Published in Proceedings of the 1st ISEW Cleanroom Workshop (held with ICSE-18), Berlin, Germany, March 1996.

Integrating SCR Requirements into Cleanroom Software Engineering

Christian Bunse and Erik Kamsties

Fraunhofer Institute for Experimental Software Engineering

Technopark II

Sauerwiesen 6

D-67661 Kaiserslautern, Germany

{bunse, kamsties}@informatik.uni-kl.de

Abstract

This paper describes the combination of two industrially proven methods, SCR Requirements and Cleanroom Software Engineering, to form a seamless method for the formal specification and design of real-time systems. SCR documents functional and non-functional requirements such as timing and precision using a tabular notation of mathematical functions. Cleanroom supports the development of near-zero-defect software through formal methods and statistical quality control. The formalism primarily used in Cleanroom for specification and design is called Box Structure Method (BSM). We show how SCR can be integrated in BSM as a black-box-like description, and how the syntax and semantics of box structures can be extended to serve for real-time systems. Subsequently we describe how BSM's refinement and verification procedures have to be modified according to our previous definitions. The presentation is illustrated with a simplified example of a safety injection system for a reactor core.

Keywords: Real-Time Systems, Formal Specification, Cleanroom, Box Structure Method

1. Introduction

Today's competitive pressure among software organizations and society's increasing dependence on software have led to a strengthened focus on software quality. The community not longer tolerates that software is released in the belief that it isn't possible to develop zero-defect software. When considering embedded real-time systems, it must be recognized that not only the company itself but also thousands of people heavily depend on software with certifiable quality (e.g., imagine defects in safety critical applications like reactor shutdown or air traffic systems [7]).

To assure the high quality of embedded real-time software, various formal methods supporting software development have been published [12]. Cleanroom originally proposed by H.D. Mills [15] allows the development of near-zero-defect software with certifiable quality by using formal techniques, organizational principles, and statistical quality control. The application of Cleanroom has shown remarkable results. So NASA/SEL [6], IBM [11], University of Maryland [19], Ericsson [22], and various other organizations have recognized that not only can productivity and product quality be increased, but effort and failure rates can also be decreased by applying the principles espoused by Cleanroom.

In Cleanroom projects different weaknesses are encountered: Cleanroom claims to support the development of different system types, but applying box structures[1] to embedded real-time systems reveals a need for modification concerning the following areas [22].

- The very early phase of requirements analysis is not very well integrated with Cleanroom-based software development processes (e.g., box structures do not give enough support and do not provide sufficient formality in the analysis phase).
- The amount of functionality to be specified in real world projects is normally large. Therefore it is difficult to express and completely specify the corresponding black box.

Some solutions have been published to improve box structures to cope with these weaknesses. On the one hand it was tried to formalize box structures to support a more systematic way for requirements representation and system design [4,5]. Notations like Z or the Vienna Design Method have been introduced into box structures. These integrated approaches allow expressing of requirements with a formal notation and help the specifier develop more

1. Box structures are a method for systematic system development used in Cleanroom [14].

precise and concise documentation as well as a better basis for communication and verification [5]. Although Z or VDM allow for formalization they don't support needs encountered with the domain of real-time systems such as timing or precision. On the other hand SMO [2] tries to combine box structures with SDL. Using documents of different notations (e.g. SDL, Box Structures) in parallel results in extra effort for keeping consistency.

In this paper we present a technique which circumvents the weaknesses presented above. As mentioned in [4] tabular notation is particularly useful for specifying function rules to produce crisper, clearer, and more precise specifications. We integrate SCR styled requirements, a formal method for requirements representation [3,10], as a black box like description into box structures. SCR was developed for embedded real-time systems and has been used in various projects. The integration results in a development process which allow the development of near-zero-defect software for embedded real-time systems.

The remainder of this paper is organized as follows. Section 2 gives an overview about SCR, Cleanroom, the Box Structure Method (BSM), and a comparison of SCR and BSM. In section 3, we describe the theory and practice behind transforming SCR specifications into state boxes. Section 4 provides an example of the transformation. Finally we summarize the integration of SCR into Cleanroom and give directions for future work.

2. Review of SCR Requirements and Cleanroom/BSM

2.1 SCR Requirements

The basic techniques of the Software Cost Reduction (SCR) requirements method were developed by D. Parnas and co-workers at the US Naval Research Laboratory (NRL) [10]. The method has been extended to describe system and software requirements and to represent functional as well as non-functional requirements (i.e. timing, precision). A toolset called SCR* to support writing and checking SCR specifications was recently presented [8]. SCR consists of two main parts: a mathematical model of a computer system called the *Four Variable Model* [17, 21], and a tabular language of events and modes with formal semantics [20], we called in the sequel *A7 language*.

Four Variable Model. The Four Variable Model (see Figure 1[1]) describes requirements as a set of mathematical relations on four sets of quantities, called the *monitored*, *controlled*, *input* and *output* quantities. Monitored quantities influence the system behavior, whereas controlled quantities are manipulated by the system. The environmental quantities of interest for the system (those to be monitored and/or controlled) are expressed as mathematical functions over time (time-functions), since their values change with time.

