct and straightforward. Each function defined in the subsystem requirement **REQ** maps directly into an identical function in the software requirement **REQ'**. In similar fashion, **IN'** and **OUT'** map directly to the hardware specification.

It is important to note that *MON'* and *CON'* are not the same as the system level variables represented by *MON* and *CON*. Small differences in value are introduced both by the hardware and software, and differences in timing are introduced when sensing and setting the input and output variables. For example, the value of an aircraft's altitude created in software is always going to lag behind and differ somewhat from the aircraft's true altitude. In safety-critical applications, the existence of these differences must be considered. However, if they are well within the tolerances of the system they can be treated as perturbations and Figure 2 provides an intuitive model relating the subsystem requirements to the software requirements.

A Small Example

This section illustrates the extended four-variable model by applying it to a small example, the Altitude Switch (ASW). While far simpler than an actual avionics component, the ASW is nicely suited for illustrating the four-variable model because most of its complexity lies in its **IN** and **OUT** relations.

The Altitude Switch receives altitude information from two digital altimeters and computes an estimate of the aircraft's altitude. When the aircraft descends below a threshold altitude, it turns on power to a Device of Interest (DOI). In this example the ASW also accepts a reset signal that returns it to its initial state and an inhibit signal that inhibits turning on the DOI. The example is specified in the style recommended in the CoRE methodology [3]. An overview of the

ASW is shown in the dependency diagram of Figure 3.

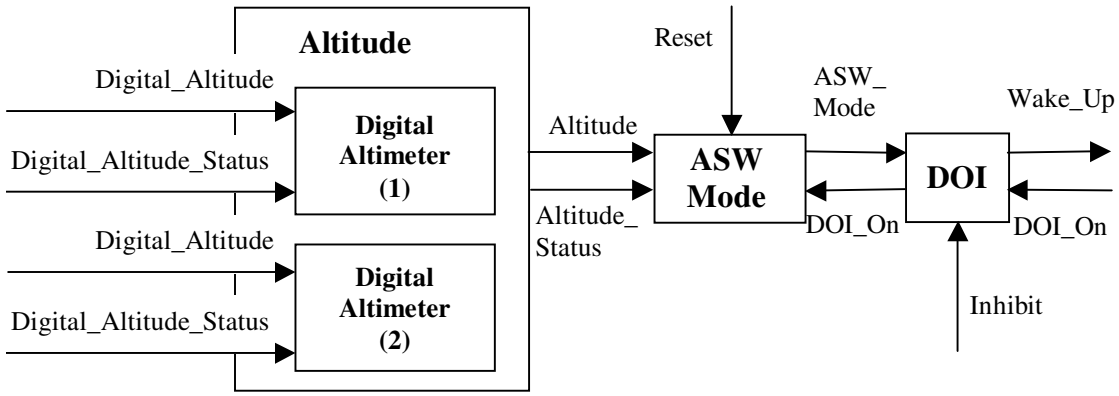

Figure 3 - Dependency Diagram of the Altitude Switch (ASW)

The main components are the CoRE classes of Altitude, ASW Mode, and DOI. The monitored variables are shown as arrows into the diagram, the controlled variables are shown as arrows out of the diagram, and intermediate values, or *terms* are shown as arrows between classes.

The Altitude class defines a term *Altitude* that is the estimate of the aircraft's true altitude constructed from the readings provided by the two digital altimeters, and a term *Altitude_Status* that indicates whether that estimate is valid. It has two internal classes, Digital Altimeter(1) and Digital Altimeter(2), representing two digital altimeters. Each digital altimeter defines two monitored variables, *Digital_Altitude* and *Digital_Altitude_ Status*.

The ASW Mode class defines the system modes of the ASW. For this example, these consist of only two modes (not shown), *DOI_Needed*, indicating that the DOI should be powered on, and *DOI_OK*, indicating that the DOI is either not needed or is already powered on. In a more realistic example, modes for system initialization, self-test, and error modes would also be defined. To transition between these two modes, the ASW Mode class makes use of the terms *Altitude* and *Altitude_Status* defined in the Altitude class, the *DOI_On* monitored variable defined in the DOI class, and the monitored variable *Reset* defined in ASW Mode.

