Journal of Universal Computer Science, vol. 6, no. 7 (2000), 650-678
submitted: 19/1/00, accepted: 20/4/00, appeared: 28/7/00

Applying the SCR Requirements Method to the Light Control Case Study

Constance Heitmeyer
Naval Research Laboratory (Code 5546)
Washington, DC 20375
heitmeyer@itd.nrl.navy.mil

Ramesh Bharadwaj
Naval Research Laboratory (Code 5546)
Washington, DC 20375
ramesh@itd.nrl.navy.mil

Abstract: To date, the SCR (Software Cost Reduction) requirements method has been used in industrial environments to specify the requirements of many practical systems, including control systems for nuclear power plants and avionics systems. This paper describes the use of the SCR method to specify the requirements of the Light Control System (LCS), the subject of a case study at the Dagstuhl Seminar on Requirements Capture, Documentation, and Validation in June 1999. It introduces a systematic process for constructing the LCS requirements specification, presents the specification of the LCS in the SCR tabular notation, discusses the tools that we applied to the LCS specification, and concludes with a discussion of a number of issues that arose in developing the specification.

Keywords: software, software engineering, requirements, specifications, tools and techniques, formal verification.

Categories: D.2.1, D.3.1

1 Introduction

The SCR (Software Cost Reduction) requirements method is a formal method based on tables for the specification and analysis of the required behavior of complex software systems. Originally developed by NRL researchers to document the requirements of the operational flight program of the US Navy's A-7 aircraft [2, 12, 13], SCR has also been applied by a number of organizations in industry (e.g., Grumman, Bell Laboratories, Ontario Hydro, and Lockheed) to a wide range of practical systems, including avionics systems, telephone networks, and safety-critical components of nuclear power plants. For example, in 1994, in the largest application of SCR to date, Lockheed used SCR to specify the requirements of the C-130J flight program [8], a program containing more than 250,000 lines of Ada code.

To provide tool support for the SCR method, our group at the Naval Research Laboratory has developed an integrated suite of tools called the SCR* toolset [10]. The toolset includes a *specification editor* for creating and modifying a requirements specification and several analysis tools, including a *consistency checker* for checking the specification for defects such as type errors, missing cases, and unwanted nondeterminism [11]; a *dependency graph browser* for displaying the dependencies among the variables in the specification; and a *simulator* for symbolically executing the system based on the specification. Currently, more than 100 academic, industrial, and government organizations in the US, Canada, and several other countries are experimenting with the SCR* toolset.

The utility of the SCR* toolset has been evaluated in four pilot projects. In one, NASA researchers used the toolset's consistency checker to detect several errors in the requirements specification of the International Space Station [7]. In a second project, Rockwell engineers used the SCR* toolset to detect 28 errors, many of them serious, in the requirements specification of a flight guidance system [17]. In a third project, our group at NRL used the SCR* toolset to expose several errors, including a safety violation, in a contractor-produced specification of a US military system [9]. In a fourth project, our group used the SCR* toolset to specify the requirements of a cryptographic device (CD), to verify that the CD specification satisfies seven security properties, and to demonstrate that the specification violates an eighth property [16]. To be useful in practice, the benefits of using a method should be sufficient to warrant the cost in human effort of applying the method. In the latter two projects, the potential cost-effectiveness of the SCR method and the SCR* toolset was demonstrated: in each case, specifying and analyzing a moderately complex system required only one person-month of effort.

This paper describes the use of the SCR method to specify the requirements of the Light Control System (LCS), the subject of a case study at the Dagstuhl Seminar on Requirements Capture, Documentation, and Validation in June 1999. We present the LCS specification in the SCR tabular notation to allow a comparison between the SCR requirements method and other alternative requirements methods. To develop the LCS requirements specification, we follow a four-step process. This process provides a systematic approach to developing and organizing a requirements specification of a nontrivial system. The product of this process is a "build-to specification," a requirements specification that describes the set of all acceptable system implementations. Thus, the SCR method would capture the results of the requirements elicitation phase. In developing the LCS requirements specification with SCR, one would construct the specification with the specification editor and apply the analysis tools of the SCR* toolset to validate and verify that the specification satisfies desired properties.

The remainder of this paper is organized as follows. Section 2 describes the four-step process we propose for specifying the system requirements using SCR. It also reviews the constructs and notation used in SCR specifications. Section 3 uses the process presented in Section 2 to specify the requirements of a significant component of the LCS, namely, the component that controls the light setting in offices. In specifying the LCS requirements, we had two options. We could attempt to construct a complete specification of the required behavior of the LCS presented in [1]. Alternatively, we could focus our attention on a critical piece of the required LCS behavior. We chose the latter option because we believe that it is better to demonstrate our method on a carefully chosen component, to obtain feedback about our specification of that component from system experts, and then to revise and extend our specification based on the feedback. Section 3 also shows how the SCR method can be used to handle hardware malfunctions and to specify timing constraints. Section 4 describes the results of applying the SCR* toolset to the LCS specification. Section 5 discusses issues that arose when we applied our requirements method to the LCS, and Section 6 compares our approach to requirements specification with other similar approaches. Finally, Section 7 presents some conclusions and future work.

2 A Process for Specifying Requirements

By *specification*, we mean a description of the *required behavior* of an entire system, subsystem, or component. A specification should describe *what* is to be built, omitting details of *how* this will be achieved. A system or component that satisfies the specification can be implemented in hardware, software, or a combination of both. An important goal is to avoid both overspecification and underspecification. Thus a specification must describe the required black-box behavior of *every* acceptable implementation and must exclude all unacceptable implementations.

Figure 1 is the basis for a four-step process for constructing a requirements specification. The first step creates the *System Requirements Specification* (SRS), which describes the required external behavior of the system in terms of monitored and controlled quantities in the system environment. The remaining steps refine and extend the SRS. The second step creates the *System Design Specification* (SDS), which identifies the input and output devices (e.g., sensors and actuators) of the system. The third step creates the *Software Requirements Specification* (SoRS), which refines the SRS by adding modules which use values read from input devices to calculate values of the monitored quantities and which use the computed values of controlled quantities to drive output devices. The fourth step extends the SRS by adding behavior to handle hardware malfunctions, e.g., sensor failures.

By applying the information hiding principle [18] to the requirements specification produced by this process, parts of the specification that are unlikely to change together are assigned to different modules. In applying information hiding to the specification, all of the ways in which the requirements are likely to change are identified, and the required system behavior is decomposed into modules so that exactly one module is associated with a single change. The goal is to organize the requirements specification such that each change requires a change to only a single module. How this is achieved is described below.

2.1 System Requirements Specification

To construct an SRS using SCR, environmental quantities relevant to the system behavior are identified, and each quantity is represented by a mathematical variable. The environmental quantities consist of both *controlled quantities* – quantities in the environment that the system controls – and *monitored quantities* – quantities in the environment that can influence system behavior. In Figure 1, M represents the monitored quantities and C represents the controlled quantities.

The desired *system behavior* is documented in the SRS by describing two relations, NAT and REQ, on the monitored and controlled quantities; these

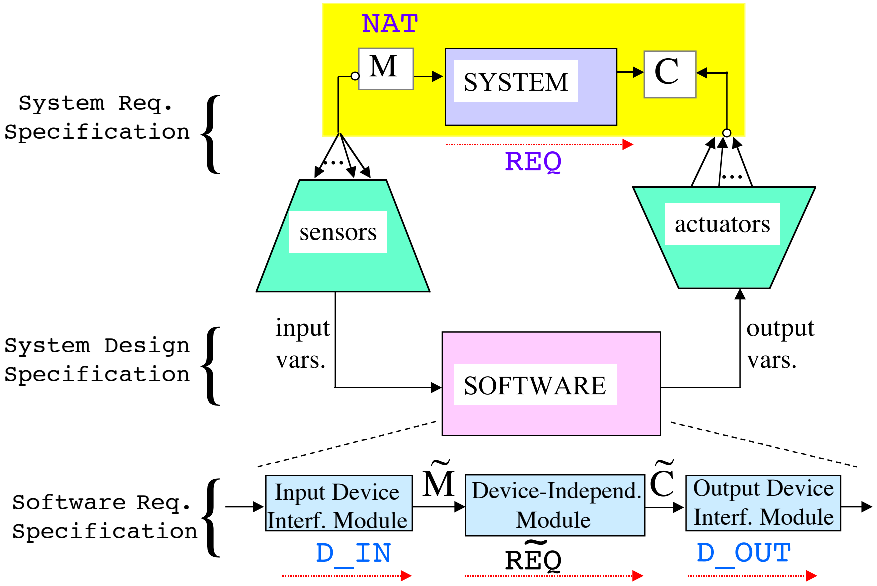

Figure 1: Relationship between the SRS, the SDS, and the SoRS.

relations are part of the Parnas Four Variable Model [19]. The relation NAT describes the constraints imposed on the environmental quantities by physical laws and the system environment. REQ describes the relation between the monitored and controlled variables that the system must enforce to produce the required system behavior. In developing the SRS, we initially specify REQ in terms of the *ideal* behavior of the system; that is, we assume that the system can obtain perfect values of the monitored quantities and compute perfect values of the controlled quantities. Later, for each controlled variable, we specify timing constraints (and possibly tolerances).