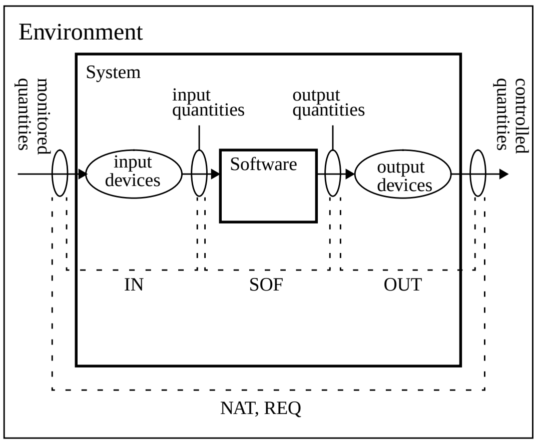

Figure 1: Four Variable Model

The system requirements are given by two relations, namely NAT (*nature*), and REQ (*required*) from the monitored to the controlled quantities. NAT defines constraints imposed on the system by its environment, it explicitly captures the limits of required behavior. Making constraints explicit helps define completeness of requirements. REQ defines the system's required external behavior.

The inputs and outputs that are associated with the system's input and output devices are resources available to the software. The mapping from the monitored quantities (system external) onto inputs (done by sensors) is specified by the relation IN. Similar, the relation OUT specifies a mapping from outputs to controlled quantities (done by actuators). These two relations, when considered in conjunction with NAT (if there are constraints) and REQ, form software requirements. The actual behavior of the software is later specified by relation SOF, supplied by the programmers, to enable verification against the requirements (NAT, REQ, IN, OUT).

A7 language. The A7 language has been proven to be a practical approach for specifying relations of the Four Variable Model in the context of real-time systems. The A7 language is comprised of three constructs, namely *conditions, events*, and *modes*. These are used together in a *tabular notation.*

A *condition* is a predicate on the environment (e.g. whether a button is being pressed). Conditions are boolean, although first-order predicate conditions that can be represented as a finite number of boolean conditions (e.g., value ranges) are also expressible.

1. This figure is taken from [9]

A change of a condition's value is an *event* and is denoted as

@T(*condition*)
or @F(*condition*)

The first form specifies the point in time when the value of *condition* changes from false to true. Similarly, the latter form specifies the time when *condition* becomes false. An event might also depend on the values of other conditions. A so-called *conditioned event* is denoted as

@T(*cond1*) WHEN [*cond2*]
or @F(*cond1*) WHEN [*cond2*]

whereby the first form specifies the point in time when *cond1* becomes true *while cond2* is also true. The latter form specifies the time when *cond1* becomes false *while cond2* is true. The semantics of SCR propose three slightly different definitions of conditioned events as discussed in [1]. More complex events can be created by nesting events or by using the boolean operators AND, OR.

A *mode* is a set of system states that share a common property. A mode captures the system's history. A *modeclass* is a set of modes. The system is in exactly one mode of each modeclass at all times. A *mode transition* occurs between modes in the same modeclass, caused by an event (event and mode transition happen at the same time). Modeclasses are independent from each other. A modeclass is a state machine, the modes are its states, the events form its input language.

A *tabular notation* is used for writing specifications because tables have been found to be easily read and understood by engineers. Three types of tables, namely condition table, event table, and mode transition table, are defined. Each table represents a mathematical function. A *condition table* defines a controlled quantity as a function of a mode and a condition, whereas an *event table* defines a controlled quantity as a function of a mode and an event. These tables define the output of the state machine. If a controlled quantity is purely a function of current inputs from monitored quantities, then the mode can be omitted. A *mode transition table* defines a mode as a function of the previous mode and an event; by this table a modeclass and its modes are specified. Examples are given in Tables 1, 2, and 3.

A set of standard functions and relations is provided by van Schouwen [20]. Among these there are functions to access time-functions of quantities:

- Next(*event*), Last(*event*) refers to the next (last) occurrence of this event
- Drtn(*condition*) refers to the duration the condition holds

Safety injection example. To illustrate the A7 language, we consider a control system for a safety injection of a nuclear plant [3,9]:

"The system uses a sensor to monitor water pressure and adds coolant to the reactor core when the pressure falls below some threshold. The system operator blocks safety injection by turning on a "Block" switch and resets the system after blockage by turning on a "Reset" switch."

Figure 1 shows the system in the view of the Four Variable Model.

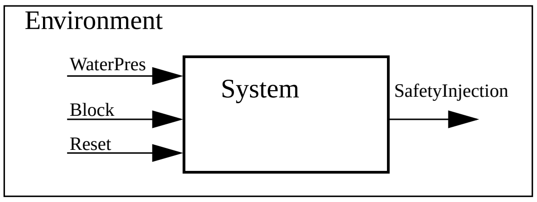

Figure 2: Safety Injection System

The switches and the sensor for water pressure are modeled by three monitored quantities WaterPres, Block and Reset. The safety injection is modeled by the controlled quantity SafetyInjection. The resulting SCR tables are shown in Tables 1, 2, and 3; SCR's bracketing symbols for identifiers are omitted.