The DOI class represents the ASW's view of the Device of Interest. It contains the definition of the only controlled variable, *Wake_Up*, representing whether the actual DOI should be powered on. It also defines the monitored variables *Inhibit*, indicating if *Wake_Up* should be supressed, and *DOI_On*, indicating if the DOI is powered on. The *DOI_On* monitored variable is passed on to the ASW Mode class as the term *DOI_On.*

While space does not permit giving the definition of each class in detail, the full specification of the Digital Altimeter class is shown in Figures 4 and 5 and the DOI class is given in Figures 6 and 7. The monitored variables *Altitude and Altitude_Status* are defined on the class's interface, indicating that they can be referenced by name in other classes. Their definition consists of their name, type, possible values, and a brief description. This is all any other class needs to know about the Digital Altimeter class.

Encapsulated within the Digital Altimeter class is the information needed to define how the input variables *Digital_Altimeter_Word*, *Digital_Altimeter_Input*, *Digital_Altimeter_Status_Input*, and *Digital_Altimeter_Label* are related to the monitored variables. These input variables are read by the software over an ARINC-429 bus [9], [10]. Since misunderstanding of the hardware interface is known to be a frequent source of errors [11], [12], the input variables are carefully defined by giving a brief description, details of their data representation, legal values, and information about how they are physically located in memory. In this case, examples and diagrams of word layouts are used to

make clear the relationships between individual bits, fields, and words.

Class Digital Altimeter(I: 1..2)

This class describes the interface to a Digital Altimeter. It exports the estimate of the aircraft's altitude sensed by the altimeter and that altitude's status. It encapsulates the physical details of the interface to the device

Name/Definition		Type	Values	Physical Interpretation
Digital_Altitude	Monitored	Real	-20.0..2,500.0	Distance in feet above ground level (AGL) as sensed by digital altimeter.
Digital_Altitude_ Status	Monitored	Enum	Invalid	Digital altitude is not valid.
			Valid	Digital altitude is valid.

Encapsulated Information

Input Variables

Digital_Altimeter_Word

Description ARINC-429 Word read from Bus I (I in [1..2])
Data Representation Bit[32]
Values Unconstrained
Data Transfer Word from Bus 1 located at memory address H'0014' - H'0017',

Word	H'0014'								H'0015'								H'0016'								H'0017'							
Bit	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01

Word from Bus 2 located at memory address H'0018' - H'001B',

Word	H'0018'								H'0019'								H'001A'								H'001B'							
Bit	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01

Digital_Altitude_Input

Description Altitude in feet AGL reported by digital altimeter I
Data Representation 17-bit 2's complement
Values [-8,192.0 .. +8,191.875]
Data Transfer Bits 29-13 of Digital_Altimeter_Word, sign in bit 29.

Bit	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
Value	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	1/2	1/4	1/8

Figure 4 – Definition of Class Digital Altimeter

Digital_Altitude_Status_Input

Description Status of altitude reported by digital altimeter I
Representation Bit[2]
Data Transfer Bits 31-30 of Digital_Altimeter_Word

Bits 31	Bits 30	Values	Meaning
0	0	FAIL	Failure Warning
0	1	NCD	No Computed Data
1	0	TEST	Functional Test
1	1	NORM	Normal Operation

Digital_Altitude_Label

Description Label of ARINC-429 Word I
Representation Octal, in reverse order
Values [000 - 377]
Data Transfer Bits 8-1 of Digital_Altimeter_Word
Example: representation of Octal Number 164.