2.2 System Design Specification

The SDS identifies and documents the characteristics of the resources that are available to estimate values of the monitored quantities and to set values of the controlled quantities. These values are usually read from or written to hardware devices, such as sensors and actuators. (Although we assume below that the resources are hardware devices, what follows also applies when the resources are external computers or other software modules.) The values in the system's hardware/software interfaces are denoted by mathematical variables. These variables are partitioned into *input variables* – values read by input devices – and *output variables* – values written to output devices. The product of this step is a description of the input and output devices and of the relationship between the input and output variables and the monitored and controlled variables.

2.3 Software Requirements Specification

The SRS and the SDS are the foundation for the SoRS, which describes how the input variables are to be used to estimate values of the monitored variables and how estimates of the controlled variables are to be used to control the output devices using the output variables. Figure 1 shows the relationship between the SRS, the SDS, and the SoRS.[1] Figure 1 also shows the decomposition of the SoRS into three modules: two *device-dependent* modules called the *input device interface module* and the *output device interface module*, and a single *device-independent* module called the *function driver* module. This organization was influenced by the module structure of the A-7 flight software [20]. In Figure 1, relation D_IN specifies how estimates of the monitored variables, represented by $\widetilde{\mathrm{M}}$, are computed in terms of the input variables, and relation D_OUT specifies how estimates of the controlled variables, represented by $\widetilde{\mathrm{C}}$, are used to compute the values of the output variables. The outputs of the input device interface module, i.e., the estimated values of the monitored variables, form the inputs to the function driver module. The function driver module uses these estimates to compute estimates of the controlled variables.

The required behavior of the function driver module is already defined by the REQ relation, specified as part of the SRS during the first step of our process. What remains is to document the required behavior of the device-dependent modules, i.e., D_IN and D_OUT. To satisfy the information hiding principle, the input device interface module only uses values of input variables to estimate values of the monitored variables, and the output device interface module only uses values of controlled variables to compute the values of the output variables.[2] The benefit of this approach is that it makes the specification easy to change. For example, to introduce a new input or output device or to modify or add a system function, usually only a small part of a single module will change.

In Figure 1, the relation $\widetilde{\mathrm{REQ}}$ specifies the relation between estimates of the monitored quantities $\widetilde{\mathrm{M}}$ and estimates of the controlled quantities $\widetilde{\mathrm{C}}$. In most cases, $\widetilde{\mathrm{REQ}}$ will extend REQ because $\widetilde{\mathrm{REQ}}$ not only describes the ideal behavior captured by REQ but also describes externally visible behavior that is not part of the ideal behavior. Because REQ is based on perfect knowledge of the monitored quantities and perfect computations of the controlled quantities, REQ does not describe how the system responds to hardware malfunctions. In practical systems, hardware devices, such as sensors, will fail, and the system will need to provide external notification of such failures. $\widetilde{\mathrm{REQ}}$ extends the required behavior described by REQ by describing how notification of hardware malfunctions is presented to the system users.

2.4 The SCR Notation

To specify the required system behavior in a practical and efficient manner, the SCR method uses terms and mode classes. A *term* is an auxiliary variable that helps keep the specification concise. A *mode class* is a special case of a term, whose values are modes. Each mode defines an equivalence class of system states useful in specifying the required system behavior. In SCR specifications, we often use the following prefixes in variable names: "m" to indicate monitored variables, "t" for terms, "mc" for mode classes, "c" for controlled variables, "i" for input variables, and "o" for output variables.

[1] Although the structure of the diagram in Figure 1 resembles a commuting diagram, it does not actually commute.

[2] In some cases, this assumption is too strong. For an example, see Section 3.3.2.

Conditions and events are important constructs in SCR specifications. A *condition* is a predicate defined on one or more state variables (a *state variable* is a monitored or controlled variable, a mode class, or a term). An *event* occurs when a state variable changes value. The notation "`@T(c) WHEN d`" denotes a *conditioned event*, defined as

$$\texttt{@T(c) WHEN d} \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d, \tag{1}$$

where the unprimed conditions c and d are evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. Informally, this expression denotes the event "predicate c becomes *true* in the new state when predicate d holds in the old state". The notation "`@F(c)`" denotes the event `@T(NOT c)` and "`@C(x)`" denotes the event "variable `x` has changed value". The notation $\texttt{DUR}(c)$ indicates the length of time that condition c has been true. For example, if c becomes true when the system time, represented by `time`, is 10, then if c remains true at system time 15, the condition $\texttt{DUR}(c) > 8$ is false at $\texttt{time} = 15$, since $15 - 10 = 5 \not> 8$; if c still remains true at system time 25, the condition $\texttt{DUR}(c) > 8$ is true at $\texttt{time} = 25$, since $25 - 10 = 15 > 8$.

To specify the REQ relation, SCR specifications use a set of tables. Each table defines the value of a dependent variable (a term, mode class, or controlled variable) as a function.[3] A table may be either a condition table or an event table. Typically, a condition table describes the value of a controlled variable or term as a function of a mode class and a *condition*, whereas an event table describes the value of a controlled variable or term as a function of a mode class and an *event*. A mode transition table is a special case of an event table. Although many SCR tables use modes to define the value of a variable, some SCR tables omit modes. In Section 3.3, we also use tables to specify the functions that make up D_IN and D_OUT.

Figures 2 and 4 contain examples of a condition table and an event table. (Like many tables in Section 3, the rightmost column of Figures 2 and 4, labeled Trac. for "traceability," identifies the associated requirement in the LCS description [1].) Figure 2 contains a moded condition table that defines the value of the term `tRemLL` as a function of the mode class `mcStatus`, the terms `tCurrentLSVal` and `tOverride`, and the monitored variable `mIndoorLL`. (Section 3 describes the meaning of these variables.) The table's first two rows state that `tRemLL` is zero if `mcStatus` is `unoccupied`, or if `mcStatus` is `occupied` and $\texttt{mIndoorLL} > \texttt{tCurrentLSVal}$, and that `tRemLL` has the value $\texttt{tCurrentLSVal} - \texttt{mIndoorLL}$ if `mcStatus` is `occupied` and $\texttt{mIndoorLL} \leq \texttt{tCurrentLSVal}$. The function defined by Figure 2 is shown in Figure 3.

Figure 4 contains a modeless event table that defines the new value of the controlled variable `cWallLights` as a function of `mcStatus`, the monitored variables `mWallLights` and `mFMOverride`, and the old value of `cWallLights`. The table states that `cWallLights` changes to on if `mWallLights` becomes on when `cWallLights` is off or if `mcStatus` changes to `occupied`; that `cWallLights` changes to off if `mWallLights` changes to off when `cWallLights` is on, if `mcStatus` changes to `unoccupied`, or if `FMOverride` becomes true when `mcStatus` is not `occupied`; and that otherwise the value of `cWallLights` does not change. By applying (1), the event table in Figure 4 can be translated into a function, part of whose definition is given in Figure 5.

[3] Some tables in Section 3 describe two functions. See, e.g., Figure 14.

Mode Class = mcStatus			Trac.
Mode	Condition		
unoccupied	true	false	FM3
occupied	mIndoorLL > tCurrentLSVal	mIndoorLL ≤ tCurrentLSVal	FM1
temp_empty	mIndoorLL > tCurrentLSVal OR tOverride	mIndoorLL ≤ tCurrentLSVal AND NOT tOverride	FM1, FM6
tRemLL	0	tCurrentLSVal − mIndoorLL	FM1

Figure 2: Condition table defining the value of term tRemLL

$$\texttt{tRemLL} = \begin{cases} 0 & \text{if } (\texttt{mcStatus} = \texttt{unoccupied}) \lor (\texttt{mcStatus} = \texttt{occupied} \land \\ & \quad \texttt{mIndoorLL} > \texttt{tCurrentLSVal}) \lor \\ & \quad (\texttt{mcStatus} = \texttt{temp\_empty} \land \\ & \quad\quad (\texttt{mIndoorLL} > \texttt{tCurrentLSVal} \lor \texttt{tOverride})) \\ \texttt{tCurrentLSVal} & \text{if } (\texttt{mcStatus} = \texttt{occupied} \land \texttt{mIndoorLL} \leq \texttt{tCurrentLSVal}) \\ \quad - \texttt{mIndoorLL} & \quad \lor (\texttt{mcStatus} = \texttt{temp\_empty} \land \\ & \quad\quad \texttt{mIndoorLL} > \texttt{tCurrentLSVal} \land \neg\texttt{tOverride}) \end{cases}$$

Figure 3: Function defined by condition table for term tRemLL

Event	cWallLights′	Trac.
@T(mWallLights = on) WHEN cWallLights = off OR @T(mcStatus = occupied)	on	- U1
@T(mWallLights = off) WHEN cWallLights = on OR @T(mcStatus = unoccupied) OR @T(mFMOverride) WHEN mcStatus ≠ occupied	off	- FM3 FM6

Figure 4: Event table for controlled variable cWallLights

$$\texttt{cWallLights}' = \begin{cases} \texttt{on} & \text{if } (\texttt{mWallLights}' = \texttt{on} \land \texttt{mWallLights} \neq \texttt{on} \land \\ & \quad \texttt{cWallLights} = \texttt{off}) \lor \ldots \\ \texttt{off} & \text{if } (\texttt{mWallLights}' = \texttt{off} \land \texttt{mWallLights} \neq \texttt{off} \land \\ & \quad \texttt{cWallLights} = \texttt{on}) \lor \ldots \\ \texttt{cWallLights} & \text{otherwise} \end{cases}$$

Figure 5: Function defined by event table for cWallLights

3 Specifying the LCS in SCR

To illustrate the above process, we apply it to the Light Control System (LCS) described in [1]. The LCS controls the ambient light level in a collection of offices and corridors. Each office contains a group of wall lights and a group of window lights. The LCS has two possible light scenes, a “chosen” light scene and a “default” light scene. For each light scene, the user sets the desired light level in lux and the desired distribution of the lighting between the wall and window light groups. When someone enters an empty office, the default light scene defines the office light setting. Users may choose a different light scene when they are in the office. When an office is reoccupied within T1 minutes after the last person left the office, the chosen light scene is re-established. When an office is unoccupied for more than T3 minutes, the system must turn the lights off. The value of T1 is set by a user of the office, whereas the value of T3 is set by a Facilities Manager. The Facilities Manager can also push an override button to turn off both lighting groups in an empty office. To the extent feasible, the system must use natural light to achieve the desired light level.