Old Mode	Event	New Mode
TooLow	@T(WaterPres ≥ Low)	Permitted
Permitted	@T(WaterPres ≥ Permit)	High
Permitted	@T(WaterPres < Low)	TooLow
High	@T(WaterPres < Permit)	Permitted

Table 1: Mode Transition Table for *Pressure*

Mode	Events	
High	False	@T(Inmode)
TooLow OR Permitted	@T(Block = On) WHEN Reset = Off	@T(Inmode) OR @T(Reset = On)
Overridden =	True	False

Table 2: Event Table for *Overridden*

Mode	Conditions	
High or Permitted	True	False
TooLow	Overridden	NOT Overridden
SafetyInjection =	Off	On

Table 3: Condition Table for *SafetyInjection*

The Tables 1, 2, and 3 are part of the REQ relation. The modeclass Pressure (Table 1) divides the continuous value of water pressure into three classes TooLow, Permitted, and High. Overridden (Table 2) indicates whether the safety injection is blocked or not; it is a both monitored and controlled quantity that is established to make the specification more readable. The condition "Inmode" in the event @T(Inmode) is triggered if a mode transition into the mode of that row happens (Table 2); for example @T(Inmode) in the second row of Table 2 means: "if the mode High is entered, then Overridden becomes false."

2.2 Cleanroom and the Box Structure Method

Cleanroom is a systematic process for developing near-zero-defect software and was developed by H.D. Mills (IBM, USA) [4,15]. In using Cleanroom the software development organization makes extensive use of formal methods and engineering discipline across all life cycle phases, and puts the process under statistical quality control. Key features of Cleanroom are:

- Organizational aspects (e.g. software development by small teams instead of individual heroes).
- Incremental development. The system specification is analyzed to identify functions which can be developed in increments. These increments must fulfill various conditions and can be merged to form the complete system. Increments are used to allow system development in an intellectually controllable manner.
- Rigorous specification and design by usage of box structures and correctness verification.
- Usage testing. In Cleanroom the specification doesn't contain only functional requirements. The future system usage is additionally modeled by identifying externally visible system states, their interdependencies, and probabilities of state transitions. This model is used later to develop test cases and to perform certification.
- Reliability Certification by applying statistical models onto test results.

The Box Structure Method (BSM), originally developed by Harlan D. Mills [14,16], provides a rigorous framework for specification and design [2]. BSM relies on three basic structures called black box (BB), state box (SB) and clear box (CB) which can be nested in a hierachical system structure. To develop a system by using BSM, Mills defined a twelve-step algorithm [14] which allows building a hierarchy of box structures in a stepwise refinement manner. The main steps of the algorithm are:

- Define black box
- Define state box
- Define clear box

The system description is expanded and refined with each of these steps.

The black box describes a system by modeling its externally visible behavior. All details about internal structure and operations are omitted. The modeled behavior is stated in terms of stimuli (S) and responses (R). A stimulus is an input to the system (e.g. a command with parameters). An response is an externally visible reaction of the system (e.g. printing on a screen). Each response of the system is determined by the system's stimuli history (S^*), which contains all previously received stimuli. Overall a function f: $S^* \longrightarrow R$ is described.

The previously defined black box can be refined into a state box. A state box describes a system's behavior by using an internal state (T), defined by analyzing required stimulus history and responses. The internal state is used to encapsulate parts of the stimulus history. Each response is determined by stimulus and state histories. Overall a function g: $(S \times T)^* \longrightarrow R \times T$ is described.

Finally the state box is refined into a clear box, which provides a procedural description of the system by presenting more details about how stimuli are processed to determine responses. The black box function g of the state box is replaced by concurrent or sequential usage of new black boxes. These boxes can be expanded at the next level into new state and clear boxes.

The refinement of the three box types produces a hierarchy of box structures. This hierarchy provides an effective means for management control in system development. The black box subsystems become well defined and independent modules in the overall system by further refinements [22].

Different notations have been published to describe box structures. Mills [16] introduced the Box Design Language (BDL) as a "language for recording, communicating, and analyzing box structures." The BDL organizes black boxes in three parts (stimuli, response, and transition) and state boxes in four parts (stimuli, responses, state, and machine). A more formal syntax for describing box structures can be found in [13].

2.3 Comparison of SCR and Cleanroom/BSM

The requirements specification is the concern of SCR and BSM; it is the focus of SCR and the starting point of BSM (the top-level black box of the hierarchy is taken as a requirements specification). The REQ relation in SCR's Four Variable Model describes the system behavior. REQ can be interpreted as a black box function from the BSM point of view, because time-functions of monitored quantities (i.e. stimuli) are mapped onto time-functions of controlled quantities (i.e. responses) without consulting an internal state. Therefore, SCR's Four Variable Model and BSM's top-level black box have the same abstraction level.

Due to the fact that SCR was originally developed for real-time systems and BSM for information systems, we found differences shown in Table 4, but no direct conflicts.

