Hardware/Software Co-Design and Co-Validation Using the SCR Method[*]

Ramesh Bharadwaj and Constance Heitmeyer

Naval Research Laboratory (Code 5546)

Washington, DC   20375

{ramesh,heitmeyer}@itd.nrl.navy.mil

Abstract

To date, the SCR (Software Cost Reduction) method has been used to specify system requirements. This paper extends the SCR method to hardware/software co-design and co-validation. Our approach consists of three steps. First, the SCR method is used to specify the required system behavior, i.e., the required relation between environmental quantities (called monitored quantities*) that the system monitors and environmental quantities (called* controlled quantities*) that the system controls. Next, the system designers specify the I/O devices required to compute estimates of the monitored quantities and to set values of the controlled quantities. Finally, the required software behavior is specified as three modules: a device-independent module, specifying how the (estimated) monitored quantities are to be used to compute estimates of the controlled quantities, and two device-dependent modules: an input device interface module, specifying how data from the input devices are to be used to compute estimates of the monitored quantities, and an output device interface module, specifying how the values of controlled variables are written to output devices. To illustrate the approach, we use SCR to specify a simple light control system.*

1   Introduction

The SCR (Software Cost Reduction) requirements method is a formal method based on tables for the specification and analysis of the black-box behavior of complex systems. Designed for use by engineers, the SCR method has been applied to a variety of practical systems, including avionics systems, telephone networks, and safety-critical components of nuclear power plants. Originally formulated by NRL researchers to document the requirements of the Operational Flight Program (OFP) of the US Navy's A-7 aircraft [1, 7, 8],

SCR has been used by a number of industrial organizations such as Grumann, Bell Laboratories, Ontario Hydro, and Lockheed, to specify the requirements of other practical systems. For example, in 1994, Lockheed used SCR to specify the C-130J OFP [4], which contains more than $250,000$ lines of Ada code.

SCR* is an integrated suite of tools supporting the SCR method [6]. The toolset includes a *specification editor* for creating and modifying a requirements specification, a *consistency checker* for checking the specification for application-independent properties (e.g., type correctness and unwanted nondeterminism), a *simulator* for symbolically executing the system based on the specification, a *model checker* [2] for analyzing the specification for critical application properties, and a *dependency graph browser* for displaying the dependencies among the variables in the specification. Our ongoing research also includes a new effort to automatically generate source code from SCR specifications. Currently, more than 70 organizations in the US, Canada, UK, and Germany, including industrial and government organizations and universities, are experimenting with the SCR* toolset.

The practical utility of the SCR method for developing requirements specifications has been demonstrated in four pilot projects. In one, NASA researchers used the SCR consistency checker to detect several errors in the requirements specification of the International Space Station [3]. In a second project, Rockwell-Collins engineers used the SCR tools to detect 24 errors, many of them serious, in the requirements specification of a flight guidance system [11]. In a third project, our group at NRL used the SCR tools to expose several errors, including a safety violation, in a contractor-produced specification of a US military system [5]. In a fourth project, our group used SCR* to specify the requirements of a cryptographic device (CD), verifying that the CD specification satis-

---

1. REPORT DATE **1999**	2. REPORT TYPE	3. DATES COVERED **00-00-1999 to 00-00-1999**
4. TITLE AND SUBTITLE **Hardware/Software Co-Design and Co-Validation Using the SCR Method**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5546,4555 Overlook Avenue, SW,Washington,DC,20375**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**6**	

fies seven security properties and demonstrating that it violates an eighth [10]. Especially noteworthy is that, in each of the latter two projects, using the SCR method to specify and analyze the required behavior of a moderately complex system required only one person-month of effort.

In applying the SCR method and tools, our emphasis so far has been on specifying, validating, and verifying *system* requirements. This paper describes how, in addition, the SCR method and tools can be applied to *software* requirements specification and also to hardware/software co-design and co-validation. To illustrate the extended method, we use a simplified specification of a light control system (LCS).

2 Background

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. Hence, every implementation that satisfies the specification must be acceptable to the customer. Further, it should be free of "implementation bias". That is, every implementation that is acceptable to the customer must satisfy the specification. The SCR method includes a set of guidelines for achieving these goals in practice. These guidelines are typically tailored to a specific problem domain, e.g., embedded control systems.