To specify the requirements of the LCS in SCR, we have made the following simplifying assumption. We only specify the required behavior of the control system for a single office. Extending this specification to make it applicable to all offices should be straightforward. Also, since the required behavior of the control system for a section of the corridors is a special case of the required behavior for an office, it should be possible to *specialize* the specification for offices to corridors. However, this paper does not discuss the details of this specialization. An important engineering consequence of this approach is that contractors who ultimately develop an implementation would only be required to implement *one* lighting system instead of *two* (one for offices and one for corridors), thus resulting in considerable savings in time and cost.

This section applies each step of the four-step process described in Section 2 to create the LCS specification. Section 3.1 describes the SRS for the LCS by specifying the monitored and controlled variables and the relation REQ. Section 3.2 describes the input and output variables associated with the LCS hardware devices and how these variables are related to the monitored and controlled variables. Section 3.3 describes the SoRS by specifying the relations D_IN and D_OUT, i.e., how estimates of the monitored variables are computed from values of the input variables and how the values of the controlled variables are written to output devices. Finally, Section 3.4 shows how the SoRS and the SRS are extended to report hardware malfunctions and how timing constraints are added to the specifications. The complete SCR requirements specification of the LCS is located at `http://chacs.nrl.navy.mil/LCS`.

3.1 System Requirements Specification

The SRS that we have developed for the LCS contains 21 variables: 12 monitored variables, four controlled variables, a single mode class, and four terms. Below, we describe how we applied the following four steps to create the SRS:

1. Identify and describe the controlled variables.
2. Identify and describe the monitored variables.
3. Identify and describe the mode class(es).
4. Specify the required relation REQ between the monitored and the controlled variables.

3.1.1 Identifying the Controlled Variables

In the LCS, the controlled variables – i.e., the environmental quantities that the system controls – are the two groups of lights, the wall lights and the window lights. For each group, the system determines the light level in lux and whether the light group is on or off. We represent the desired ambient light levels provided by the wall and window lights with the two controlled variables `cWallLL` and `cWindowLL` and the on/off status of each group of lights with the controlled variables `cWallLights` and `cWindowLights`. Figure 6 lists the controlled variables for the LCS along with their types, initial values, and brief descriptions.

Name	Type	Init. Val.	Description
cWallLL	yLightLevel	0	Intensity level of wall lights
cWindowLL	yLightLevel	0	Intensity level of window lights
cWallLights	yLight	off	On/off status of wall lights
cWindowLights	yLight	off	On/off status of window lights

Figure 6: Controlled Variables of the Light Control System

Name	Type	Init. Val.	Description
mOccupied	boolean	*false*	*True* when office is occupied
mT1	yTimer	10	Time to compute reoccupancy
mT3	yTimer	15	Time until empty room
mFMOverride	boolean	*false*	*True* if Fac. Mgr Override
mWallLights	yLight	off	On/off status of wall lights
mWindowLights	yLight	off	On/off status of window lights
mDefLSVal	yLightLevel	100	Default ambient light level
mChosenLSVal	yLightLevel	200	Chosen ambient light level
mDefLSOpt	yOption	wall	Default ambient light option
mChosenLSOpt	yOption	wall	Chosen ambient light option
mIndoorLL	yLightLevel	0	Level of nat. light in office
time	integer	0	System time

Figure 7: Monitored Variables of the Light Control System

According to the system description [1], both the system and the environment control the on/off status of the wall and window lights. To allow a user to turn the wall and window lights off and on manually, each office contains two switches, one to control the wall lights and another to control the window lights. The two controlled variables `cWallLights` and `cWindowLights` are used in the specification to indicate the desired status (on or off) of the wall lights and the window lights, respectively. Because the environment can turn each group of lights on and off independently of the system, the LCS specification must take these environmental actions into account.

3.1.2 Identifying the Monitored Variables

Next, we identify the environmental quantities that the system monitors to determine when to change the on/off status and the light level of each light group. Figure 7 lists the monitored variables along with their types, initial values, and brief descriptions. The status of the office – whether it is occupied or not – is represented by the boolean monitored variable `mOccupied` and the two time intervals by the monitored variables `mT1` and `mT3`. The boolean monitored variable `mFMOverride`, which represents the state of the Facilities Manager override pushbutton, is true when the pushbutton is depressed and false otherwise. The on/off status of each light group is represented by the monitored variables `mWallLights` and `mWindowLights`. Because both the LCS and the environment can change the on/off status of the lights, these monitored variables are needed to keep the values of the corresponding controlled variables, `cWallLights` and `cWindowLights`, consistent with the actual on/off status of each lighting group. The selected light level and light distribution are represented as `mChosenLSVal` and `mChosenLSOpt`

Name	Type	Units
yLightLevel	integer in [0, 10000]	lux
yLight	enum in {off, on}	-
yTimer	integer in [0, 30]	minutes
yOption	enum in {both, wall, window}	-

Figure 8: Type dictionary for the Light Control System

for the chosen light scene and `mDefaultLSVal` and `mDefaultLSOpt` for the default light scene. System time is represented by the distinguished monitored variable `time`, a nondecreasing, non-negative variable implicit in all SCR specifications. In the LCS specification, `time` is an integer with initial value 0, whose units are minutes.

The amount of natural light in an office is represented by the monitored variable `mIndoorLL`. We decided that the outdoor light level described in [1] is not a monitored quantity. The monitored quantity of interest is instead the amount of natural light *in the office*, i.e., `mIndoorLL`. In Section 3.3, we assume that we can use information from the outdoor light sensor and other information to estimate the value of `mIndoorLL`.

In specifying the controlled and the monitored variables, a number of user-defined types are useful. These are listed in Figure 8. Note that the distribution of the light level between the wall and window groups is either `both`, `wall`, or `window`, where `both` indicates that the light level should be distributed equally among the two light groups, and `wall` (`window`) indicates that, to the extent possible, the wall (window) lights are used to achieve the desired light level and the remaining light is provided by the window (wall) group.

The information about the monitored and controlled variables in Figures 6-8 can be regarded as part of the NAT relation. SCR specifications also include an assumptions dictionary in which additional information about NAT can be recorded. One example of a NAT assumption required in the LCS specification is the constraint $\texttt{cWallLL} \leq 5000 \wedge \texttt{cWindowLL} \leq 5000$, which restricts the maximum brightness of each lighting group to at most 5000 lux as required by the LCS description [1]. (The definition of the data type `yLightLevel` allows the light level to be as high as 10000 lux.) A second example is a constraint which keeps the monitored and controlled variables representing the current on/off status of each light group consistent. For example, whenever the system changes the wall lights to on by setting $\texttt{cWallLights}'$ to `on`, the environment must change the monitored variable `mWallLights` to `on` to keep the system and the environment consistent. In SCR specifications, we usually assume that the updating of the monitored variable occurs in the next step after the system has changed the corresponding controlled variable.

3.1.3 Identifying the Mode Class(es)

As noted above, each *mode* in a mode class defines an equivalence class of system states. Modes are useful in defining the required relation REQ between the monitored and controlled variables; in particular, each mode divides the definition of many of the functions used to define REQ into different parts. In the LCS, the value of each of the four controlled variables is a function of the occupancy

Mode Class = `mcStatus`			Trac.
Old Mode	Event	New Mode	
`unoccupied`	`@T(mOccupied)`	`occupied`	U1, U2, U4
`occupied`	`@F(mOccupied)`	`temp_empty`	U3, U4
`temp_empty`	`@T(DUR(NOT mOccupied) > mT3)`	`unoccupied`	U4
	`@T(mOccupied)`	`occupied`	U3

Figure 9: Mode transition table for the mode class `mcStatus`

status of an office. Hence, we define a mode class called `mcStatus` to indicate when an office is occupied, when it is temporarily empty (i.e., unoccupied for up to T3 minutes), and when it is unoccupied (i.e., unoccupied for more than T3 minutes). Figure 9 contains a *mode transition table* which specifies new values for the mode class `mcStatus` as a function of the monitored variables `mOccupied` and `mT3`. We assume that in the initial state, `mcStatus` has the value `unoccupied`.

