Specifying Software Behavior for Requirements and Design

James KIRBY JR.
Code 5542, Naval Research Laboratory
4555 Overlook Ave. SW,
Washington, DC 20375, U.S.

ABSTRACT

It would be useful to write one description of software behavior to serve both requirements and design. Having one description could reduce effort by eliminating the work of developing two descriptions and of keeping them consistent and relevant throughout development, evolution, and sustainment. It would also eliminate the inconsistency inherent in having two descriptions, a fertile source of error. A question paramount to software engineers is, Could one description of behavior for a real system serve both requirements and design? This paper answers that question by describing one such description of the software behavior of a real system.

Keywords: Software Requirements, Software Design, Reactive Systems, Computational Behavior, Software Methods

1. INTRODUCTION

A typical software development project produces several descriptions of software behavior. Often, natural language provides one description of behavior, while UML, data flow diagrams, and pseudocode provide others. Programmers encode software behavior in one or more of a variety of programming languages. Such redundant recording of software behavior is a significant source both of unnecessary work and of error. Often, the descriptions are in different languages, making it difficult to compare with one another for consistency. As discoveries of what the behavior should be are made during development, evolution, and sustainment, it is difficult to keep the various descriptions consistent; often, they are not. [13]

Reference [13] argues that it would be useful to write one description of required software behavior to serve both requirements and design. Having one description could reduce effort by eliminating the work of developing two descriptions and of keeping them consistent and relevant throughout development, evolution, and sustainment. It could also eliminate the inconsistency inherent in having two descriptions, a fertile source of error.

Software behavior[1] (also called "computational behavior") is the changes in value over time of quantities and qualities characterizing the system environment that the software controls or affects (e.g., positions of symbols on a head-up display, when to energize an actuator, whether to light an indicator or sound an audible signal [13]). Mathematical variables denote these quantities and qualities. Mathematical functions, whose domains comprise (1) variables denoting environmental quantities and qualities and (2) variables representing system state, specify the values of those variables. Each of these functions, which can be understood to specify behavior for requirements, can also serve design by specifying behavior of the software module responsible for providing that function.

While [13] describes a small example of a specification of behavior serving requirements and design, it does not address a question paramount to software engineers: Can the same be done for a real system? Answering that question is the purpose of this paper, which describes a *unified* specification of the software behavior of a real system intended to serve both requirements and design. [14]

2. COMPARING BEHAVIOR DESCRIPTIONS

In the late 1970s an NRL team embarked on a project whose goal was to redesign and reimplement the operational flight program (OFP) for the Navy's A-7E aircraft [19]. As part of that project, the team produced a set of publicly available development documents. While the A-7E software requirements [1] and design documents [7], [8], [15] are not written in widely differing languages, they provide redundant and quite different descriptions of the behavior of the A-7E OFP. This can be seen by comparing functions in the software requirements [1] and functions in the function driver module [7] of the design.

The A-7E documents use bracketing notation to indicate names and to distinguish different sorts of names. Table 1 lists bracketing notation adopted from the A-7E documents and used in this paper. Note that the table provides several interpretations for *!+term+!*.

The A-7E software requirements [1] use values written to physical device outputs to specify behavior. Functions of air-

1. While the notion of software behavior is not restricted to reactive systems that monitor and affect a physical environment, such reactive systems are the focus of this paper.

TABLE 1. Bracketing Notation

Brackets	Example	Interpretation
!+Term+!	!+Aud signal+!	Describes meaning of program parameter.
		Monitored or controlled variable, or term
!Requirements term!	!A/C facing target!	Monitored variable or term
!!Local term!!	!!time beeped!!	Local term
$Value$	$On$, $Off$, $True$, $False$	Value of enumerated variable
Mode	*A/A Manrip*	Mode
/Input/	/RE/	Input from physical device
//Output//	//BMBTON//	Output to physical device
+Program+	+S_AUDIBLE_SIGNAL+	Callable program

craft operating conditions (e.g., whether the aircraft is airborne) and of values read from physical device inputs specify those values. Table 2 illustrates device output //BMBTON//. [1] The Description field in Table 2 describes how to interpret the impact on the system environment of writing a particular value to output //BMBTON//: the associated device issues an audible tone in the aircraft cockpit when //BMBTON// = $On$. It is this impact on the system environment (an audible tone in the cockpit) that is software behavior. The values and devices are mechanisms to accomplish that behavior.