System Requirements Specification. The *System Requirements Specification* (SRS) describes the required black-box behavior of an entire system including its interfaces, hardware, software, peripheral devices, etc. To construct an SRS, the set of environmental quantities that are relevant to the system behavior must be identified, and each quantity must be represented by a mathematical variable. The set of environmental quantities consists of both *controlled quantities* – quantities in the environment that the system controls – and *monitored quantities* – quantities in the environment that can influence system behavior. The SRS should contain a detailed description of each quantity, including how it is measured, acceptable values, ranges, etc.

The desired system behavior is documented in the SRS by describing the relationship between the values of the monitored and controlled quantities. To describe this relationship, two relations, REQ and NAT, of the Parnas Four Variable Model (FVM) [12] are used. Relation NAT describes the constraints imposed on the environmental quantities by physical laws and the system environment; relation REQ describes additional constraints on the values of controlled quantities that the system must enforce. 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, we specify timing and accuracy requirements for each controlled variable.

System Design Specification. The *System Design Specification* (SDS) identifies and documents the characteristics of all resources that are available to the software to compute estimates of the monitored quantities and set values of the controlled quantities. These values are usually read from or written to hardware devices, such as sensors and actuators. They may also be obtained from or written to external computers or other software modules. The values in the system's hardware/software interfaces are denoted by mathematical variables. This set of system values is partitioned into *input data items* – values that the input devices provide to the software – and *output data items* – values the output devices obtain from the software.

SCR Notation. To specify the REQ and NAT relations in a practical and efficient manner, the SCR method uses four constructs – mode classes, terms, conditions, and events. The mode classes and terms capture historical information – the changes that have occurred in the values of monitored variables – and thus help make the specifications concise. A *mode class* may be viewed as a state machine, whose states are called *modes* and whose transitions are triggered by events. A *term* is a state variable defined on monitored variables, mode classes, or other terms. 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

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

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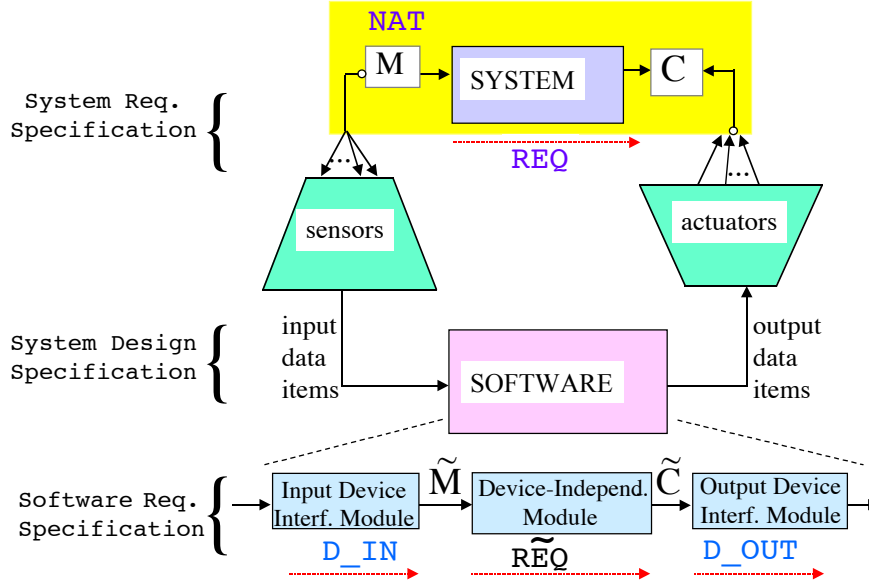
Figure 1: Relationship between the SRS, the SDS, and the SoRS.

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".

3 Software Requirements in SCR

To extend the SCR method, we use the System Requirements Specification and the System Design Specification as the foundation for the *Software Requirements Specification* (SoRS). For ease of change, we organize the Software Requirements Specification into three modules: two *device-dependent* modules and a *device-independent* module. The two device-dependent modules are the *input device interface module*, and the *output device interface module*. This organization allows us to easily change the software requirements specification, e.g., to introduce a new input or output device or to modify or add a system function, by changing a small part of one of its three modules. Figure 1 shows the relationship between the SRS, the SDS, and the SoRS and the decomposition of the SoRS into three modules[1].

An important observation we make in this paper is that the interfaces of the device-dependent modules constituting the SoRS are most conveniently specified in terms of the *environmental variables* , i.e., the monitored and controlled variables, already identified in the System Requirements Specification (SRS). This has two important consequences:

1. Because the SRS describes all the environmental quantities, the interfaces of the device-dependent modules of the SoRS are easily documented by providing appropriate references to the SRS.