3.1.4 Specifying the Required Relation REQ

After the environmental variables and the mode classes have been defined, the next step is to specify the relation REQ. To do so, we must define each of the four controlled variables listed in Figure 6 as a function of the monitored variables listed in Figure 7 and the mode class `mcStatus`. Specifying the controlled variables, `cWallLights` and `cWindowLights`, which change the on/off status of the two lighting groups, is somewhat complex because the system does not have complete control of the office lighting. Specifying the two remaining controlled variables, `cWallLL` and `cWindowLL`, which define the light level the system must assign to each light group, is even more complex, because the setting for each group depends on the parameters defined for the chosen and default light scenes, the amount of natural light in the office, and whether the Facilities Manager has pushed the override button. To specify the value of `cWallLL` and `cWindowLL` in a concise and understandable manner, we define four terms: two terms define the current desired light level and option, the third term defines the amount of artificial light that is needed, and the fourth term defines whether the Facilities Manager has turned off the lights by pushing the override button when the office is empty.

The event table in Figure 4 (see Section 2.4) describes `cWallLights`′, the new value of the wall lights, as a function of office occupancy, the monitored variables `mWallLights` and `mFMOverride`, and the old value of `cWallLights`. The second row of the table states that if the office becomes occupied, the system must turn the wall lights on, whereas the fourth row states that if the office remains unoccupied after some time interval (i.e., `mcStatus` becomes `unoccupied`), the system must turn the wall lights off. Because our LCS specification represents the on/off status of the wall lights using both a monitored and a controlled variable, the specification must maintain consistency between the two variables. Hence, if `mWallLights` turns to on (because the user turns the wall lights on) when `cWallLights` is off, the first row of the table updates the value of `cWallLights` to reflect this change. Similarly, the third row states that the variable `cWallLights` must reflect the status of the wall lights if the user turns them off, i.e., if `mWallLights` turns to off when `cWallLights` is on. According to the LCS description [1], the Facilities Manager may turn off both light groups

Name	Type	Init. Val.	Description
tOverride	boolean	*false*	True if Fac. Mgr. pushed override
tCurrentLSVal	yLightLevel	0	Current desired light level
tCurrentLSOpt	yOption	wall	Current desired light distribution
tRemLL	yLightLevel	0	Lighting produced by artificial light

Figure 10: Terms Used to Define cWallLL and cWindowLL

Event	tOverride′	Trac.
@T(mFMOverride) WHEN mcStatus $\neq$ occupied	true	FM6
@T(mcStatus=occupied)	false	U1

Figure 11: Event table for term tOverride

Event	tCurrentLSVal′	Trac.
@T(mcStatus = occupied) WHEN (DUR(mcStatus $\neq$ occupied) $\geq$ mT1)	mDefLSVal	U4
@C(mChosenLSVal)	mChosenLSVal′	U2

Figure 12: Event table for term tCurrentLSVal

Event	tCurrentLSOpt′	Trac.
@T(mcStatus = occupied) WHEN (DUR(mcStatus $\neq$ occupied) $\geq$ mT1)	mDefLSOpt	U4
@C(mChosenLSOpt)	mChosenLSOpt′	U2

Figure 13: Event table for term tCurrentLSOpt

when an office is unoccupied or temporarily empty. This behavior is captured by the fifth row of Figure 4, which states that the system must turn off the wall lights when the pushbutton mFMOverride is depressed and the office is not occupied. In all other cases (e.g., when the office becomes temporarily empty), the on/off status of the wall lights is unchanged. An event table similar to the table shown in Figure 4 defines the value of cWindowLights.

Figure 10 lists the four terms used to define the values of cWallLL and cWindowLL along with their types, initial values, and brief descriptions. Figure 11 contains an event table which defines the first term, tOverride, as a function of mFMOverride and the mode class mcStatus. The two terms tCurrentLSVal and tCurrentLSOpt use user inputs and information about office occupancy to determine the desired ambient light level and the distribution of the ambient light among the wall and window lights. The event table in Figure 12, which specifies the new value of the current light level tCurrentLSVal, states that if the office becomes occupied after being unoccupied for at least mT1 minutes, then the light level associated with the default light scene is selected as the current light level, whereas if the chosen light level changes, then the light level associated with the chosen light scene is selected. The new value of tCurrentLSOpt, which is defined by the event table in Figure 13, is similarly defined.

Based on the amount of natural light in the office, the term tRemLL represents the amount of artificial light that is needed to achieve the desired light level.

Condition	cWallLL	cWindowLL	Trac.
tCurrentLSOpt $=$ both	tRemLL/2	tRemLL/2	Def. of light scene
tCurrentLSOpt $=$ wall AND tRemLL > 5000	5000	tRemLL $- 5000$	
tCurrentLSOpt $=$ wall AND tRemLL ≤ 5000	tRemLL	0	
tCurrentLSOpt $=$ window AND tRemLL > 5000	tRemLL $- 5000$	5000	
tCurrentLSOpt $=$ window AND tRemLL ≤ 5000	0	tRemLL	

Figure 14: Condition table for controlled variables cWallLL and cWindowLL

The condition table in Figure 2 (see Section 2.4) states that no artificial light is needed (tRemLL $= 0$) if the office is unoccupied for some period, if the office is temporarily empty or occupied and the amount of natural indoor light exceeds the current desired light level, or if the override button is pushed when the office is temporarily empty. The table states further that when the office is occupied or if the override button has not been pushed when the office is temporarily empty, the amount of needed artificial light is the difference between the current desired light level and the amount of natural indoor light.

Figure 14 shows a condition table which defines the values of the controlled variables cWallLL and cWindowLL. This table states that if the option selected is both, then half of the requested light is provided by the wall lights and the other half by the window lights. If the option selected is wall, then the wall lights are used to provide the requested light level. If the requested light level exceeds the maximum provided by the wall lights, then the remaining light is provided by the window lights. If the option selected is window, then the situation is reversed.

Figure 15 shows the dependency graph for the SRS of the LCS. This graph, which was generated by the dependency graph browser of the SCR* toolset, illustrates the relationship between the 21 variables in the SRS. In the graph, the eleven monitored variables and the distinguished variable time appear as the leftmost nodes and the four controlled variables as the rightmost nodes. The graph shows the importance of the single mode class mcStatus: the value of every dependent variable in the specification depends either directly or indirectly on this mode class. The graph also shows the dependence of the controlled variables cWallLL and cWindowLL on the four terms listed in Figure 10.

3.2 System Design Specification

First, this section describes the two control panels needed in the LCS. Next, the section describes the input and output variables associated with the selected LCS hardware devices and the correspondence between these variables and the monitored and controlled variables specified in Section 3.1. To keep the paper concise, this section omits details of the hardware device interfaces that would be provided to the software (e.g., whether the devices are memory- or I/O-mapped, interrupt driven or polled; their physical addresses; details of their control and data registers; etc). However, these device details need to be recorded in the SDS because the software designers require this information to design and write code for the device-dependent modules.

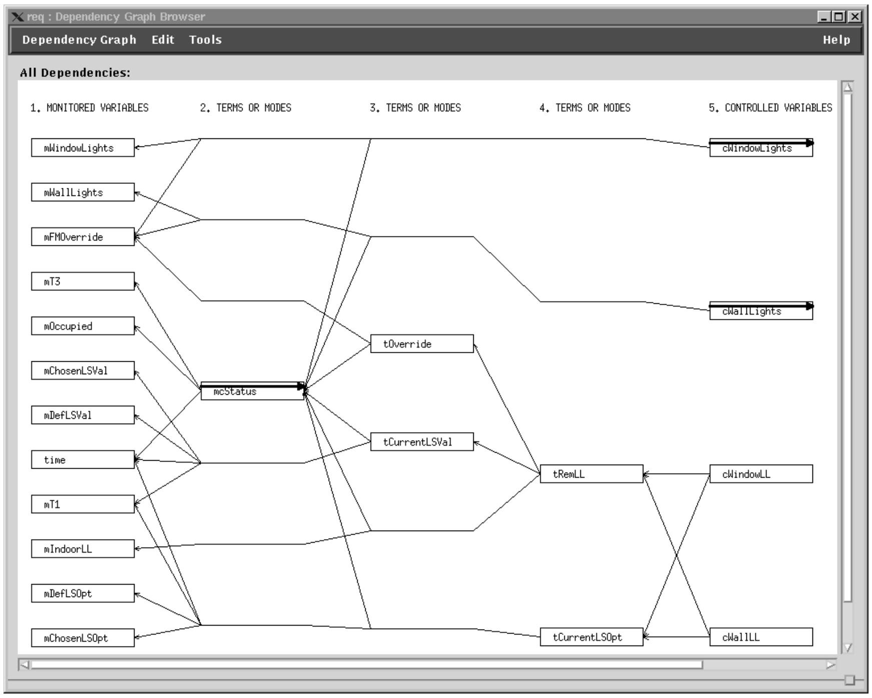

Figure 15: Dependency Graph for Specification of LCS Ideal Behavior

3.2.1 LCS Control Panels

The part of the LCS that we specify includes two control panels: an office control panel and a Facilities Manager control panel. Figure 16, which illustrates the office control panel, contains three sliders, one for setting the value of the time interval `mT1` and the others for setting the light levels defined by the monitored variables `mDefLSVal` and `mChosenLSVal`. Each of the two option switches, which are represented by the monitored variables `mDefLSOpt` and `mChosenLSOpt`, indicates a value in the set {`wall`, `window`, `both`} and describes how the wall and window lights are used to achieve the desired ambient light level. The office control panel also contains the two on/off switches described above for manually turning the wall and window lights off and on. The Facilities Manager control panel, shown in Figure 17, contains the override pushbutton, a slider for setting the value of the time interval `mT3`, and two lights indicating when a sensor malfunction has occurred. One light reports a malfunction of the motion detector, and the other reports a malfunction of the outdoor light sensor.