TABLE 2. Bomb Tone (BMBTON)

Output data item	Bomb Tone
Acronym	//BMBTON//
Hardware	Bomb Tone
Description	There is an audible tone in the cockpit when //BMBTOM// = $On$
Value encodings	$Off$ (0), $On$ (1)
Instruction Sequence	WRITE 8 (Channel 0)
Data representation	Discrete output word 1, bit 8

Table 3 illustrates physical device input /RE/. Analogous to physical device outputs, the Description field describes how the input reflects the system environment. When the pilot presses the release enable button on the pilot grip stick (PGS), device input /RE/ = $On$. [1]

TABLE 3. Release enable button

Input data item	Release Enable
Acronym	/RE/
Hardware	Pilot Grip Stick
Description	/RE/ indicates the position of a momentary contact push button switch on the PGS.
Value encodings	$Off$ (0), $On$ (1)
Instruction Sequence	READ 2 (Channel 0)
Data representation	Discrete output word 3, bit 8

Table 4 provides an example function, a subset taken from a function table in Section 4.4.3 *Switch Bomb Tone On/Off* [1]. It specifies required behavior by describing when the audible tone is on and when it is off (see ACTION row at bottom of table). The domain of the function comprises **modes**, *!requirements terms!*, */inputs/*, and *//outputs//*. The first column indicates that the behavior is mode-dependent. When the system is in any of the modes listed in the first column of the first row, the first row provides rules for turning the audible signal on (when the pilot presses release enable) and off (when the pilot releases release enable when the audible signal is sounding). @T(/RE/=$On$) indicates the event of the pilot pressing release enable. The conditioned event expression @T(/RE/=$Off$) WHEN (//BOMBTON//=$On$) indicates the event of the pilot releasing the release enable button when the *condition* holds that the audible signal is sounding.

TABLE 4. Switch Bomb Tone On/Off

MODES	EVENTS	
CCIP *Manrip*	@T(/RE/=$On$)	@T(/RE/=$Off$) WHEN (//BMBTON//=$On$)
A/AGuns	@T(/RE/=$On$) WHEN (!Rockets!)	@T(/BMBREL/=$On$) WHEN (//BMBTON//=$On$ OR @T(/RE/=$Off$) WHEN (//BMBTON//=$On$
ACTION	//BMBTON//:=$On$	//BMBTON//:=$Off$

The A-7E design provides *virtual* devices that encapsulate characteristics of physical devices. It provides programs that facilitate writing physical device outputs and reading physical device inputs [5], [15]. Table 5 describes two such programs. For both, the bracketed name (*!+term+!*) in the description field provides the meaning of the corresponding parameter and the behavior of the program in terms of the system environment. Table 6 defines the two *!+terms+!*, used in Table 5, which this paper subsequently reinterprets.

To describe behavior, the design specifies when to call virtual device programs and what values to pass to them. The program +S_AUDIBLE_SIGNAL+, which accepts one input parameter of type AUD_ind_cntrl, sounds and silences the audible signal in the A-7E cockpit. AUD_ind_cntrl is an enumerated type which allows values $On$, $Off$, and $Intermittent$, the latter of which is not used in this paper. The definition of !+Aud Signal+! in the dictionary extract in Table 6 indicates that the effect of the input parameter is to allow the caller of +S_AUDIBLE_SIGNAL+ to control the audible signal, which is the behavior of the program. The program +G_RE+ provides one output, which indicates the position of the release enable button on the PGS as indicated by the definition in Table 6 of !+RE pressed+!.

TABLE 5. DIM Access Programs

Program	Parameters	Description
+S_AUDIBLE_SIGNAL+	p1: AUD_ind_cntrl; I	!+Aud Signal+!
+G_RE+	p1: boolean, O	!+RE pressed+!

TABLE 6. Dictionary

Name	Interpretation
!+Aud signal+!	The current state of the audible signal.
!+RE pressed+!	$true$ iff the release enable button is currently pressed.

Specifying when to call virtual device programs and what values to write provides a description of behavior comparable to Table 4. Table 7 and Table 8 (a subset selected from p. 5-3 of [7]) describe behavior as in Table 4. Table 7 illustrates a *function description* which identifies which access program to call and the type of its input parameter. Table 8, which can be understood analogously to Table 4, describes when to call that program and what value to write (see Output value: row at bottom of table). The domain of the function in Table 8 comprises **modes** and *!+terms+!*, whose values programs in the design provide the function. In addition to the program +G_RE+ which reports the position of the release enable button, the design provides programs reporting events, e.g., of the pilot pressing and releasing the release enable button. It also provides programs for obtaining !+Rel in Progress+! and !+Weapon Class+! (see Table 11).

TABLE 7. FUNCTION DESCRIPTION: Audible signal mode

Function type:	demand
Result type:	DI.AUD_ind_cntrl
Access program:	+DI.S_AUDIBLE_SIGNAL+

Note that Table 8 and a more complete Table 6 that defined all *!+terms+!* referenced could provide a precise specification of software behavior (when the audible tone in the cockpit should sound) without referencing programs required to implement it.