2. Because the SRS describes the required relation REQ between the monitored and controlled variables, the black-box behavior of the device-independent module is already described (by REQ).

What remains is to document the black-box behavior of the device-dependent modules, which we specify using two relations D_IN and D_OUT. Relation D_IN specifies how *estimates* of the monitored quantities are computed in terms of the input data items. Relation D_OUT specifies how the *estimates* of the controlled quantities, specified by the device-independent module, are used to compute the output data items that drive the output devices. Relation $\widetilde{\text{REQ}}$ specifies the relation between estimates of the monitored quantities and the estimated values of the controlled quantities. For the purpose of this paper, we assume that $\widetilde{\text{REQ}}$ is isomorphic to relation REQ.

4 Specifying the LCS in SCR

Suppose there is a need for a new light control system (LCS) for a group of windowless offices. If a person occupies an office, the lights must go on. A facilities manager sets the *default ambient light level*. Users may assign an *alternate ambient light level*. To save energy, the system must turn off the lights if an office is unoccupied for more than *T3* minutes. However,

---

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

Mode Class = mcStatus		
Old Mode	Events	New Mode
unoccupied	@T(mOccupied)	occupied
occupied	@F(mOccupied)	temp_empty
temp_empty	@T(DUR(NOT mOccupied) > mT3) @T(mOccupied)	unoccupied occupied

Figure 2: Mode transition table for the Light Control System

Events	New tCurrentLL
@T(mcStatus = occupied) when (DUR(mcStatus ≠ occupied) ≥ mT1)	mDefaultLL
@C(mAssignedLL)	mAssignedLL

Figure 3: Event table for tCurrentLL of the Light Control System

if an office is reoccupied within *T1* minutes after it becomes empty, the previous ambient light level must be reestablished. If the office is reoccupied at or after *T1* minutes, the default ambient light level must be established.

4.1 System Requirements Specification.

To produce the system requirements specification, we first identify the environmental quantities, denoting each by a mathematical variable. To make the specification concise, next we define mode classes and terms. We use the prefix "m" to indicate the names of monitored variables, the prefix "t" for terms, the prefix "mc" for mode classes, and the prefix "c" for the names of controlled variables. The *type* of a variable indicates the range of values that may be assigned to that variable.

To specify the LCS requirements, we require five monitored variables – mOccupied, mDefaultLL, mAssignedLL, mT1, and mT3 – and one controlled variable cAmbientLL. The monitored variable mOccupied, which has type *boolean*, indicates whether an office is occupied. The monitored variables mDefaultLL and mAssignedLL, both of type *integer*, represent the two ambient light levels. We assume that light levels are measured in lux and that the ambient light in offices may vary from $0 - 10,000$ lux. The monitored variables mT1 and mT3 represent the lengths in minutes of the two time intervals, each in the range 0 to 30. The controlled variable cAmbientLL, of type *integer*, represents the ambient light level of an office.

Next, we specify the modes of operation of

Mode Class = mcStatus	
Modes	cAmbientLL
occupied, temp_empty	tCurrentLL
unoccupied	0

Figure 4: Condition table for cAmbientLL

the system. We identify a single mode class mcStatus which takes on values from the set {unoccupied, temp_empty, occupied}. Finally, we use an integer-valued term tCurrentLL to represent the current chosen light level. Note that the current light level may differ from the ambient light level in situations where the lights are off because an office is unoccupied.

The relation REQ is specified by three tables. Figure 2 shows a *mode transition table* which specifies new values for the mode class mcStatus. Figure 3 shows an *event table* which specifies new values for the term tCurrentLL. In Figures 2 and 3, the expression DUR(c) indicates the length of time (in minutes) that condidton c has been *true*. Figure 4 shows a *condition table* which defines the value of the controlled variable cAmbientLL. Relation NAT (not shown) defines the initial values of the monitored and controlled variables and constrains how each may change from one state to the next.

Events	New mOccupied
@T(iMD)	true
@F(iMD) when (not iDCC or (DUR(iDCC and iMD) < 1))	false

Figure 5: Event table for `mOccupied` of the Light Control System

4.2 System Design Specification

In the system design phase, we identify and specify the hardware devices that will be available to the software in the proposed system. The software can use these devices to compute the estimated values of all the monitored variables. 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).