3.2.2 Specifying the Input and Output Variables

Figure 18 lists the 14 input and six output variables associated with these hardware devices along with their types, initial values, and brief descriptions. The data types of many of these variables are defined in the type dictionary in Figure 8. The only new user-defined type is `yAmbientLevel`, an integer in $[0, 100]$.

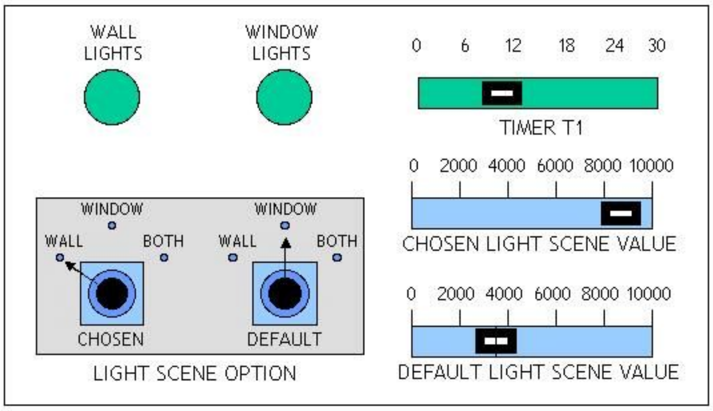

Figure 16: Office Control Panel for the LCS.

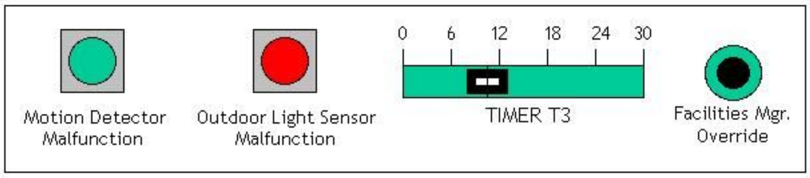

Figure 17: Facilities Manager Control Panel for the LCS.

Figure 19 shows the correspondence between the input variables and the monitored variables. To enable the software to determine whether a room is occupied (i.e., to estimate the value of `mOccupied`), each office is equipped with a passive infrared motion detector and a door closed contact. The output of the motion detector is represented by boolean variable `iMD`, which is *true* when there is movement in the range of the detector and *false* otherwise. The output of the door closed contact is represented by the boolean variable `iDCC`, which is *true* if the door is fully closed and *false* otherwise. The level of ambient light outdoors is sensed by an outdoor light sensor and is represented by the input variable `iOLS`. This variable denotes the value recorded by the outdoor light sensor in lux. The value of the outdoor light sensor is used to compute the indoor light level (represented by the monitored variable `mIndoorLL`). The status lines for the window and wall lights, `iSLLWindow` and `iSLLWall`, sense if the light voltage is high or low and thus are used to estimate whether the window lights and the wall lights, represented by the monitored variables `mWindowLights` and the `mWallLights`, are off or on.

The LCS description states that two hardware devices, the motion detector and the outdoor light sensor, can malfunction. Malfunctions in these de-

Input Data Items			
Name	Type	Init. Val.	Description
`iMD`	`boolean`	*false*	*True* when motion is sensed
`iMDmalfunction`	`boolean`	*false*	*True* if motion detector malfunctions
`iDCC`	`boolean`	*false*	*True* if door is closed
`iDefLSOpt`	`yOption`	`wall`	Default light scene option
`iDefLSVal`	`yLightLevel`	200	Default light scene value
`iChosenLSOpt`	`yOption`	`wall`	Chosen light scene option.
`iChosenLSVal`	`yLightLevel`	100	Chosen light scene value
`iT1`	`yTimer`	10	Time to compute reoccupancy
`iT3`	`yTimer`	15	Time to empty room
`iOLS`	`yLightLevel`	0	Output of outdoor light sensor
`iOLSmalfunction`	`boolean`	*false*	*True* if outdoor light sensor malfunctions
`iFMOverride`	`boolean`	*false*	Override button of Fac. Mgr.
`iSLLWindow`	`boolean`	*false*	Status of window lights
`iSLLWall`	`boolean`	*false*	Status of wall lights

Output Data Items			
Name	Type	Init. Val.	Description
`oPulseWall`	`boolean`	*false*	Toggles wall lights on and off
`oPulseWindow`	`boolean`	*false*	Toggles window lights on and off
`oDimmerWall`	`yAmbientLevel`	0	Controls wall light
`oDimmerWindow`	`yAmbientLevel`	0	Controls window light
`oOLSmalfunction`	`boolean`	*false*	Turns on OLSmalfunction light
`oMDmalfunction`	`boolean`	*false*	Turns on MDmalfunction light
`oCSAWall`	`boolean`	*false*	Sends signal to wall lights
`oCSAWindow`	`boolean`	*false*	Sends signal to window lights

Figure 18: Input and Output Variables for the LCS

vices are represented by assigning the input variables `iMDmalfunction` and `iOLSmalfunction` the value *true*. Unlike the other input variables, neither of these input variables is (at this point) associated with any monitored variable. This is because the ideal system behavior is defined without knowledge of the selected hardware devices. Section 3.4 describes how the set of monitored and controlled variables is extended to include variables that report hardware malfunctions.

The LCS description of dimmable lights also includes a signal "control system active" that is to be sent to each light cluster every 60 seconds. If the signal does not arrive on time, the corresponding light cluster enters a fail-safe mode in which the brightness is assumed to be 100%. We define two output variables, `oCSAWall` and `oCSAWindow`, one for each light cluster, to represent these signals.

Each of the seven remaining input variables is associated with a single monitored variable with the same name, except for the prefix. Thus, the values of the seven input variables, `iDefLSVal`, `iChosenLSVal`, `iT1`, `iT3`, `iDefLSOpt`, `iChosenLSOpt`, and `iFMOverride`, are used to estimate the values of the corresponding monitored variables, `mDefLSVal`, `mChosenLSVal`, `mT1`, `mT3`, `mDefLSOpt`, `mChosenLSOpt`, and `mFMOverride`.

Figure 20 shows the correspondence between the controlled variables and the output variables. The output variables `oPulseWindow` and `oPulseWall` determine the on/off status of the window and wall lights, whereas the output

Input Variable	Monitored Variable
iMD	mOccupied
iMDmalfunction	-
iDCC	mOccupied
iDefLSOpt	mDefLSOpt
iDefLSVal	mDefLSVal
iChosenLSOpt	mChosenLSOpt
iChosenLSVal	mChosenLSVal
iT1	mT1
iT3	mT3
iOLS	mIndoorLL
iOLSmalfunction	-
iFMOverride	mFMOverride
iSLLWindow	mWindowLights
iSLLWall	mWallLights

Figure 19: Correspondence between Input and Monitored Variables

Output Variable	Controlled Variable
oPulseWall	cWallLights
oPulseWindow	cWindowLights
oDimmerWall	cWallLL
oDimmerWindow	cWindowLL
oMDmalfunction	-
oOLSmalfunction	-
oCSAWall	-
oCSAWindow	-

Figure 20: Correspondence between Controlled and Output Variables

variables `oDimmerWindow` and `oDimmerWall` determine the brightness of the corresponding lights. The values of the latter variables range between 0 and 100. Lights on the Facilities Manager's control panel reporting hardware malfunctions are represented by the output variables `oMDmalfunction` and `oOLSmalfunction`.

3.3 Software Requirements Specification

As described above, the SoRS is organized into two device-dependent modules and a single device-independent module. Because the required behavior of the device-independent module is already defined by the relation REQ in the SRS, what remains is to specify the input and output device interface modules, i.e., the relations D_IN and D_OUT.

3.3.1 Specifying the Relation D_IN.

The relation D_IN specifies how the input variables listed in Figure 19 are used to compute estimates of the monitored variables. (In our approach, estimates of the monitored and controlled variables are denoted as $\widetilde{\mathtt{mOccupied}}$, $\widetilde{\mathtt{mT1}}$, etc. To improve readability, we have omitted the tildes in the tables and the rest of this section.) Estimating the values of five of the monitored variables – `mChosenLSVal`,

Condition	mWallLights
iSLLWall	on
NOT iSLLWall	off

Figure 21: Condition table for estimate of mWallLights

Event	mOccupied′
@T(iMD)	true
@F(iMD) WHEN (NOT iDCC OR (DUR(iDCC AND iMD) < 1))	false

Figure 22: Event table for estimating mOccupied

mChosenLSOpt, mT1, mT3, and mFMOverride – from the corresponding input variables – iChosenLSVal, iChosenLSOpt, iT1, iT3, and iFMOverride – is trivial. In each case, the estimated value of the monitored variable is simply the value of the corresponding input variable, i.e., mT1 = iT1, mT3 = iT3, etc.