TABLE 8. Audible Signal Function

MODES	EVENTS	
A/A Manrip *CCIP* *Manrip*	@T(!+RE pressed+!)	@F(!+RE pressed+!)
A/A Guns	@T(!+RE pressed+!) WHEN (!+Weapon Class+! = $RK$)	@T(!+Rel in Progress+!) OR @F(!+RE pressed+!)
Output value:	$On$	$Off$

Also note that the first column of the first row of Table 8 includes additional mode *A/A Manrip*. While this mode is not included in the corresponding cell of Table 4 from [1], the mode is defined and used elsewhere in the A-7E software requirements. And note that the last column of the first row of Table 8 does not mention !+Aud Signal+! (corresponding to the use of //BMBTON// in Table 4). The author is unaware of a rationale for these discrepancies. Nor is he aware of which, if either, is correct. This paper regards these discrepancies as exemplars of “the inconsistency inherent in having two descriptions.”

3. BACKGROUND

The approach taken by the unified specification of behavior of the A-7E OFP [14] is related to that of Heninger [11] (and applied in [1]) and to the Four-Variable Model of Parnas and Madey [17], adopting ideas and terminology from the latter and mechanisms from the former. In the unified specification, the values that a set of variables takes over time describe software behavior. Called *controlled variables*, they denote aspects of the environment that the software controls or affects. A mathematical function, usually tabular, gives the value of each variable at any point in time. In the domain of the function are *monitored variables* which denote aspects of the environment that the software monitors or measures; *terms* that simplify the specification by representing repeated or complex expressions; and *modes*, classes of system state which abstract system history [11]. While a function may specify the value of more than one variable, the value of each variable is given by exactly one function. In some instances, the function may be broken into distinct pieces that the specification presents together.

The Four-Variable Model [17] abstracts from the A-7E software requirements model [11]; that is, in place of tabular functions specifying required behavior, [17] leaves open the form that descriptions of required behavior may take. Mathematical relations on vectors of time functions for monitored, controlled, input, and output variables replace the conditions, events, modes, and tables of the A-7E requirements [11]. *REQ* in Eq. (1), a relation from all possible histories (where *possible* means allowed by environmental constraints) of the monitored variables to all possible histories of the controlled variables, describes required system behavior. *M*, the domain of *REQ*, is a set of vectors. For each monitored variable (m_i), a vector has one element, a time function ($m_i^t(t)$). The time function, which specifies the value of the monitored variable as a function of time, describes a possible history of that monitored variable. Each vector of monitored variable time functions (see Eq. (2)) describes a possible history of all of the monitored variables. *M* is the set of all possible histories of the monitored variables. *C*, the range of *REQ*, is a similar set of vectors of time functions specifying possible histories of the controlled variables, the behavior of the system. For each possible history of the monitored variables in the set *M*, *REQ* specifies one or more possible histories of the controlled variables in the set *C*. Below, this paper will use similar relations on vectors of time functions for other variables to describe other models.

$$REQ: M \rightarrow C, \qquad \textbf{Eq. (1)}$$

$$M^t = (m_1^t(t), m_2^t(t), ..., m_p^t(t)) \qquad \textbf{Eq. (2)}$$

Analogous to *M* and *C*, *I* and *O* are sets of possible histories of the system's physical device inputs and outputs, respectively. An element from a vector in *I* is a time function representing a possible history of a physical device input. Similarly, an element from a vector in *O* is a time function representing a possible history of a physical device output. The relation *IN* (Eq. (3)) specifies the behavior of the input devices. The relation *OUT* (Eq. (4)) specifies the behavior of the output devices.

$$IN: M \rightarrow I \qquad \textbf{Eq. (3)}$$

$$OUT: O \rightarrow C \qquad \textbf{Eq. (4)}$$

The A-7E requirements model of [1] and [11] can be represented approximately by Eq. (5), where *M* represents aircraft operating conditions of the OFP informally described by *!requirements terms!* of [1], *I* represents physical device inputs, *O* represents physical device outputs, and *Z* represents modes. The representation is approximate since the specifications are semi-formal. While much of the notation comprising the specification is formal, no explicit formal model guided the writing of the specification. Eq. (6) represents the mode tables of [1].

$$REQ_{A-7E}: M \times I \times Z \rightarrow O, \qquad \textbf{Eq. (5)}$$

$$Z_f: M \times I \times Z \rightarrow Z \qquad \textbf{Eq. (6)}$$

Following [13], for the unified specification, requirements and design specifications share the relation describing behavior from the Four-Variable Model described by Eq. (1). In addition, the design describes virtual devices [5], [6]. It defines virtual device inputs (I_v), such as !+RE pressed+!, and virtual device outputs (O_v), such as !+Aud Signal+!, analogously to *I* and *O*. The design specification of software behavior records the relations of Eq. (7), which describes the behavior of virtual device inputs, and Eq. (8), which describes the behavior of the virtual device outputs.

$$IN_v: M \rightarrow I_v, \qquad \textbf{Eq. (7)}$$

$$OUT_v: O_v \rightarrow C \qquad \textbf{Eq. (8)}$$

4. READING THE UNIFIED SPECIFICATION OF BEHAVIOR

The unified specification of the A-7E OFP [14] comprises functions, mostly tabular, specifying the values of 106 controlled variables, organized into the 15 sections of [7]. Following them are a specification of modes of operation, a type dictionary, a system generation parameter dictionary, and a variable dictionary.