SCR’s Four Variable Model	Cleanroom/BSM
Inputs and outputs to the system are related to environmental quantities which represent physical devices.	Inputs and outputs are considered, but not physical devices.
Inputs and outputs to the system are functions of time. Timing can be described explicitly by mathematical functions.	Time is considered implicitly. A response is an immediate reaction to a stimulus. The stimulus history contains the stimuli ordered by the time of reception.
The contents of the specification comprise system requirements (REQ), software requirements (derived out of NAT, REQ, IN, OUT), hardware characteristics (IN, OUT), and constraints of the environment (NAT).	Functional requirements are considered in BSM exclusively, no differences between system and software are made. The top-level black box and the usage model forms the specification in Cleanroom.

Table 4: Comparison of the Four Variable Model and Cleanroom/BSM

SCR’s A7 language implements the time-function concept of the Four Variable Model in a restricted way. It deals with events and reactions at the current time. Therefore it seems suitable for use in BSM where time is only considered implicitly. The A7 language introduces new constructs like events and modes; we found a pragmatic mapping onto BSM/BDL constructs as listed below:

- Event → Stimulus
  We interpret an event (a point in time at which a certain condition changes its value) on a monitored quantity as a stimulus.
- All changes to quantities due to an event → Response
  An event triggers reactions, i.e., changes to the value of one or more quantities. We interpret the set of all changes to external visible (i.e. not established) quantities due to an event as a response.
- Using time-functions → Stimulus history
  The A7 language provides the functions Last(), Next(), and Drtn() for accessing the event history stored in the time-functions of the environmental quantities. Desired access functions can be specified. We interpret the use of these functions as a definition of stimulus histories.
- Mode, environmental quantity → State
  We interpret modes as states. Further, every environmental quantity used in a condition (except the conditions used in an event, i.e. @T(*condition*), @F(*condition*)) can be seen as a state. This applies to quantities used in the WHEN-clause of conditioned events, quantities used in the condition columns of condition tables, etc.
- Event table/mode transition tables → state box transition functions
  Event table/mode transition tables and BDL’s transition functions are semantically equivalent. The difference is the syntax. An event table shows for each value of a controlled quantity (part of response) the events (stimuli) and modes (states) which can lead to this value. A state box transition function shows for a stimulus the response and next state depending on the (abstract) state (compare Tables 1-3 and Figure 4 for example).

In summary, the constructs of SCR’s A7 language match the constructs of both BSM’s black box (stimulus history, response) and state box (stimulus, state, response) description. The level of abstraction found in an A7 specification depends on the use of the language. If modes are omitted and only time-function are accessed, it is a black box description. If modes are used, it is more of a state box description but with one major difference: data is only specified as far as externally visible as environmental quantities; the specification of internal data isn’t a concern of SCR.

The properties of SCR such as precise notation, comprehensive documentation of environmental constraints, hardware characteristics, etc. form an enhancement of Cleanroom/BSM in the context of real time systems.

3. Integration of SCR Requirements into Cleanroom/BSM

The analysis in section 2.3 implies *integrating* SCR in BSM rather than using SCR as a front-end because SCR’s abstraction level is between black- and state box of BSM. We decided to use SCR as a top-level description because SCR has been proven to be readable for customers and developers [8]. The main steps of our modified BSM algorithm are:

- Write SCR specification using A7 language
- Define state box
- Define clear box

This approach of integration resulted in the following modifications to BSM, which are discussed in the remainder of this section:

- Syntax and semantics for stimuli, responses, states, and transitions are extended to guarantee traceability in terms of constructs between SCR and the top-level state box.
- The algorithm for state box derivation is modified so that a SCR specification can be used.
- A method is defined to verify the state box against the SCR specification.

Finally, the implications on the Cleanroom process, especially usage modeling and incremental planning, are analyzed.

3.1 Syntactic and semantic extension of the top-level state box

We have developed an extended notation for state boxes which is based on the concepts of BSM/BDL and borrows the event construct and the tabular notation from SCR. This notation guarantees traceability from SCR specification to the top-level state box and keeps documentation consistent.

The syntax of a stimulus is equal to those of an event in SCR without the WHEN-clause.

Definition 1: Stimulus

Stimuli are noted by:
S_i: <stimulus> with $i \in |N_0$
The BNF[1] for <stimulus>:

<stimulus>	→	@T(<predicate>) | @F(<predicate>)
<predicate>	→	<monitored-quantity> <relational-operator> <value>
<monitored-quantity>	→	*name of a monitored quantity*
<relational-operator>	→	*one of* { =, ≠, <,>, ≤, ≥}
<value>	→	*as declared in the SCR specification*

As mentioned in section 2.3, all changes to not-established quantities due to an event form a response. Therefore we make a distinction between *atomic responses* and *responses*. An atomic response is a change of *one* not-established quantity caused by an event. A response consists of *all* atomic responses caused by a single event. Atomic responses are denoted by assignments.

1. Non-terminal symbols are "→", "|", and identifiers written in "<brackets>". Informal statements are printed in *italic*.