Obtaining estimates of the values of monitored variables mWallLights and mWindowLights from the values of the corresponding input variables iSLLWall and iSLLWindow is also straightforward. The condition table in Figure 21 states that the wall lights, represented by mWallLights, are on if iSLLWall is true and off if iSLLWall is false. A similar condition table defines the value of mWindowLights as a function of the input variable iSLLWindow.

Obtaining the estimated value of the monitored quantity mOccupied is more complex and may be done in many ways. One way is to set mOccupied = iMD; that is, the estimate is that the room is occupied iff the output of the motion detector is *true*. However, if a room is occupied but there is insufficient motion to trigger the motion detector, iMD will be false, thereby providing an inaccurate estimate of room occupancy. To better estimate the value of mOccupied, we use the output of the door contact iDCC in conjunction with the value read by the motion detector iMD. The event table in Figure 22 states that the value of mOccupied is *true* whenever the output of the motion detector (i.e., the input variable iMD) becomes *true*. When iMD becomes *false*, mOccupied remains *true* if the door has been fully closed *and* iMD has been *true* for a continuous period of at least a minute (if this is the case, the presence of a motionless person in an office is highly likely), and is set to *false* otherwise.

Section 3.4 describes how we estimate the values of the monitored variables representing the indoor light level, mIndoorLL, and the default light level and option, mDefLSVal and mDefLSOpt.

3.3.2 Specifying the Relation D_OUT

Relation D_OUT specifies how estimates of the controlled quantities are used to drive the output devices. The variables cWallLL and cWindowLL are used to control the two clusters of dimmable lights. Each light group has a *dimmer* line. The values of the dimmers range between 0% and 100%, where 0% corresponds to the associated light group being off and 100% corresponds to the lights being fully on. The dimmers are specified as oDimmerWall = cWallLL/50 and oDimmerWindow = cWindowLL/50.

Event	oPulseWall′
@T(cWallLights = on) AND NOT iSLLWall′ OR @T(cWallLights = off) AND iSLLWall′	true
@T(iSLLWall) WHEN cWallLights = on OR @F(iSLLWall) WHEN cWallLights = off	false

Figure 23: Event table for computing output oPulseWall

Each light group also has a *pulse* line. A change in the value of a pulse line from *false* to *true* toggles the status of the corresponding light group, i.e., the lights will be turned on if they are off and off if they are on. The event table in Figure 23 defines the new value of the pulse line for the wall lights, oPulseWall, as a function of the controlled variable cWallLights and the input variable iSLLWall.

Because changing oPulseWall to true to turn the wall lights on when they are already on will actually turn the wall lights off (and changing oPulseWall to true to turn the wall lights off when they are already off will actually turn the wall lights on), the wall lights should be turned on by setting oPulseWall to true only when the wall lights are off and should be turned off by setting oPulseWall to true only when the wall lights are on. To obtain this behavior, we need the current status of the wall lights (on or off) to compute the value of oPulseWall. We obtain this information from the input variable iSLLWall. This violates the rule described in Section 2.3 stating that only controlled variables should be used to calculate the values of the output variables. However, because the input device, the status line, and the output device, the pulse line, are part of the same device *and* because any future input device and output device that monitor and control the on/off status of the lights are likely to be coupled (and hence changes will most likely involve all the associated variables), we can communicate output data from the status line directly to the software that controls the value of the pulse line without violating the information hiding principle. An event table similar to that shown in Figure 23 defines the value of oPulseWindow as a function of the controlled variable cWindowLights and the input variable iSLLWindow.

3.4 Handling Hardware Malfunctions and Timing Constraints

Practical systems must be designed to tolerate and report hardware malfunctions and to satisfy timing constraints. This section illustrates how SCR handles LCS hardware faults and specifies LCS timing constraints. This is the last step of our four-step process.

3.4.1 Hardware Malfunctions

The input variables, iMDmalfunction and iOLSmalfunction, represent malfunctions of the motion detector and the outside light sensor, respectively. Each input variable is true when the corresponding device fails and false otherwise. When the motion detector fails, the LCS description [1] states that the office should be treated as occupied. In our LCS specification, this means that the monitored variable mOccupied is *true*. To include this fault tolerant behavior, we replace

Event	mOccupied′	Trac.
@T(iMD) OR @T(iMD_malfunction)	true	NF4
@F(iMD) WHEN ((NOT iDCC OR (DUR(iDCC AND iMD) $<$ 1)) AND NOT iMD_malfunction) OR @T(DUR(NOT iMD_malfunction) $\geq$ 2)	false	NF4

Figure 24: Event table for mOccupied modified to handle device malfunction

Condition	mDefLSVal	mDefLSOpt	Trac.
NOT iOLSmalfunction	iDefLSVal	iDefLSOpt	-
iOLSmalfunction	10000	both	NF2

Figure 25: Fault-tolerant specification of mDefLSVal and mDefLSOpt

Event	mIndoorLL′	Trac.
@C(iOLS) WHEN (NOT iOLSmalfunction)	RI(iOLS)	NF1

Figure 26: Fault-tolerant specification of mIndoorLL

the event table for mOccupied in Figure 22 by the table shown in Figure 24. In the revised table, mOccupied is set to true if motion is detected *or* if the motion detector malfunctions (row 1). It is not allowed to go false as long as the malfunction persists (rows 2 and 3) and is set to false when there is no malfunction for a period of two minutes.

A malfunction in the outdoor light sensor affects the values of two monitored variables, mDefLSVal and mDefLSOpt. As shown in Figure 25, when the outdoor light sensor is functioning properly, the values of these variables are the same as the values of the corresponding input variables, iDefLSVal and iDefLSOpt. However, when the sensor malfunctions, the variables are assigned the values 10000 and both. A malfunction in the outdoor light sensor also affects the estimate of the monitored variable mIndoorLL, which records the indoor light level. When the outdoor light sensor is working, the indoor light level is specified as a function RI of the outdoor light level. (This function is not defined here; we assume that such a function would be defined with the aid of domain experts.) When the sensor malfunctions, the value of the indoor light level is the last value that was read when the sensor was working. Figure 26 provides an event table for computing an estimate of the indoor light level.

According to the LCS description [1], the LCS must turn on a light on the Facility Manager's control panel when the motion detector malfunctions. To support this behavior in a manner that satisfies information hiding, we define a new boolean monitored variable mMDmalfunction, which has the value true iff the input variable iMDmalfunction is true, and a new controlled variable cMDmalfcnLight, which has type yMalfcnLight, where yMalfcnLight is an enumerated type with values red and green. Then, cMDmalfcnLight has the value red if mMDmalfunction is true and the value green otherwise. The output variable oMDmalfunction is defined by the condition table in Figure 27 as a function of cMDmalfcnLight. We use the same approach to create a new monitored

Condition	oMDmalfunction
cMDmalfcnLight = red	true
cMDmalfcnLight = green	false

Figure 27: Notification of motion detector malfunction

Event	oCSAWall′
@T(DUR(NOT oCSAWall) $\geq$ 30)	true
@T(DUR(oCSAWall) $\geq$ 30)	false

Figure 28: Event table defining signal `oCSAWall` for wall lights

variable `mOLSmalfunction` and a new controlled variable `cOLSmalfcnLight` and to define `mOLSmalfunction`, `cOLSmalfcnLight`, and `oOLSmalfunction`.

Figure 28 contains an event table that defines `oCSAWall`. (In this table, the unit of time is seconds and not minutes as in the other tables. Hence, conversion of minutes to seconds is required when this function is used in conjunction with the other functions in the LCS specification.) The first row of the table states that `oCSAWall` is set to true when it has remained false for 30 seconds; the second row states that `oCSAWall` is set to false when it has remained true for 30 seconds. This ensures that every 60 seconds the control system sends a signal, i.e., a transition from false to true, to the wall light cluster. A similar table defines the output variable `oCSAWindow`.

Adding the fault-tolerant behavior illustrates how the relation REQ specified as part of the SRS differs from the $\widetilde{\mathrm{REQ}}$ specified at the level of the SoRS. When we specify the ideal system behavior, there are no hardware failures or malfunctions, so notification of any malfunction is absent. In contrast, in the *real* system, hardware may fail. At times, the system will be fault-tolerant and hide malfunctions from users. In other cases, the system must notify the user about a hardware malfunction so that corrective action can be taken. We describe information about hardware malfunctions that are visible externally in $\widetilde{\mathrm{REQ}}$.