Table 9, an example tabular function taken from the unified specification, specifies the value of the controlled variable, !+Aud signal+!, which denotes whether the aircraft's audible signal is on or off. The bracketing notation used in !+Aud signal+! indicates that the specification interprets it as either a monitored variable, controlled variable, or term defined in the variable dictionary. In the variable dictionary extract in Table 10, the *C* in the first column of the entry for !+Aud signal+! indicates that the specification interprets it as a controlled variable. Similarly, the *M* in the first column of the three following entries indicates the specification interprets them as monitored variables. The domain of the function in Table 9 comprises **modes** and *!+terms+!* interpreted as monitored variables.Other functions may interpret some *!+terms+!* as terms, which simplify the specification by representing repeated or complex expressions.

TABLE 9. Audible Signal Mode

MODES	EVENTS	
A/A Manrip *CCIP* *Manrip*	@T(!+RE pressed+!)	@F(!+RE pressed+!)
A/A Guns	@T(!+RE pressed+!) WHEN (!+Weapon Class+! = $RK$)	@T(!+Rel in Progress+!) OR @F(!+RE pressed+!)
!+Aud signal+!	$On$	$Off$

The definition of a monitored or controlled variable in the variable dictionary includes the variable type and its *interpretation*, which describes how the value of the variable relates to the aspect of the environment that the variable denotes. The definition of a term also includes its type. The interpretation of a term may contain either the expression that the term represents or an informal description of its value.

Rules describing when the audible signal beeps (sounds intermittently) are not included in this limited subset of the complete function; hence, the value $Intermittent$ is not used in Table 9. More examples of tabular functions specifying the values of controlled variables are in [14]. References [2]

TABLE 10. Variable Dictionary

	Names	Type	Interpretation
C	!+Aud signal+!	AUD_ind_cntrl	The current state of the audible signal.
M	!+RE pressed+!	boolean	$true$ iff the release enable button is currently pressed.
M	!+Rel in Progress+!	boolean	$true$ iff a release pulse is currently being issued to the active weapon station(s).
M	!+Weapon Class+!	weap_class	The class of the weapon loaded on the currently active weapon station(s). If no weapon station is active, then GN.

and [10] describe formal semantics of such functions, which were created before the semantics were available.

5. CREATING THE UNIFIED SPECIFICATION OF BEHAVIOR

Once we understand how the A-7E design organizes its description of the behavior of the OFP, extracting that description to create the unified specification (see Table 9 and Table 10) is conceptually simple. Fig. 1 provides a graphical overview of the module structure of the A-7E OFP design. Each of the boxes in the figure represents an information hiding module. [16] System details that are likely to change independently are assigned to separate modules. In the figure, arrows point from a module to its submodules, e.g., Behavior Hiding is comprised of the Function Driver and Shared Services modules.

Blacked-out modules, which are concerned with mechanisms for accomplishing behavior, are not relevant to this discussion of behavior. The Software Decision module concerns decisions made by software designers based upon "mathematical theorems, physical facts, and programming considerations" [6]. The Extended Computer module concerns characteristics of the A-7E computer.

The Behavior Hiding module is concerned with descriptions of behavior recorded in requirements. It's comprised of the Function Driver and Shared Services modules. "The Function Driver Module consists of a set of modules called Function Drivers; each Function Driver is the sole controller of a set of closely related outputs." [6] These outputs, e.g., !+Aud signal+!, constitute the behavior of the OFP, i.e., its impact on the system environment. Functions that capture the rules determining these output values specify that behavior. The Shared Services module concerns aspects of behavior common to several Function Drivers.

The Hardware Hiding module comprises two modules, one of which (Extended Computer) is irrelevant to the discussion of software behavior. The other is the Device Interface module (DIM), which "provides virtual devices to be used by the rest of the software." [6] The DIM defines the outputs, e.g., !+Aud signal+!, that describe OFP behavior. It defines also many of the *!+terms+!* that comprise the domains of the functions that specify the values of outputs. The Shared Services module defines other *!+terms+!* in the domains of the functions.