Input Data Items. To enable the software to determine whether a room is occupied (i.e., to estimate the value of `mOccupied`), each office will be equipped with a passive infrared motion detector and a door closed contact. The motion detector is represented by a boolean variable `iMD`, which is *true* when there is movement in the range of the detector and *false* otherwise. Similarly, the door closed contact is represented by a boolean variable `iDCC`, which is *true* if the door is fully closed and *false* otherwise.

Each office will also be equipped with a control panel containing four displays, each with an associated pair of buttons. Each pair of buttons is used to control the value, shown in the display, of one of the four monitored quantities, `mDefaultLL`, `mAssignedLL`, `mT1`, and `mT3`. Associated with the four monitored quantities are four input data items, `iDefaultLL`, `iAssignedLL`, `iT1`, and `iT3`. To change the value contained in one of these displays, the user would depress the appropriate button. For example, if the operator wishes to change the default light level from 10 lux to 20 lux, he would depress the button labeled `Up` next to the display `iDefaultLL`, which initially contains 10, until the display contains the value 20.

Output Data Items. The only controlled quantity is the brightness of the cluster of dimmable lights represented by controlled variable `cAmbientLL`. The lights are controlled by two output data items: a *pulse line* represented by a boolean variable `oPL` which determines whether the lights are on or off, and a *dim value*

represented by an integer variable `oDV`. The dim value, which is between 0 and 100, sets the brightness level of the lights between 0% (off) and 100% (fully on).

4.3 Software Requirements Specification

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

Relation D_IN The relation D_IN specifies how the input data items `iMD`, `iDCC`, etc., are used to compute *estimates* of the monitored quantities, `mOccupied`, `mDefaultLL`, `mAssignedLL`, `mT1`, and `mT3`[2]. Computing `mDefaultLL`, `mAssignedLL`, `mT1`, and `mT3` from `iDefaultLL`, `iAssignedLL`, `iT1`, and `iT3` is trivial. In contrast, the estimated value of the monitored quantity `mOccupied` may be derived in different ways. One way is to define `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 actually occupied but there is insufficient motion to trigger the motion detector, `iMD` will be false thereby providing an inaccurate estimate of room occupancy. As a better estimate for the monitored variable `mOccupied`, we could use the output of the door contact `iDCC` in conjunction with the output of the motion detector `iMD`. This is specified by the event table of Figure 5. The value of `mOccupied` is set to *true* whenever the output of the motion detector (i.e., the input data item `iMD`) becomes *true*. When `iMD` becomes *false*, however, `mOccupied` remains *true* if the door has been fully closed *and* the value of `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.

---

[2]In our approach, estimates of the monitored quantities are denoted by `m̃Occupied`, etc. To improve readability, we have omitted the tildes.

Relation D_OUT The device-independent module of the SoRS specifies how estimates of the monitored quantities relate to estimates of the controlled quantity `cAmbientLL`. Relation D_OUT specifies how these estimates of `cAmbientLL` are used to compute the output data items `oPL` and `oDV`, corresponding to the pulse line and the dim value of a dimmable light cluster. The pulse line is defined as `oPL = (cAmbientLL ≠ 0)`; the definition of dim value is `oDV = (cAmbientLL/100)`.

5 Hardware/Software Co-Validation and Co-Synthesis

Above, we demonstrated how the SCR notation and toolset may be used to specify the relations REQ and NAT. We can also use the SCR notation and toolset to specify the relations D_IN and D_OUT. Just as for system requirements specifications, the available verification and validation features of the toolset (e.g., consistency checking, simulation, model checking, and theorem proving) may be used to verify that the relations D_IN and D_OUT are well-formed and that they satisfy critical properties. Also, the proposed code generation facility of the toolset may be used to automatically generate code for both the device-dependent and the device-independent modules. What remains to be done is to verify end-to-end system behavior, i.e., to verify that the timing and accuracy constraints specified in the system requirements specification are feasible. Developing tool support for this is a current focus of our research.

Acknowledgments

References

[1] 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.

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

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

[4] Stuart R. Faulk, John Brackett, Paul Ward, and James Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[5] 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.

[6] Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

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

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

[9] E. Kamsties et al. Case study: Dagstuhl seminar June 1999. Technical report, Schloss Dagstuhl, www.iese.fhg.de/Dagstuhl/seminar99241.html, June 1999.

[10] J. Kirby, M. Archer, and C. Heitmeyer. Applying formal methods to an information security device: an experience report. In *Proc. 4th IEEE Intern. Symp. on High Assurance Systems Eng. (HASE99)*, November 1999.

[11] Steve 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 (FMSP'98)*, 1998.

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

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