Extending the LCS specification to handle notification of hardware malfunctions required four more variables: two monitored variables `mMDmalfunction` and `mOLSmalfunction` and two controlled variables `cMDmalfcnLight` and `cOLSmalfcnLight`. Once these variables are added, the LCS specification contains a total of 45 variables: six controlled variables, 13 monitored variables, one mode class, four terms, 14 input variables, and six output variables together with the distinguished variable `time`. The relationship between these variables is shown in the dependency graph in Figure 29. (Because the variable names become unreadable, we have omitted them from the graph.) This graph, created by the dependency graph browser in the SCR* toolset, illustrates the complexity of the LCS specification. In the graph, the leftmost nodes represent input variables and the rightmost nodes represent output variables. As required, the nodes representing input variables are only connected to nodes representing monitored variables and the nodes representing output variables are only connected to nodes representing controlled variables. The only exceptions are the two nodes in the top right-hand corner of Figure 29 representing the output variables `oPulseWall` and `oPulseWindow`. As stated in Section 3.3, each of these output variables depends on both a controlled variable *and* an input variable. The output variables

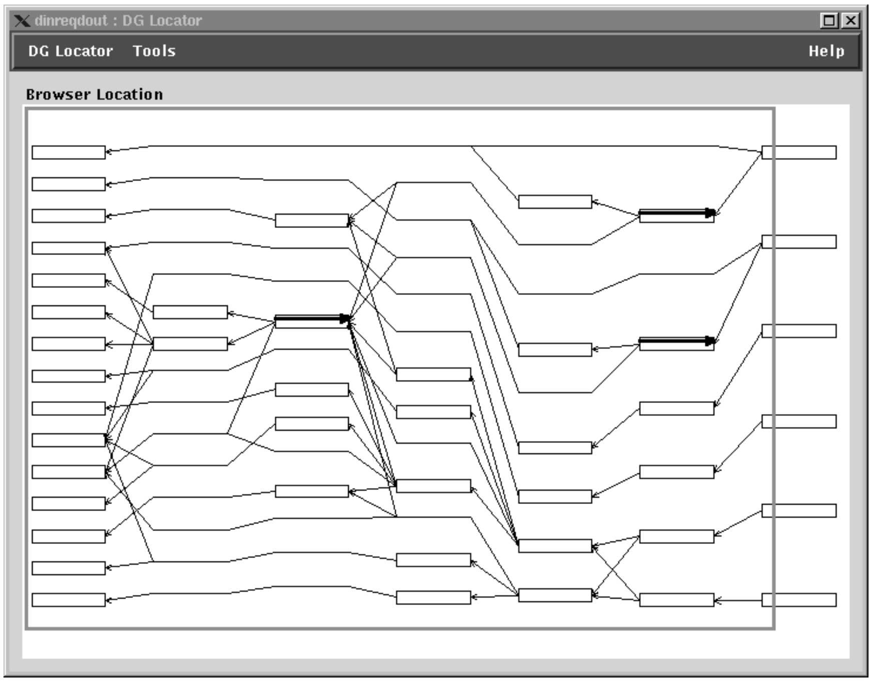

Figure 29: Dependency Graph for Requirements Specification of LCS

`oCSAWall` and `oCSAWindow` do not appear in the graph shown in Figure 29, because they only depend on time and thus can be considered independently in another much simpler specification.

3.4.2 Timing Constraints

In addition to tolerating and reporting hardware failures, the LCS must also satisfy a number of timing constraints. As noted above, we assume that a timing constraint may be associated with every controlled variable. In our four-step process, first, a system requirements specification is constructed that omits timing. Once the developers have significant confidence in the untimed specification, then they may attach timing constraints to each controlled variable. To illustrate how our approach associates timing constraints with controlled variables, we add a timing constraint to the condition table describing the values of the controlled variables `cWallLL` and `cWindowLL`. Figure 30 indicates that the light level must be updated no later than 10 seconds after the monitored variable change that triggered the light level update. That is, if the user changed the light level of the chosen light scene from 3000 lux to 4000 lux at time t, then the wall lights and window lights must be set to the new light level within $t + 10$ seconds. In Figure 30, the constraint is represented as $[0, 10]$, where 0 is the lower bound and 10 is the upper bound. This timing constraint is described in seconds. Given that most other times in the specification are described in minutes, unit conversion from minutes to seconds will be needed in simulation and in reasoning about the system timing behavior.

Condition	cWallLL	cWindowLL
tCurrentLSOpt = both	tRemLL/2	tRemLL/2
tCurrentLSOpt = wall AND tRemLL $>$ 5000	5000	tRemLL $-$ 5000
tCurrentLSOpt = wall AND tRemLL $\leq$ 5000	tRemLL	0
tCurrentLSOpt = window AND tRemLL $>$ 5000	tRemLL $-$ 5000	5000
tCurrentLSOpt = window AND tRemLL $\leq$ 5000	0	tRemLL
Timing constraint	$[0, 10]$ sec.	$[0, 10]$ sec.

Figure 30: Table for cWallLL and cWindowLL with timing constraint

Event	mOccupied′
@T(iMD)	true
@F(iMD) WHEN (NOT iDCC OR (DUR(iDCC and iMD) $<$ 1))	false
Timing constraint	$[0.0, 0.1]$ sec

Figure 31: Event table for mOccupied with timing constraint

In addition to associating a timing constraint with each controlled variable, we also associate timing constraints with each table that defines the estimate of a monitored variable or the value of an output variable. For example, Figure 31 states that the estimate of the monitored variable mOccupied must be updated within 0.1 seconds after the room became occupied; i.e., if the room became occupied at time t, then the estimate must be computed within $t + 0.1$ seconds. The triggering time t is the time that the change in the monitored quantity actually occurred, not the time that the motion detector detected motion or the time that the software sampled the values of the motion detector and the door closed contact.

In a similar fashion, timing constraints can be added to tables which use the values of controlled variables to compute the output variables. By considering the times needed to compute estimates of the monitored variables from the input variables, the times needed to write the values of the controlled variables to output devices, and the timing constraints associated with the controlled variables, one can deduce the amount of time the system has available to compute the values of the controlled quantities. One can thus determine the feasibility of constructing a system that satisfies the end-to-end timing requirements.

4 Applying the SCR* Toolset

We used the specification editor to develop the SCR specification of the required behavior of the LCS and three analysis tools in the SCR* toolset to check the specification for desired properties. These tools were the consistency checker, the simulator, and a new analysis tool called Salsa [6]. Salsa analyzes a specification for desired properties using an algorithm based on Binary Decision Diagrams (BDDs) and a linear integer constraint solver. We performed the analysis in two stages. First, we used the tools to analyze the SRS, i.e., the specification presented in Section 3.1. Once we had confidence in the quality of this specification, we used the specification editor to add the refinements and extensions presented in Sections 3.2-3.4 to the SRS and again applied our analysis tools.

We used our automated consistency checker [11] to check for syntax and type errors, missing cases, nondeterminism, and other application-independent properties. Applying the consistency checker exposed some minor errors, such as incorrect variable names. Applying the tool Salsa was useful for consistency checking and in addition helped to verify application properties. Salsa detected an instance of nondeterminism in the definitions of variables `oPulseWall` and `oPulseWindow`, which we subsequently corrected in the final version of the specification. Later, we used the simulator of the SCR* toolset [10] to symbolically execute the requirements specification to ensure that the formal specification correctly captures the customers' intent. Running scenarios through the simulator exposed flaws in the definitions of the controlled variables `cWallLights` and `cWindowLights`, which we corrected in the final version of the specification.

We also formulated the following three application properties:

- P1 When `mFMOverride` is set to true when an office is unoccupied or temporarily empty, (i.e, `mcStatus` is not `occupied`), then `cWindowLL` and `cWallLL` are both set to zero.
- P2 When `mcStatus` is `unoccupied`, then `cWindowLL` and `cWallLL` are both zero.
- P3 If `mcStatus` is `unoccupied`, then `cWallLights` and `cWindowLights` are off.

All three properties were verified formally using the tool Salsa on an early version of the SRS. However, after we redefined the controlled variables `cWallLights` and `cWindowLights`, Property P3 proved to be false. The problem is that nothing is assumed in the specification about the ability of users to switch the wall (or window) lights on when an office is unoccupied (`mStatus` = `unoccupied`). One way to prove this property is to include in NAT an assumption that light groups cannot be turned on unless the office is occupied (`mStatus` = `occupied`). However, this assumption may be incorrect if it is possible to operate the user control panel, which is portable, from outside the office.

Analyzing the LCS specification with our tools increased our confidence in the correctness of the specification. Further analysis would give us even more confidence. One benefit of the three tools that we used to analyze the LCS specification is that applying these tools is relatively easy. Two other tools associated with the SCR* toolset could also be used but applying them would require more effort. These tools are the model checker Spin [14] and the TAME tool [4], an interface to the theorem prover PVS. Because the LCS specification contains many numbers and large ranges of numbers (e.g., the light level can vary between 0 and 10,000), its state space is very large. Hence, a barrier to using a model checker is the state explosion problem. Running TAME requires the ability to do deductive reasoning and some knowledge of PVS. Another problem is that the LCS description does not list a set of application properties that the LCS specification must satisfy. Hence, we would need to formulate a set of properties.

5 Discussion

The development of the LCS requirements specification using SCR raised a number of issues. Below, we discuss these issues and describe how each was resolved.

5.1 Managing Complexity

The LCS is a relatively complex system. Specifying just the behavior of the system that controls the lighting in a single office required significant effort and resulted in the moderately large specification presented in Section 3. A systematic way to specify and to organize the LCS requirements was therefore crucial. The four-step process that we applied appears to be effective for handling requirements specifications of moderate to high complexity. However, more experience is needed in applying the process to practical systems before we can conclude that the process is useful.

5.2 Focusing on the Essential Behavior

Specifying the ideal system behavior in step 1 of our process focuses attention on the essence of the required system behavior and postpones consideration of the more detailed requirements, e.g., the selected input and output devices, the system's timing requirements, how to represent fault-tolerance, and how to specify and report hardware malfunctions. In specifying the ideal behavior of the LCS first, we are also forced to clarify the required system behavior. For example, for the ideal system behavior, we want to know whether an office is occupied, *not* whether motion has been detected in the office or whether a motion detector has failed.