The functions in the unified specification of behavior and those in the Function Driver Module (see Table 7 and Table 8) share the same domains. They use the same bracketed variable names in the same way to specify the events and conditions in the rules determining the values of the outputs. Functions in the Function Driver Module specify also the programs to be called to set those outputs and the values of parameters to be passed to those programs (see, e.g., Table 7, Table 8). In contrast, functions in the unified specification specify the values of *!+terms+!* identified as controlled variables (e.g., Table 9).

Adapting design documents for the A-7E OFP [7], [8], [15] produces the unified specification. The unified specification reinterprets *!+terms+!* that the design documents define to describe the behavior of programs. It interprets *!+terms+!* that describe parameters of programs that write to virtual device outputs as *controlled* variables, quantities and qualities in the system environment that the OFP controls. For example, the program +S_AUDIBLE_SIGNAL+ in Table 5 writes to virtual device output !+Aud Signal+!. The unified specification interprets !+Aud Signal+! as a controlled variable. It interprets *!+terms+!* that describe output parameters of programs that read virtual device inputs as *monitored* variables, quantities and qualities in the system environment that the OFP monitors. For example, the program +G_RE+ in Table 5 reads virtual device input !+RE pressed+!. The unified specification interprets !+RE pressed+! as a monitored variable.

Adapting tabular functions from the Function Driver [7] to produce corresponding functions for the unified specification involves removing *function descriptions*, e.g., Table 7. It also involves replacing the contents of the left hand column of the bottom row of the table (e.g., Output value: in Table 8) with

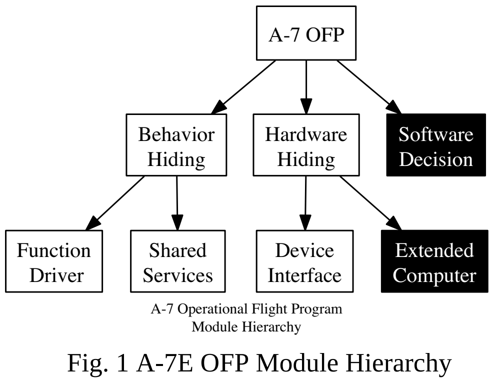

Fig. 1 A-7E OFP Module Hierarchy

the name of what is now the controlled variable, e.g., !+Aud Signal+!. The adapted functions specify the values of controlled variables, the behavior of the OFP.

Usually, the module that provides the program that the function driver calls to set a virtual device output defines a *!+term+!* describing the output. The variable dictionary of the unified specification collects such *!+term+!* definitions, labeling them *controlled variables* (see, e.g., row 1 of Table 10). When the module does not provide suitable *!+terms+!*, the author of the unified specification defines them and includes them in the variable dictionary, labeling them *controlled variables*.

Modules also define *!+terms+!* in the domain of each function and provide programs for obtaining their values. The unified specification interprets these *!+terms+!* as monitored variables if they denote aspects of the environment of the OFP and as terms otherwise. In either case, the unified specification collects them and their definitions from the defining module into the variable dictionary. In a small number of cases, function drivers reference *!requirements terms!* defined in the A-7E requirements [1]. The unified specification interprets them as monitored variables or terms and collects them in the variable dictionary. In instances that design documentation (or requirements, in the case of *!requirements terms!*) does not provide suitable definitions, the author of the unified specification defines them and includes them in the variable dictionary, labeling them *monitored variables* or *terms*, as appropriate. Of the 106 controlled variables in the unified specification, the A-7E design defines 81; the author defined the remaining 25. The variable dictionary contains 307 monitored variables of which the author defined 26. It contains 22 terms, of which the author defined 14. The variable dictionary contains 8 *!requirements terms!* from the A-7E requirements [1], four of which the author defined. See [14] for details.

!!Local terms!! in the local dictionary of each function driver are, for the most part, copied into the local dictionary of the corresponding function in the unified specification. In some instances, the definition of a *!!local term!!* includes a call to a program performing some calculation. In the unified specification, the calculation replaces the program call in the definition. For example, the local dictionary of the function driver *Set HUD flight director azimuth position* in [7] defines !!ltd brg ac ftpt!! as the results returned by the program call +SU.LIMIT_2+(!!steering error to ftpt!!, 0.5). The function in the unified specification corresponding to this function driver specifies the value of the controlled variable !+FLTDIR azimuth+!. Its local dictionary defines !!ltd brg ac ftpt!! as (!!steering error to ftpt!! / ABS(!!steering error to ftpt!!)) x MIN(!!steering error to ftpt!!, 0.5), the calculation that the called program performs.

Since the Mode Determination Module, a submodule of Shared Services, hides how to determine the current modes of the OFP [6], the rules specifying initial modes and mode transitions rules don't appear in the module's specification. While the A-7E OFP requirements [1] describe these rules, the conditions used to construct transition events reference *!requirements terms!* defined in the requirements and inputs from physical devices, rather than definitions in the unified specifications's variable dictionary. Consequently, specifying the modes of operations for the unified specification requires translating conditions used in [1] to conditions based on the variable dictionary of the unified specification [14].