Definition 2: Atomic response

Atomic responses are noted by:
R_j:<atomic-response> with $j \in |N$
The BNF for <atomic-response>:

<atomic-response>	→	<controlled-quantity> := <value>
<controlled-quantity>	→	*name of a controlled quantity*
<value>	→	*as declared in the SCR specification*

Since we interpret modes and environmental quantities used in certain conditions as states, we introduce *state variables* to represent these modes and quantities. A state variable is declared as an enumeration of its values.

Definition 3: State variable declaration

State variables are declared by:
E_k: <state-definition> with $k \in |N$

The BNF for <state-definition>:

<state-definition>	→	<name> ::= {<value-list>}
<value-list>	→	<value-list>,<value> | <value>
<name>	→	*#name of the modeclass#* | *#name of the environmental quantity#*
<value>	→	*name of mode defined for this modeclass* | *name of a value of this quantity*

The state T of the state box is the set of state variables. The concrete value of a state variable can be questioned or assigned.

Definition 4: State comparison and state assignment

The BNF for <state-comparison> *and* <state-assignment>:

<state-comparison>	→	<name> <relational-operator> <value>
<state-assignment>	→	<name> := <value>
<relational-operator>	→	*one of* { =, ≠, <, >, ≤, ≥ }
<name>	→	*name of state variable*
<value>	→	*name of a value of this state variable*

Since the experiences with tables for documentation are promising [8,10] we use modified *selector tables* from SCR originally introduced by van Schouwen [20] to describe the state box transition functions.

Definition 5: State box transition function

Let *s* be a stimulus, *T* be the state, $S_1, ..., S_n$ be boolean expressions from state comparisons, $R_{1,1}, ..., R_{n,m}$ be atomic responses and $T_{1,1}, ..., T_{n,m}$ state assignments. Then the tabular notation

Stimulus *s*

State	Response	Next State
S_1	$R_{1,1}, ..., R_{1,m}$	$T_{1,1}, ..., T_{1,m}$
...	...	...
S_n	$R_{n,1}, ..., R_{n,m}$	$T_{n,1}, ..., T_{n,m}$

defines a state box transition function Y that is part of the state box function:

$$Y(s,T) = \begin{cases} (\{R_{1,1}, ..., R_{1,m}\}, \{T_{1,1}, ..., T_{1,m}\}) \text{ if } S_1 \in T \\ \vdots \\ (\{R_{n,1}, ..., R_{n,m}\}, \{T_{n,1}, ..., T_{n,m}\}) \text{ if } S_n \in T \end{cases}$$

A state box description consists of a list of stimuli, a list of responses, a list of state variable declarations, and a set of state box transition functions noted by tables. An example of this state box notation is given in chapter 4.

We enrich the semantics of stimulus and response by naming the quantities (associated with physical devices) and their (abstract) values. Instead of an informal specified stimulus "request to start continuous SOS broadcast", we can write more precisely "@T(Receiver = transmit-SOS)". Note that the Four Variable Model and its time-functions is not referred in our state box notation, despite of using identifiers of environmental quantities. The next section shows how a transformation of a SCR specification into a state box with the syntax given above can be done.

3.2 Derivation of the top-level state box

With the extended syntax and semantics of top-level state boxes we can explain how to extract them from SCR specifications. To do so we have defined a seven step algorithm (see Figure 3). This algorithm replaces the activities proposed by Mills [14] to derive a system state box from a system black box. The algorithm itself uses SCR's REQ relation for state box derivation. The relations IN and OUT are used in later steps of BSM. The contents of these relations are too specific for top-level boxes but have to be defined in lower levels of design. The algorithm depends on four major steps. These are:

1. Find all stimuli and atomic responses
2. Find the state box state
3. Define the state box transition function
4. Verify the state box against the SCR specification

1. **Understand** the SCR specification of the system.
2. **Find all stimuli**.
   2.1 Define @T(SystemInit) as stimulus for system initialization.
   2.2 Each event on a monitored quantity that doesn't use the WHEN-clause is a stimulus.
   2.3 Each event on a monitored quantity that uses the WHEN-clause is a stimulus. *Note: only the first event part without WHEN-clause is the stimulus.*
3. **Find all atomic responses**.
   3.1 The range of all event/condition table functions used for not established quantities form the set of atomic responses.
4. **Define the state of the state box**
   4.1 Take modeclasses as state variables and the single modes as values.
   4.2 Take each quantity mentioned in a condition (except those used in an event, i.e. @T(condition), @F(condition)) as state variable. *Note: it may be necessary to invent new stimuli to guarantee that those state variables are manipulated correctly.*
5. **Define the transition functions of the state box**

   For all stimuli:

   5.1 Get the response and next state caused by a stimulus
   - Collect all atomic responses caused by the actual stimulus.
   - If the event corresponding to the actual stimulus is used within a mode transition table, insert a state assignment.
   - If the stimulus was invented within step 4.2, an adequate state assignment has to be inserted
   - Each change in value of a quantity mentioned in a condition (step 4.2) is modeled as an assignment to the corresponding state variable.
   - Reactions correlated to @T(Inmode) events are used as atomic responses for the current transition function, if the state variable is set to a value correspondend to the mode used for @T(Inmode) by the actual stimulus.