The specification of the ideal behavior should not be influenced by the particular input and output devices that have been selected. For example, our initial specification of the controlled variables `cWallLights` and `cWindowLights` was influenced by the special characteristics of the pulse line, the output that turns the lights on and off. Once aware of this bias, we respecified the required behavior in a manner that omits bias toward the selection of a particular hardware device.

5.3 System and Environmental Control of a Variable

Defining a variable that is controlled both by the system and by the environment required careful thought. To represent the on/off status of each lighting group, we defined both a monitored variable – to keep track of when the environment turned the lights on and off – and a controlled variable – to indicate when the system needs to turn the lights on or off. Maintaining the consistency of the two variables also required careful thought.

The definition of `cWallLights` presented in Figure 4 and the NAT assumption described in Section 3.1.2 are designed to solve these problems. The specification in Figure 4 turns the lights on or off in response to specified events. It also uses the value of the monitored variable `mWallLights` to ensure that the value of the controlled variable `cWallLights` is consistent with the current on/off status of the wall lights when the user turns them on and off manually. The NAT assumption described at the end of Section 3.1.2 ensures that the environment will change `mWallLights` to the proper value when the system changes the value of `cWallLights` in order to turn the lights on or off.

5.4 The Importance of Tool Support

Applying tools to our LCS specification had three benefits. First, a number of errors in the specification were identified and corrected with only a small investment in human time and effort. Second, several important questions were raised about the required system behavior and about the underlying assumptions about the system environment (e.g., where are the light switches located?). Many of these errors and questions were unlikely to have been identified by inspection. The LCS is simply too big and too complicated. Finally, running the consistency checker and finding no problems, running a series of scenarios through the simulator and finding that the simulated behavior was consistent with the expected behavior, and verifying application properties such as P1–P3 increased our confidence in the correctness of the specification.

5.5 Organizing the Requirements

An issue is how the three products described in Section 3 – the SRS, the SDS, and the SoRS – are organized into requirements documents. In our view, the information provided could be presented as a single document or as three separate documents. Probably more important is that these specifications can be organized as a database from which many different documents could be generated. Further, users could find answers to queries using the database when they want to know how the specification addresses particular questions about the required behavior of the LCS.

5.6 Redundancy in the Specification

One important question is why the software is required to copy the value of an input variable to the corresponding monitored variable and the value of a computed controlled variable to an output variable. For example, why not use the value of the input variable `iChosenLSVal` rather than the value of the corresponding monitored variable `mChosenLSVal` to compute the term `tCurrentLSVal` (see Figure 12)? One reason has been already stated: Hiding the identity of the specific I/O devices that the system uses from the software that computes the values of the controlled variables leads to software that is organized for ease of change. The other reason is ease of development. We can specify REQ first and then refine the specification by adding the specifications of D_IN and D_OUT later. Adding the specification of D_IN and D_OUT in this manner *will not change* the specification of REQ.

5.7 Requirements Not Addressed

The requirements specification in this paper addresses all of the user, facility manager, and non-functional needs (including fault tolerance and user interface requirements), with the following exceptions:

1. The office control panel does not include lights to indicate failure of the outdoor light sensor and the motion detector. These lights are added trivially, since their behavior is analogous to the behavior of the corresponding lights on the Facilities Manager control panel.

2. We made no attempt to address the facility manager need FM9, since this need cannot be reasonably met without employing an energy meter.
3. Our specification does not address the facility manager need FM11, i.e., provide the ability to the facility manager to enter malfunctions manually. Adding this capability to the specification is straightforward, and we leave this as an exercise to the reader.

6 Related Work

The process described above for constructing a requirements specification was significantly influenced by Parnas' Four Variable Model [19] but differs in two fundamental ways. First, the Parnas model uses the IN and OUT relations to represent the required tolerances, i.e., the precision with which the input devices must measure the values of the monitored quantities and the output variables must assign values to the controlled quantities. Although such information would be useful, the specification described in Section 3 provides no information (except some minimal timing constraints) about the required tolerances. Instead, we use the D_IN and D_OUT relations to describe how the software is required to transform data from the input devices into estimates of the monitored quantities and to write estimates of the controlled quantities to the output devices. Just as we use tables to represent the REQ relation, we also use tables to represent the D_IN and D_OUT relations.

Second, whereas in the Parnas model, the relation SOFT, which describes the required software behavior, is derived from the REQ, NAT, IN, and OUT relations, we specify the required behavior in terms of the relations NAT, D_IN, D_OUT, and $\widetilde{\mathrm{REQ}}$. As described above, we decompose the system software into three modules, the device-independent module and the two device-dependent modules, and formally specify the required behavior of each module. The D_IN and D_OUT relations define the required behavior of the device-dependent modules. The relation $\widetilde{\mathrm{REQ}}$ defines the required behavior of the device-independent module. As noted above, $\widetilde{\mathrm{REQ}}$ contains more information than REQ because, $\widetilde{\mathrm{REQ}}$ describes the required system response to hardware malfunctions and has as inputs the *estimated values* of the monitored variables rather than the exact values of the monitored variables.

An approach similar to ours, also influenced by the Four Variable Model, has been developed by Thompson, Heimdahl, and Miller [21] in the context of simulation. As in our approach, the required system behavior is initially specified as a relation between monitored and controlled variables. Then, the specification is refined by using input devices to estimate values of the monitored quantities and by writing values of the controlled variables to output devices. Thompson et al. refer to the inverses, IN^{-1} and OUT^{-1}, of the IN and OUT relations of the Parnas model, but their relations are similar to our relations D_IN and D_OUT and do not describe tolerances. A more fundamental difference between our approach and that of Thompson et al. is that the explicit application of information hiding is missing from their approach. Moreover, Thompson et al. appear to consider the externally visible behavior about hardware malfunctions (e.g., that the value of a monitored variable is unavailable) at the same time that the ideal system behavior is considered, and they do not associate timing constraints with the controlled variables.

7 Conclusions and Future Work

This paper has demonstrated how the SCR method may be used to specify the required behavior of the LCS in terms of four relations: NAT, $\widetilde{\text{REQ}}$, D_IN, and D_OUT. Just as for system requirements specifications, the available verification and validation features of the SCR toolset (e.g., consistency checking, simulation, and property checking using Salsa) were used to verify that the relations D_IN and D_OUT are well-formed and that they satisfy critical properties. The utility of analyzing the LCS specification for desired properties using either model checking or theorem proving is still an open question given the high overhead usually associated with these techniques. However, we are developing approaches that reduce this overhead by using automatic abstraction methods to limit state explosion in model checking [5, 9] and by using the automatic generation of invariants [15] and more automatic, more natural theorem proving methods [3, 4, 16] to facilitate the use of mechanical theorem provers.

Our application of the SCR simulator to the LCS specification proved to be especially valuable. Once the specification was entered into the SCR toolset, a user could run scenarios through the simulator to validate that the specification captures the intended behavior. However, the initial simulator has an interface generic to all SCR specifications. Developing a customized front-end for the LCS simulator based on Figures 16–17 would require only one or two days of work. The existence of a customized LCS simulator will allow us to demonstrate the system behavior captured in the requirements specification to customers, domain experts, and user interface design experts and thus provide an easy way to obtain both user and expert feedback.

Our new research is in automatic code generation from requirements specifications and mechanized analysis of timed specifications. The code generation facility that we are developing may be used eventually to automatically construct code for both the device-dependent and the device-independent software modules. What also remains is to verify the end-to-end system timing behavior, i.e., to verify that the timing constraints specified in the system requirements specification are feasible. Developing tool support for this is a current focus of our research.

Acknowledgments

This work is funded by the Office of Naval Research. Our extension of SCR to the specification and analysis of software requirements benefited from discussions at the Dagstuhl Seminar on Requirements Capture, Documentation and Validation in June 1999. We especially acknowledge the helpful comments of Dave Parnas during the seminar. We also acknowledge useful discussions with our colleague Jim Kirby, who suggested that we organize the requirements information as a database. Finally, we thank Myla Archer, Ralph Jeffords, Jim Kirby, and the anonymous referees for helpful comments on earlier drafts of this paper.

References

[1] The light control case study: Problem description. *Journal of Universal Computer Science, Special Issue on Requirements Engineering* (This Volume).

[2] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Lab., Wash., DC, 1992.

[3] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, pages 192–203. IEEE Computer Society Press, 1996.

[4] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers*, Eindhoven, Netherlands, July 1998. Eindhoven Univ. of Technology.

[5] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.

[6] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '2000)*, Berlin, March 2000.

[7] S. Easterbrook and J. Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 1997.

[8] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[9] 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 Softw. Eng.*, 24(11), November 1998.

[10] C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, CAN, 1998.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[12] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[13] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, January 1980.

[14] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[15] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, November 1998.

[16] J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proc. 15th Annual Computer Security Applications Conf. (ACSAC '99)*. IEEE Computer Society Press, December 1999.

[17] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd ACM Workshop on Formal Methods in Software Practice*, 1998.

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

[19] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[20] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Trans. Softw. Eng.*, SE-11(3):259–266, March 1985.

[21] J. M. Thompson, M.P.E. Heimdahl, and S. P. Miller. Specification-based prototyping for embedded systems. In *Proc. 7th ESEC/FSE*, September 1999.