Bit	8	7	6	5	4	3	2	1
Binary	1	0	0	1	1	0	0	1
Octal	4			6			1	

IN Relations

Digital_Altitude

Digital_Altitude_Input < -20	-20 <= Digital_Altitude_Input <= 2,500	Digital_Altitude_Input > 2,500
Altitude = -20.0	Altitude = Digital_Altitude_Input	Altitude =2,500.0

Digital_Altitude_Status

Parity(Digital_Altitude_Word) = EVEN **OR** Digital_Altitude_Label /= '164' **OR** Digital_Altitude_Status_Input /= NORM **OR** Digital_Altitude_Input < -20 **OR** Digital_Altitude_Input > 2,500	Parity(Digital_Altitude_Word) = ODD **AND** Digital_Altitude_Label= '164' **AND** Digital_Altitude_Status_Input = NORM **AND** -20 <= Digital_Altitude_Input **AND** Digital_Altitude_Input <= 2,500
Altitude_Status = Invalid	Altitude_Status = Valid

Figure 5 -Definition of Class Digital Altimeter (Continued)

The **IN** relations are then given (see Figure 5) to describe the relationship between the monitored variables *Digital_Altitude* and *Digital_Altitude_ Status* and these input variables. Looking at the four-variable model of Figure 1, one would expect the **IN** relations to describe each input variable as a function of the monitored variables. In practice, this is seldom practical.

For example, the physical representation of the *Digital_Altimeter_Input* input variable can range between -8,192.0 and 8,191.875 feet in increments 0.125 feet, while the monitored variable *Digital_Altitude* has been defined as a real number that ranges between –20 and 2,500 feet (see Figures 4 and 5). It is unclear how the input variable could ever take on a value outside the range of –20 to 2,500 feet, and it is even less clear what the system should do if such a value is seen.

The relationship of the monitored variable *Digital_Altitude_Status* to the input variables *Digital_Altitude_Status_Input* and *Digital_ Altitude_Label* is even more difficult to sort out. This is due to the fact that the monitored variable *Digital_Altitude_Status* does not represent a concrete physical quantity, but rather represents the system's ability to read the altitude produced by the digital altimeter, a difficult notion to define precisely.

These problems disappear if we define **IN** so that it describes each monitored variable as a function of the input variables, i.e., if we define **IN'** rather than **IN**. This is shown in Figure 5 where we define one **IN** relation for each monitored variable. The **IN** relation for *Digital_Altitude* maps values of *Digital_Altimeter_Input* outside the range of –20 to 2,500 feet to either –20 or 2,500 feet.

At the same time, the **IN** relation for *Digital_Altitude_Status* maps these out of range values to the value *Invalid*. Since our requirements model only needs to know if the monitored variable *Altitude* is valid or invalid, the IN relationship for *Altitude_Status* maps several possible physical errors to the value *Invalid*. In general, monitored variables that define the health or status of a physical quantity should only make distinctions that are actually used in the specification or likely to be used in future versions.

As mentioned earlier, an advantage of the extended four-variable model is that it clarifies the roles of **IN** and **OUT**. Conceptually, **IN** is the relationship we would like to specify, but practically, it is simpler and more useful to specify **IN'**. This is the relationship the software developer needs to implement. More importantly, the system and hardware information that is available usually makes it simpler to specify **IN'** than **IN**.

To illustrate the mapping of controlled variables to output variables, the definition of class DOI is given in Figures 6 and 7. The DOI class exports the definition of the controlled variable *Wake_Up* that indicates if the DOI is to be powered on and the monitored variable *DOI_On* that indicates if the DOI is already powered on. Since the monitored variable *Inhibit* is used only by the DOI class, its definition is encapsulated within the class.

Figure 6 also contains the definition of the **REQ** relation for the *Wake_Up* controlled variable. This relation uses the *ASW_Mode* defined in the ASW_Mode class (not shown). If the current value of *ASW_Mode* is *DOI_Needed* and the *Inhibit* monitored variable is false, Wake_Up takes on the value *True* indicating that the DOI should be powered on. The initiation delay and completion deadline state that the value of *Wake_Up* can be changed immediately in response to changes in the monitored variables, but must be completed within 50 miliseconds. The REQ relation for this example is almost trivial, but this is not normally the case. A larger and more realistic example can be found in [7].