   5.2 Get preconditions on state required for actual stimulus
   - Expand modes used in condition and event tables to boolean expressions about corresponding state variables.
   - Stimuli gathered within step 2.3 use the condition of the WHEN-clause as state comparison.

   5.3 Describe transition function by using the table notation.
6. **Optimize state box Specification.**
   6.1 Remove all state variables which are only set and never used within the state box. This means removement of assignments to them either.
7. **Verify state box** by answering a set of correctness questions.

Figure 3: Algorithm for transforming an SCR document into a state box

We have described that events on monitored quantities can be mapped on stimuli (see section 2.3). But there is one exception, the event @T(Inmode) (see section 2.1) has to be treated when defining transition functions. Considering conditioned events (e.g. @T(Block = On) WHEN Reset = Off) special rules must be applied on the WHEN-clause; the first event part (e.g. @T(Block = On) is used as stimuli. Furthermore we insert a special stimulus called @T(SystemInit), since the system must be in a determined state after invocation.

In Section 3.1 we defined atomic responses as event-caused changes of non-established quantities. This definition allows to identify the range of these quantities condition and event table functions as the set of atomic responses.

The set of state variables of the state box can be found by using the modeclasses and all quantities contained in conditions (except those used in an event, i.e. @T(*condition*), @F(*condition*)). For example, the quantity used in the WHEN-clause of conditioned events has to be modeled as a state variable (e.g., insert a new state variable for Reset). But state variables are independent from monitored quantities which causes a need for inserting new stimuli to guarantee for correct handling of the newly inserted state variable.

Preconditions for defining transition functions are fulfilled, which can be modeled by using tables as defined in section 3.1 This can be done by applying algorithm rules 5.1, 5.2, and 5.3. It must be guaranteed that @T(Inmode) events which initiate reactions on entering a mode are considered in transition functions responsible for entering a specific state.

Verification of the transformation can't be done formally. Instead the belief in the correctness of transformation must be built by answering correctness questions which show that the SCR specification is equal to the newly defined state box. An informal approach naturally isn't as good as a formal one because correctness shouldn't depend on humans. But performing a formal verification means building a SCR specification out of the state box and comparing it against the original [14]. Major modifications, as needed here, make it difficult or impossible to do such a verification. The following set of questions is applicable for informal verification:

1. Do all state variables have a determined value after system invocation?
2. Are all quantities concerned?
3. Are all relevant events modeled as stimuli?
4. Are the ranges of event and condition table functions modeled as set of atomic responses?
5. Is the functionality of each stimuli complete?
6. Are all modeclasses modeled as state variables?
7. Are all states of state variables set?
8. Are all reactions on @T(Inmode) events modeled correctly?
9. Are all simplifications used correctly?

By answering each of these questions with "Yes" we believe that the algorithm was performed correctly and that the state box expresses all requirements as mentioned in the SCR document.

After performing the algorithm given above, for deriving the top-level state box, system development is continued by applying the BSM algorithm as defined by Mills [14].

The presented state box notation and algorithm are limited to discrete-valued quantities but is seems easy to cover continuous-valued quantities also. Further limitations are imposed by the absence of time-functions in the state box, thus requirements on time (expressed in SCR specifications by the functions Last(), Next(), Drtn()) cannot be propagated into the state box.

3.3 Further implications for the Cleanroom Process

The use of SCR specifications as a base that can be transformed into an extended state box is a prerequisite for a successful integration of SCR into Cleanroom. We have defined a technique to specify, design, and implement embedded real-time systems in a systematic manner. Contributions to the results of Cleanroom are also based on techniques like statistical testing and incremental development [4]. To show that our approach is usable for real projects, we have to show that the SCR specification can be used on all activities which require black box information. These activities are usage modeling and incremental development planning.

A "Usage Model" can be defined by using the SCR specification instead of the system black box. The externally visible states and their interdependencies are modeled explicitly in SCR by modes and mode transitions. These definitions can be used for usage modeling. The needed probabilities have to be identified manually.

Incremental planning can be done using SCR because needed system functionality is modeled explicitly by the table types. Furthermore increments can be found by identifying independent sets of monitored and controlled quantities. Decisions about assigning functions to increments and choosing what sequence of increments to develop have to be done by using experience and domain knowledge.

The facts stated above lead us to the assumption that our approach allows the development of embedded real-time software according to the principles of Cleanroom Software Engineering.

3.4 Preliminary Validation

To show that the presented technique can be used in practice, we show that there are no inconsistencies in syntax or semantic. Usage of a consistent notation and slight modification of the box structure algorithm allows to support traceability and changeability. It supports verification of each development step.

Uniform notation in SCR and box structures allow to trace requirements from specification to code by using the box structure hierarchy. Traceability between SCR and box structures is one supposition for propagating changes. Changes in the requirements document itself can be propagated easily to the derived state box by reapplying the defined algorithm on affected parts. The algorithm allows updating the existing state box and verifying that the performed changes are correct. For lower levels of box structure hierarchy the BSM allows propagating changes by following the rules of the box structure algorithm.

