Avionics Systems Requirements: A Comparison of RSML and SCR[1]

Bruno Dutertre; CSL, SRI International; Menlo Park, California

Victoria Stavridou; CSL, SRI International; Menlo Park, California

Keywords: Requirements Specification, Formal Analysis, Avionics, Control Systems

Abstract

This paper discusses and compares RSML and SCR, two approaches for the rigorous specification of control systems requirements. We give examples of RSML and SCR specifications for parts of a generic store management system. We discuss the merits and limitations of the two methods and examine how safety verification can be supported in both cases.

Introduction

RSML (refs. 1-2) and SCR (refs. 4-6) are two formalisms for the specification and analysis of control systems requirements. Both have been successfully applied to complex avionics systems. RSML was developed to specify the TCAS II aircraft collision avoidance system (ref. 1). SCR was introduced to specify the software requirements of the A7 flight software (ref. 4) and has since been extended to cover system as well as software requirements (refs. 7-8). The two notations are amenable to various forms of static and dynamic analysis. In particular, both SCR and RSML provide techniques for checking the consistency and, to some extent, the completeness of specifications. Furthermore, the two approaches rely on a similar state-based model and complex properties of systems can be analysed using verification techniques such as model checking or theorem proving.

The objective of this paper is to compare SCR and RSML using an avionics case study. The work was done in the context of a software/hardware codesign project, in an effort to improve requirements traceability, in collaboration with GEC Marconi Avionics. The example is based on a generic store management system (SMS) defined by GEC Marconi Avionics Ltd. This system controls the arming, fusing, firing, and release of weapons and other equipment stored under the fuselage of a military aircraft. The SMS must ensure that no weapon is armed or released accidentally but must also provide an emergency jettison capability whereby all stores are released. The system must also be fault-tolerant: a single failure shall not cause accidental weapon release and must not disable the emergency jettison capability.

It is known that the quality of system requirements can have major safety implications; errors in software requirements have been implicated in several accidents (ref. 3). Formal notations such as RSML or SCR can contribute to safety by allowing requirements to be specified in a precise and unambiguous manner and by supporting various analysis techniques for early error detection. A first level of analysis attempts to show that the functional requirements of a control system are consistent and that they possess desirable properties such as determinism and completeness. In this context, completeness means total coverage of the input domain: the response of the system to any value in the input domain is specified. Determinism and completeness ensure that the behavior of the control system is entirely predictable in all circumstances, a key property in safety-critical applications.

Consistency, completeness, and determinism of specifications are independent of the particular application considered and can be verified systematically. However, further safety analysis requires the verification of application dependent properties, namely that unsafe states cannot be reached. It may then become necessary to model components which are external to the control system or to specify assumptions about the environment.

In this paper, we examine the use of SCR and RSML for specifying the functional requirements of the main component of the SMS and for modeling the behavior of external hardware. After a short overview of the main features of the store management system, we

[1] This work was partially sponsored by the UK EPSRC under grant No GR/L49710

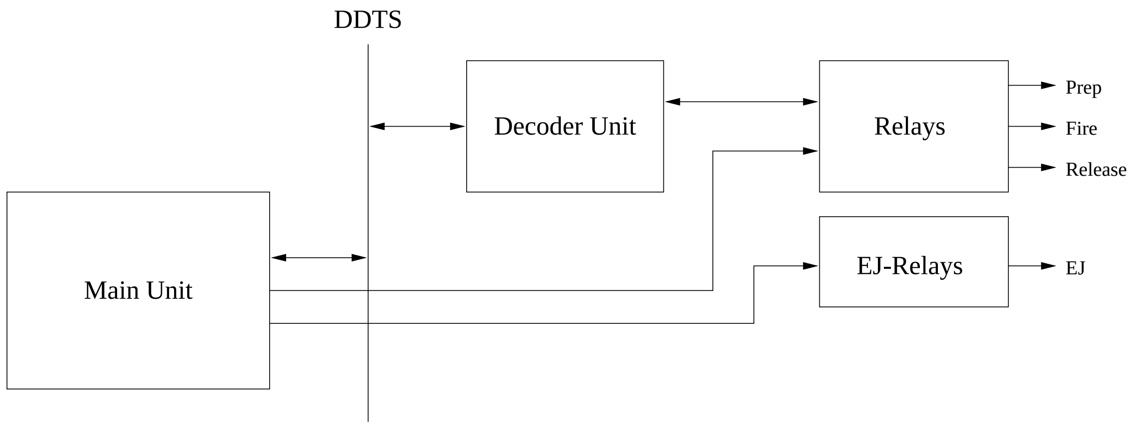

Figure 1 - SMS Architecture

present the two notations with examples based on the SMS requirements. We then compare the expressiveness and ease of use of the two formalisms and we discuss relevance of the various static analysis techniques proposed in the literature (refs. 2,5) to the SMS applications.