The mode transition tables in the requirements use 70 terms to describe mode transitions; 43 are *!requirements term!* expressions (some involving more than one *!requirements term!*) and 27 are */input/* expressions. Adapting the mode tables for the unified specification required finding 70 corresponding *!+terms+!*. The author defined or redefined 16 *!+terms+!* for *!requirements terms!* that had no suitable design equivalent.

6. INTERPRETING THE UNIFIED SPECIFICATION

This section interprets the A-7E unified specification as a specification of required behavior and as a specification of behavior in design, respectively.

A Specification of Required Behavior

In contrast to the Four-Variable Model, to the model of [13], and to the A-7E requirements discussed above, in the unified specification of the A-7E OFP, the relation REQ_U of Eq. (9) specifies the behavior of the OFP, where I_v (the virtual input variables) and O_v (the virtual output variables) are interpreted as monitored and controlled variables, respectively. Note that [13], Eq. (7), and Eq. (8) distinguish virtual inputs and outputs from monitored and controlled variables.

Z represents the modes in Eq. (9) and Eq. (10). The relation Z_U in Eq. (10) represents mode transition tables.

$$REQ_U\text{: } I_v \; X \; Z \rightarrow O_v \qquad \textbf{Eq. (9)}$$

$$Z_U\text{: } I_v \; X \; Z \rightarrow Z, \qquad \textbf{Eq. (10)}$$

Inspection of the variable dictionary in the unified specification finds many entries that clearly denote quantities and qualities in the environment of the OFP, suggesting that it is not unreasonable to interpret them as monitored and controlled variables. For example, the interpretation of the controlled variable !+Aud signal+! is *The current state of the audible signal*. The interpretation of the monitored variable !+az miss dist+! is *The distance along the ground between the target and the ground-projected line from the aircraft to the computed impact point.* The interpretations of some variables assume the reader is familiar with concepts and terms described in [1]. For example, understanding the interpretation of !+boresight azimuth+! requires the reader know what the *Ya axis* and the *Xa-Ya plane* are. Instead of just describing how a monitored or controlled variable's value relates to some aspect of the OFP environment, some interpretations also describe how to use the variable, an unwelcome redundancy with the functions that reference the variable. !+E coarse scale+! provides an example of such an interpretation: *Scale factor per pulse used for velocity calculation for the Xp axis when the velocities are being measured by the coarse scale.*

In the unified specification, functions specify the values that controlled variables must assume as relevant quantities and qualities in the environment of the OFP change over time. Monitored variables, terms, and mode transition tables describe that changing environment. This suggests that it is also not unreasonable to interpret the unified specification as a specification of required behavior of the OFP.

The unified specification of the behavior of the A-7E OFP, like the specifications of behavior in the requirements [1] and the design [7], is *semi-formal*. While much of the notation comprising the specification is formal, no formal semantics underlie it. Though formal semantics for such specifications exist (e.g., [2], [10]), it would require some work to make this specification adhere to one of them. In addition, many aspects of the specification are informally captured.

A Specification of Behavior for Design

Since the unified specification of behavior of the A-7E was adapted from design specifications of the A-7E OFP, it's reasonable to think it can serve design needs. The unified specification of behavior can be incorporated without modification (with some exceptions discussed below) into a design that adheres to the model described in [18] and exemplified by [6]. Such a design consists of a number of information hiding modules [16], some of which provide programs intended to be used by the programs of other modules ([8], [15], Table 11) and some of which comprise programs, called *function drivers*, that use programs in other modules ([7], Table 9). The function drivers, which specify the values of the controlled variables, use other programs to set the values of the controlled variables and to obtain the values of the monitored variables, terms, and modes that determine what the values of the controlled variables should be. The function drivers are incorporated into the Function Driver Module. The organization of the controlled variable functions (function drivers) into 15 sections reflects an information hiding decomposition of the Function Driver Module. Each of the fifteen sections represents a submodule of the Function Driver Module. [6]

The Mode Determination Module, a submodule of Shared Services, hides the rules specifying the transitions among the system modes. Consequently, its specification can be thought of as part of the module's internal design, specifying how to implement the module's functions, as opposed to specifying their black box behavior.