Verification of development steps is guaranteed by using BSM without changes for latter development steps (verification rules by Mills [14]) and by using the defined set of correctness questions to check the state box against the SCR specification. These combined approaches allow verification of each development step from specification to code and establishing intellectual control upon the process.

4. Example and Experiences

Safety injection example. To illustrate our approach we use the safety injection example introduced in section 2.1 By applying our algorithm to the SCR tables a state box description of the "Safety Injection System" is developed. There are seven stimuli essential for defining a state box. These are:

S_0: @T(SystemInit)
S_1: @T(WaterPres ≥ Low)
S_2: @T(WaterPres ≥ Permit)
S_3: @T(WaterPres < Low)
S_4: @T(WaterPres < Permit)
S_5: @T(Block = On)
S_6: @T(Reset = On)
S_7: @T(Reset = Off)

The stimulus S_0, although not mentioned in the SCR example, is required for defining an initial state. Stimuli S_7 has to be considered as special case because the necessity of its existence was recognized when defining state. At that time a state variable for resets had to be defined, but within the SCR specification the state is set to "On" and never to "Off." Therefore a new stimulus was defined which allows setting the "Reset" state variable back to "Off".

After completing step two of the algorithm the identification of atomic responses is required. Atomic responses are:

R_1: SafetyInjection := Off
R_2: SafetyInjection := On

By having stimuli and atomic responses defined we develop the state of the state box. Careful analysis of SCR tables leads to three state variables needed for a state box description. These are:

E_1: #Pressure# ::= {TooLow, Permitted, High}
E_2: #Overridden# ::= {True, False}
E_3: #Reset# ::= {On, Off}

The definition of state variable E_3 causes some complex actions to take place. Because the Reset quantity is used in a conditioned event, a "Reset" state variable has to be defined. To guarantee the correct manipulation of this state variable, a new stimulus (S_7) has to be invented.

Having stimuli, atomic responses, and state variables defined, we can now describe transition functions caused by a specific stimuli. Figure 4 shows the resulting tables. S_3's function was found by using its "Old Mode" as precondition (rule 5.2), by assigning its "New Mode" to the corresponding state (rule 5.1), by identifying an assignment to "#Overridden#" as result of the previous state assignment (rule 5.1), and by identifying the assignment "SafetyInjection = On" as result of the assignment to "#Overridden#". Stimulus S_7 only performs a state transition and produces no externally visible response.

Within step six of the algorithm the state box description is optimized. Careful analysis of the presented example shows, that the state variable #Overridden# is only updated but never used. Therefore the state variable itself and all related assignments can be removed. The optimization proves the statement of section 2.1, "Modeclass Overridden is established to make the specification more readable." In step seven of the algorithm all questions can be answered with yes. This leads to the assumption that the modeled state box is correct.

Experiences. In addition to the previous example, our approach was used on other examples like the buoy problem stated by Mills [14] and a parking garage system [18] with remarkable results (e.g., the parking garage system shows no failures while testing code statistically). We have had different experiences. The use of a uniform notation supports a comprehensive understanding of the complete documentation and verification of development steps. Furthermore, by providing an algorithm which doesn't require much effort to be executed and which allows for automation, our approach seems to be easily adaptable to real world projects.

While performing the algorithm different times we have learned that the abstraction levels of the SCR specification and the refined state box didn't differ as much as a black box and a state box in BSM. This is due to the fact that an SCR specification contains modes which can be mapped directly onto states, while black boxes, as defined by Mills [14], didn't contain any information on system states with the exception of stimulus history. Nevertheless the proposed transformation is useful because it provides different

Stimulus S_0: @T(SystemInit)

State	Response	Next State
True	SafetyInjection := Off	#Pressure# := Permitted #Overridden# := False #Reset# := Off

Stimulus S_1: @T(WaterPres ≥ Low)

State	Response	Next State
#Pressure# = TooLow	SafetyInjection := Off	#Pressure# := Permitted #Overridden# := False

Stimulus S_2: @T(Water Pres ≥ Permit)

State	Response	Next State
#Pressure# = Permitted	SafetyInjection := Off	#Pressure# := High #Overridden# := False

Stimulus S_3: @T(WaterPres < Low)

State	Response	Next State
#Pressure# = Permitted	SafetyInjection := On	#Pressure# := TooLow #Overridden# := False

Stimulus S_4: @T(WaterPres < Permit)

State	Response	Next State
#Pressure# = High	SafetyInjection := Off	#Pressure# := Permitted #Overridden# := False

Stimulus S_5: @T(Block = On)

State	Response	Next State
(#Pressure# = TooLow OR #Pressure# = Permitted) AND #Reset# = Off	SafetyInjection := Off	#Overridden#:= True

Stimulus S_6: @T(Reset = On)

State	Response	Next State
#Pressure# = Permitted		#Overridden# := False
#Pressure# = TooLow	SafetyInjection := On	#Overridden# := False #Reset# := On

Stimulus S_7: @T(Reset = Off)

State	Response	Next State
True		#Reset# := Off

Figure 4. State Box Function

views (*SCR view* and *state box view*) onto the system. The *SCR view* allows checking whether the device control is complete (e.g. a missing event). The *state box view* allows checking whether all stimuli are considered and whether a response to a specific stimuli is complete. By considering views we are able to detect inconsistencies and find requirements that have to be modeled more carefully.