Figure 7 shows the specification of the input and output variables for the DOI class and defines the **IN** and **OUT** relations mapping them to the monitored and controlled variables. Again, these are actually the specification of **IN'** and **OUT'**. The two input variables *DOI_Status_Input* and *Inhibit_ Input* are defined to be individual bits of a register, and the **IN** relations define how they are related to the monitored variables *DOI_On* and *Inhibit.* The output variable *DOI_Power_On_Output* is also defined as a bit of this register and the **OUT** relation defines how it is related to the *DOI_On* controlled variable.

Class DOI

This class defines the interface with the Device of Interest (DOI) that the Altitude Switch (ASW) is to control. The relevant portion of the DOI includes the signal to turn the DOI on and the signal from the DOI of whether the DOI is actually powered on. This class hides the details of the physical hardware interface to the DOI.

Class Interface

Name/Definition		Type	Values	Physical Interpretation
Wake_Up	Controlled	Bool	False	Do not apply power to the DOI.
			True	Apply power to the DOI
DOI_On	Monitored	Bool	False	DOI is not powered on.
			True	DOI is powered on.

Encapsulated Information

Name/Definition		Type	Values	Physical Interpretation
Inhibit	Monitored	Bool	False	Do not inhibit the DOI.
			True	Inhibit the DOI.

REQ Relations

Wake_Up Controlled Variable

Mode Class *ASW _Mode*

Mode	Conditions	
DOI_OK	INMODE	NEVER
DOI_Needed	Inhibit	not Inhibit
Wake_Up =	**False**	**True**

Initiation Delay 0 Milliseconds

Completion Deadline 50 Milliseconds

Figure 6 – Definition of Class DOI

In addition to the Digital_Altimeter and DOI classes, the full CoRE specification of the Altitude Switch also includes definitions for the Altitude and ASW_Modes classes. These are omitted here to save space, but the classes presented are sufficient to illustrate the four-variable model.

Given the full CoRE specification, it is straightforward to write the software that implements the specification. The developer defines virtual versions of the monitored and controlled variables *Digital_ Altitude, Digital_Altitude_Status, DOI_On, Inhibit*, and *Wake_Up*. Input routines are written that implement the inverse relation **IN'** to map the input variables into the virtual monitored variables, and output routines are written implementing **OUT'** to map the virtual controlled variables into the output variables. The developer directly implements the **REQ** relationships defined in the subsystem requirements model in software so that the virtual versions of the controlled variables

respond correctly to changes in the virtual versions of the monitored variables.

Hardware Interface Specification

Input Variables

DOI_Status_Input

Description Discrete signal indicating if the DOI is powered on.
Data Representation Bit[1]
Data Transfer Bit 2 of Register 1 (line STS)

Values		Interpretation
0b	Off	DOI is not powered on.
1b	On	DOI is powered on.

Inhibit_Input

Description Discrete signal used to inhibit the ASW.
Data Representation Bit[1]
Data Transfer Bit 3 of Register 1 (line IHB)

Values		Interpretation
0b	Norm	Do not inhibit the ASW.
1b	Inhibit	Inhibit the ASW.

IN Relations

DOI_On

DOI_Status_Input = Off	DOI_Status_Input = On
DOI_On = False	DOI_On = True

Inhibit

Inhibit_Input = Norm	Inhibit_Input = Inhibit
Inhibit = False	Inhibit = True

Output Variables

DOI_Power_Output

Description Discrete signal indicating if the DOI is to be turned on.
Data Representation Bit[1]
Data Transfer Bit 7 of Register 1 (line PWR)

Values		Interpretation
0b	Off	DOI is not to be turned on.
1b	On	DOI is to be turned on.