The definitions in the variable dictionary describe the behavior of programs in the DIM [15] that implement virtual devices and of certain programs in the Shared Services Module [8]. Associating a controlled variable with the input parameter to a program indicates that the effect of calling the program with the parameter set to a particular value is to affect the environmental aspect denoted by the controlled variable in the appropriate way. For example, +S_AUDIBLE_SIGNAL+, a program on the interface of the Audible Signal device interface module (Table 11 illustrates documentation of the program adapted from [15]), has one input parameter of type AUD_ind_cntrl. The type indicates the parameter can have values On, Off, and $Intermittent$. The controlled variable !+Aud signal+!, which denotes the "current state of the audible signal," describes the effect of setting the input parameter. Thus, the effect of calling the program +S_AUDIBLE_SIGNAL+ with input parameter p1 set to the value of the controlled variable !+Aud signal+! is to cause the audible signal either to be silent, to be on steady, or to beep (as described by !+Aud signal+!). It is the responsibility of the implementation of the function driver that specifies the value of the controlled variable !+Aud signal+! to call +S_AUDIBLE_SIGNAL+ and pass it the parameter value specified by the function.

TABLE 11. Access Program Interfaces

Program	Parameters	Description
+S_AUDIBLE_ SIGNAL+	p1: AUD_ind_cntrl; I	!+Aud Signal+!
+G_WEAPON_ RELEASE_CLASS+	p1: weap_class; O	!+Weapon Class+!
+G_RE+	p1: boolean; O	!+RE pressed+!
+G_REL_IN_ PROGRESS	p1: boolean; O	!+Rel in progress+!

Similarly, associating a monitored variable with the output parameter of a program indicates that on return from a call to the program, the output parameter's value will reflect appropriately the environmental aspect denoted by the monitored variable. For example, +G_WEAPON_RELEASE_CLASS+, a program on the interface of the Weapon Characteristic Submodule of the

Device Interface Module (Table 11), has one output parameter of type weap_class. Defined in Table 10, the monitored variable !+Weapon Class+!, which is the "class of the weapon loaded on the currently active weapon station(s)" describes the value returned by the parameter. On return from a call to +G_WEAPON_RELEASE_CLASS+, the parameter has the value, for instance, RK if and only if rockets are loaded on the currently active weapon station(s). The implementation of the function driver that specifies the controlled variable !+Aud signal+! calls the program to determine whether rockets are loaded when deciding to sound the audible signal when it detects that the pilot has pressed the release enable button, while in mode *A/A Guns*. The function depends on the DIM to signal occurrence of the event @T(!+RE pressed+!), i.e., !+RE pressed+! going from $False$ to $True$.

Each function driver implementation uses the appropriate program to set each controlled variable whose value it determines. Similarly, the function driver implementation uses the appropriate program to obtain the value of each monitored variable, term, and mode that it references. The DIM (see [6]) provides programs that manifest the values of controlled variables. The DIM and other modules provide programs to obtain the values of monitored variables, terms, and modes. In the case of events, these modules provide mechanisms that signal when the value of a variable of interest changes.

Note that starting with the unified specification, completing the design requires making and recording design decisions regarding what programs each function driver must depend upon to determine the values of relevant monitored variables and terms and when they change and to set the values of controlled variables. The model of the unified specification [13] assumes there's a simple, straightforward relationship between variables and the programs called to set them or to determine their values. Ideally, when a function in the unified specification specifies the value of a particular controlled variable, the design should provide one program to call which accepts that value as input and will manifest the value appropriately in the environment (e.g., sound the audible alarm). Analogously, when a rule in a function references a particular monitored variable or term, the design should provide one program to call that will provide the value of the referenced monitored variable or term.

Sometimes this simple, straightforward relationship between variables and the programs does not exist in the A-7E design. The Doppler Radar Set (DRS) provides two parameterless programs to turn it on and off. The value of the boolean controlled variable !+DRS on+! determines which program to call. Weapon Release System also provides a parameterless program to manifest a controlled variable value in the environment. Toggling the value of boolean controlled variable !+prepare weapon+! causes the design to call the corresponding program. To take advantage of the physical device's ability to control certain symbols together, the Head-Up Display Location-Indicator provides one program that will accept several controlled variables at once. The initialization of several physical devices does not fit well into this model. The Projected Map Display Set provides a program that returns a boolean indicating whether the PMDS can display a given point on the Earth. A subsequent call to another program will either display that position or report an error. More details are available in [14].

7. DISCUSSION

This paper understands *software behavior* to refer to the impact that software has on the environment of the system that incorporates it. The paper reports an effort to produce one description of the software behavior of a real system that can serve both requirements and design as described by [13]. It presents the A-7E unified specification as a proof of concept, more than as an exemplar. The author adapted this specification from *design* documentation. The specification reinterprets *!+terms+!* defined to describe the behavior of programs providing virtual devices as monitored and controlled variables denoting quantities and qualities in the system environment. The author adapted tabular functions describing when to call programs implementing virtual output devices and to what values to set their parameters. The adapted functions specify the values of virtual outputs that the unified specification interprets as controlled variables. Of the 106 controlled variables, a handful were unable to fit the model of [13] well.