5. Conclusions

We presented a technique for integrating SCR requirements into Cleanroom Software Engineering by combining SCR and the Box Structure Method. We have explored differences and commonalities of both methods, and have recognized that SCR concepts can be mapped onto BSM concepts and that the underlying models of both methods match. Based on these findings, we have defined how the SCR specification can be used as starting point for box structures and how to transform it into a state box. This definition has led to syntactical and semantical extensions of top-level state boxes, to an algorithm for extracting such boxes, and to a set of correctness questions for verifying the state box against the SCR specification. We have shown that our technique has no further negative implications for BSM (e.g. traceability, verifiability and changeability) or for the Cleanroom process (e.g. usage modeling).

Overall our technique provides sufficient formality for requirements representation in the analysis phase by using an exact and understandable notation. The method helps to express and specify completely specify requirements for large systems. These advantages can be propagated into box structures and developed further. By using BSM and other Cleanroom techniques it is possible to develop near-zero-defect software for embedded real-time systems.

Currently we are evaluating the work presented here in the context of student experiments and industrial case studies. We are trying thereby to validate our technique and to prove that it is applicable to real world projects.

Further work concentrates on extending the syntax and semantics of box structures to allow for mapping different SCR notational elements, that have not yet been considered. Related to this work are questions on how requirements on time can be modeled explicitly to assure their formal representation in high levels of specification and design.

Acknowledgments

The authors wish to thank Dieter Rombach for his valuable comments and for fruitful discussions throughout the research. We also would like to acknowledge Constance Heitmeyer, Stuart Faulk, and Ralph Jeffords (NRL, USA) for their help on SCR, Christopher Lott for reviewing this paper, and all the members of the Software Engineering Research Group at the University of Kaiserslautern.

References

[1] Joanne M. Atlee and John Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, Jan. 1993.

[2] H. Cosmo, A. Sixtensson, and E. Johansson. SMO - A stepwise refinement and verification method for software systems. In *Proceedings of SDL forum*, Glasgow, October 1991.

[3] P.-J. Courtois and D. L. Parnas. Documentation for Safety Critical Software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323. IEEE Computer Society Press, May 1993.

[4] M. Dyer. *The Cleanroom Approach to Quality Software Development*. Wiley and Sons, 1992.

[5] D.T. Fetzer and J.H. Poore. Using Box Structures with the Z Notation. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, January 1992.

[6] S. Green, A. Kouchakdjian, V. Basili, and D. Weidow. The Cleanroom Case Study in the SEL: Project Description and early Analysis. Technical Report SEL-90-002, NASA-SEL, March 1990.

[7] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and Consistency Analysis of State-Based requirements. In *Proceedings of the 17th International Conference on Software Engineering*, 1995.

[8] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. Scr*: A Toolset for Specifying and Analyzing Requirements. In *Proceedings of the 10th Annual IEEE Conference on Computer Assurance*, Gaithersburg, MD, USA, June 1995.

[9] C. Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency Checking of SCR-Style Requirements Specifications. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56–63, York, England, March 1995.

[10] Kathryn L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and their Application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[11] Richard C. Linger. Cleanroom Software Engineering for Zero-Defect software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 2–13, Los Alamitos California, May 1993. IEEE CS Press.

[12] Shaoying Liu, Victoria Stavridou, and Bruno Dutertre. The Practice of Formal Methods in Safety-Critical Systems. *Journal of Systems and Software*, 28(1):77–87, January 1995.

[13] Hailong Mao. *The Box-structure Development Method*. PhD thesis, University of Tennessee, Knoxville, USA, December 1993.

[14] H.D. Mills. Stepwise Refinement and Verification in Box-structured systems. *IEEE Computer*, pages 23–36, June 1988.

[15] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom Software Engineering. *IEEE Software*, pages 19–24, September 1987.

[16] H.D. Mills, R.C. Linger, and A.R. Hevner. *Principles of Information System Analysis and Design*. Academic Press, Inc., 1986.

[17] David L. Parnas and Jan Madey. Functional Documentation for Computer Systems Engineering (version 2). CRL Report No. 237, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, September 1991.

[18] Wolfram Petsch and Klaus Schmid. Projekt Parkhaussteuerung; Systemdokumentation. Technical Report, University of Kaiserslautern, 1993.

[19] R.W. Selby, V.R. Basili, and F.T. Baker. Cleanroom Software Development: An Empirical Evaluation. *IEEE Transactions on Software Engineering*, pages 1027 – 1036, September 1987.

[20] A. John van Schouwen. The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems. CRL Report No. 242, McMaster University, CRL, Hamilton, Ontario, Canada, February 1993.

[21] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of Requirements for Computer Systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, California, USA, January 1993.

[22] C. Wohlin, A. Sixtensson, P. Runesson, and E. Johansson. Cleanroom Software Engineering Applied to Telecommunications. In *Proceedings of Nordic Seminar on Dependable Computing Systems*, August 1992.