OUT Relations

Wake_Up

Wake_Up = False	Wake_Up = True
DOI_Power_Output = Off	DOI_Power_Output = On

Figure 7 – Definition of Class DOI (Continued)

The resulting system directly implements the extended four-variable model shown in Figure 2 by translating

- the monitored variables MON into the input variables INPUT via the hardware implementation of **IN**,
- the input variables INPUT into software versions of the monitored variables MON' via the software implementation of **IN'**,
- the changes in MON' into changes in the controlled variables CON' via the software implementation of **REQ'**,
- the software versions CON' of the controlled variables into the output variables OUTPUT via the software implementation of **OUT'**,
- the OUTPUT variables into the controlled variables CON via the hardware implementation of **OUT**.

Of course, the required result is to implement REQ, and the path through IN, IN', REQ', OUT', and OUT will introduce both timing delays and differences in value from REQ. These differences must be analyzed to ensure that the final implementation meets the timing and value tolerances of the system. Hopefully, these differences are considerably smaller than the allowed tolerances, making this task straightforward.

Conclusions

We have described an extension of the four-variable model of embedded systems that makes implementation of the software from the subsystem requirements straightforward. The most important contribution of this extension is that it organizes the software so that it traces clearly and directly to the subsystem and the hardware requirements. Since customer driven changes and hardware driven changes often arise for different reasons, this also helps to make the software more robust in the face of change. At the same time, this extension forces the recreation of images of physical environmental quantities in the software, an approach often advocated in object-oriented approaches. As shown in the example, such specifications can be easily organized in a manner compatible with object-oriented software development.

Acknowledgements

The authors wish to thank Dr. Stuart Faulk of the University of Oregon, Dr. Constance Heitmeyer of the Naval Research Laboratory, and Lisa Finneran and Howard Lykins of the Software Productivity Consortium for their assistance in applying SCR, CoRE, and the four-variable model.

Bibliography

[1] Parnas, D. L. and J. Madey, "Functional Documentation for Computer Systems Engineering, Vol. 2," McMaster University, Hamilton, Ontario, Technical Report CRL 237, September 1991.

[2] van Schouwen, A. J., "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems," Queens University, Hamilton, Ontario, Technical Report 90-276, 1990.

[3] Faulk, S. R., J. Brackett, P. Ward, and J. Kirby, Jr., "The CoRE Method for Real-Time Requirements," IEEE Software, Vol. 9, No. 5, pp. 22-33, September 1992.

[4] Faulk, S. R., L. Finneran, J. Kirby, and A. Moini, *Consortium Requirements Engineering Guidebook*, Technical Report SPC-92060-CMS, Herndon, VA: Software Productivity Consortium, December 1993.

[5] Faulk, S. R., L. Finneran, J. Kirby, S. Shah, and J. Sutton, "Experience Applying the CoRE Method to the Lockheed C-130J Software Requirements," *Proceedings of the Ninth Annual Conference on Computer Assurance*, Gaithersburg, MD, pp. 3-8, June 1994.

[6] Miller, S. P. and K. F. Hoech, "Specifying the Mode Logic of a Flight Guidance System in CoRE,", Rockwell Collins, Cedar Rapids, IA, Technical Report WP97-2011, August 1997.

[7] Miller, S. P., "Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR," *Second Workshop on Formal Methods in Software Practice (FMSP98)*, Clearwater Beach, FL, March 1998.

[8] Heitmeyer, C. L., J. Kirby, and B. G. Labaw, "Automated Consistency Checking of Requirements Specification," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 5, No. 3, pp. 231-261, July 1996. ARINC Characteristic 707-5, Radio Altimeter, Aeronautical Radio, Annapolis, MD, March 1993.

[9] ARINC Characteristic 707-5, Radio Altimeter, Aeronautical Radio, Annapolis, MD, March 1993.

[10] ARINC Specification 429-14, Mark 33 Digital Information Transfer System (DITS), Aeronautical Radio, Annapolis, MD, March 1993.

[11] Leveson, N. G., *Safeware: System Safety and Computers*, Reading, MA: Addison-Wesley Publishing Company, 1995.

[12] Lutz, R. R., "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems," *IEEE International Symposium on Requirements Engineering*, San Diego, CA, January 1993.