Reference [12] applies the idea of a unified specification, without using those words, to model-driven development. It refers to variables as "attributes," which are organized into four overlapping models. The Sage *behavior model* records the information in the unified specification. The *environmental model* associates environmental attributes, which includes what this paper calls *monitored variables* and *controlled variables*, with objects that they characterize in the environment of the system. The *design model* organizes the information in the behavior model into *design classes*, analogous to the information hiding modules discussed earlier. The *run-time model* organizes the behavior model into loosely-coupled, location-transparent reactive agents. Both design model and run-time model distinguish

- Attributes on a class's or agent's public interface.

- Attributes expected to be on another class's or agent's public interface.
- Attributes that are local to a class or agent.

The Sage development environment generates agents in the SOL [2] language. When translated to the language of the Salsa property checker [4], Salsa checks the consistency and completeness [9] of the SOL agents. When compiled, the SOL agents execute in the execution environment that SINS middleware [3] provides.

Success of the unified specification approach in reducing the time and effort needed to develop, evolve, and sustain a software system requires understanding the system's needs and supporting infrastructure and technology. It also requires understanding how they will evolve during the system's development and operational life. To the extent that this understanding is flawed, benefits resulting from expected reductions in effort and error may not be realized.

Acknowledgements: The author is grateful to S. Faulk, D. Weiss, and an anonymous review for thoughtful comments that led to an improved paper. Comments from P. Hurley led to an improved presentation of the paper. The author acknowledges the support of (ASD)R&E/S&T/IS/Software Technology.

8. REFERENCES

[1] T.A. Alspaugh, S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, J.E. Shore, **Software Requirements for the A-7E Aircraft**, NRL Final Report 5530, 1992.

[2] R. Bharadwaj. "SOL: A Verfiable Synchronous Language for Reactive Systems", **Proceedings, Synchronous Languages, Applications, and Programming**, (SLAP'02), 2002.

[3] R. Bharadwaj. "Secure Middleware for Situation-Aware Naval C2 and Combat Systems", **Proc., Ninth International Workshop on Future Trends of Distributed Computing Systems**, May 2003.

[4] R. Bharadwaj and S. Simms. "Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking", **Lecture Notes in Computer Science**, Volume 1785, January 2000.

[5] K. Britton, R. Parker, D. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules", **Proc., 5th International Conf. on Software Eng.**, March 1981, pp.195-204.

[6] K.H. Britton, D.L. Parnas, **A-7E Software Module Guide**, NRL Memorandum Report 4702, 1981.

[7] P.C. Clements, **Function Specifications for the A-7E Function Driver Module**, NRL Memorandum Report 4658, November 27, 1981.

[8] P.C. Clements, **Abstract Interface Specifications for the A-7E Shared Services Module**, NRL Memorandum Report 4863, 1982.

[9] C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated Consistency of Requirements Specifications", ***ACM Trans. Software Eng. and Methodology***, Vol. 5, No. 3, July 1996.

[10] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications", **IEEE Trans. on Software Engineering,** IEEE Computer Society, November 1998, pp. 927-948

[11] K.L. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and their Application", **IEEE Trans. on Software Engineering**, IEEE Computer Society, January 1980, pp. 2-13.

[12] J. Kirby, "Model-Driven Agile Development of Reactive Multi-Agent Systems", **Proc., 30th International Computer Software and Applications Conference (COMPSAC 2006)**, September 2006, Chicago, IL.

[13] J. Kirby, "Rewriting Requirements for Design". **Proceedings, IASTED International Conference on Software Engineering and Applications (SEA) 2002**, November 4-6, 2002, Cambridge, MA, USA.

[14] J. Kirby, **A Unified Specification of Behavior for Requirements and Design**, NRL Memorandum Report NRL/MR/5540-07-9094, December 10, 2007.

[15] A. Parker, K.H. Britton, D. Parnas, J. Shore, **Abstract Interface Specifications for the A-7E Device Interface Module**, NRL Memorandum Report 4385, November 20, 1980.

[16] D.L., Parnas, "On the Criteria to be Used in Decomposing Systems in Modules", **Comm. of the ACM**, 15(12), 1972, 1053-1058.

[17] D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering", **Science of Computer Programming**, Elsevier, October 1995, pp 41-61.

[18] D. Parnas, P. Clements, and D. Weiss. "The Modular Structure of Complex Systems", **IEEE Trans. on Software Engineering**, IEEE Computer Society, March 1985.

[19] D.M. Weiss, Introduction to "The Modular Structure of Complex Systems", **Software Fundamentals, Collected Papers of David L. Parnas**, ed. M.D. Hoffman and D.M. Weiss, Addison-Wesley 2001.