<u>The Store Management System</u>

Figure 1 shows the overall SMS architecture. The system consists of a main control unit connected to components known as decoder units via a bus called the DDTS (Digital Data Transmission System). There is a decoder unit for each weapon store. The main unit receives commands and weapon selections from the pilot(s) via switches, buttons, and control panels. The system can be in different modes controlled by two main switches: the master arm system switch (MASS) and the late arming switch (LAS). The MASS can be in position Off, On, or Live and the LAS can be in position Active or Inactive. The position of these switches determines which operations can be performed by the SMS. For example, selecting weapons is possible as long as MASS is not Off but firing is allowed only when MASS is Live and LAS is Active.

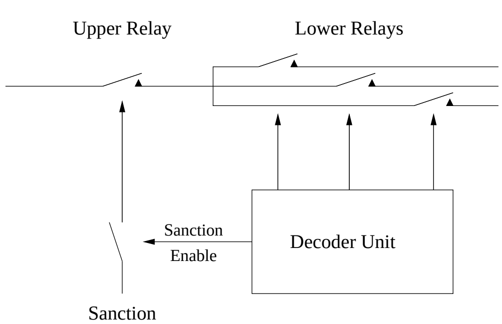

Figure 2 - Release Relays

The arming, fusing, release, fire, and jettison of weapons is achieved by closing electro-mechanical relays in each store. These relays are organised in two sets: lower relays and upper relays. The lower relays are commanded to open or close indirectly via the decoder units. The upper relays are controlled directly by the main unit via discrete sanction signals. The relays are assembled as shown in Figure 2. Each operation requires first the appropriate lower relays to be closed and the appropriate sanction signal to be enabled via the decoder unit. Then the main unit pulses the sanction signal to prepare, release, or fire the selected weapons.

Jettison is performed in a similar way via the decoder unit. In addition, each store has separate backup relays for emergency jettison (EJ-relays) which are directly controlled by the main unit. This ensures that failure of the decoder unit does not prevent emergency jettison.

The status of the relays is monitored by the decoder units and the information sent to the main unit via the DDTS. This allows the main unit to check that the proper lower relays have been closed before activating the sanction signals.

In the following sections, we examine the use of both RSML and SCR for specifying the functional requirements of the main unit and for modelling the sets of relays together with the associated decoder units. The model must specify the behaviour of the stores both in normal conditions and in the presence of component failures. We then examine how the main safety properties can be verified, namely that accidental release of weapons does not occur and that emergency jettison is available, even in the presence of a single component failure. In this analysis, we ignore the possibility of failures of the main unit and only consider failures of relays or decoder units. In practice, the main unit is implemented by two redundant software channels for normal operations and a separate hardware channel for emergency jettison which is also dual redundant.

SCR Notation

The SCR (Software Cost Reduction) approach to requirements was developed in the early 80s to specify the software requirements of the operational flight program of the A-7 aircraft (ref. 4). The essential features of the notation were introduced at that time. More recent developments have focused on formal analysis techniques for SCR and on providing tools for supporting the approach (refs. 5-6). Several variants of the notation exist; our discussion is based on the description given in reference 5 and on real-time extensions used in reference 8.

SCR is based on the four-variable model proposed by Parnas and Madey (ref. 7). In this model, the intended black-box behaviour of a control system is specified as a binary relation REQ between *monitored* and *controlled* variables. Monitored and controlled variables represent environmental quantities which are the input and output of the controlled system, respectively. SCR is essentially a tabular notation for specifying binary relations such as REQ. In addition to monitored and controlled, SCR introduces two other categories of variables called *mode classes* and *terms*. Roughly, mode classes define the state space of the control system and terms are auxiliary functions. The current value of a mode class is called a *mode* of the system.

An SCR document consists of a list of tables; each table defines a mode class, term, or controlled variable, as a function of monitored variables or of previously defined mode classes and terms. SCR uses three types of tables: mode transition tables, event tables, and condition tables.

Table 1. Mode Transition Table

Old Mode	Event	New Mode
Unpowered, LiveActive, LiveInactive	@T(Mass = On)	Safe
Safe, LiveActive, LiveInactive	@T(Mass=Off)	Unpowered
Safe	@T(Mass=Live) WHEN Las=Inactive	LiveInactive
Safe	@T(Mass=Liive) WHEN Las=Active	LiveActive
LiveActive	@T(Las=Inactive)	LiveInactive
LiveInactive	@T(Las=Active)	LiveActive

A mode transition table specifies how a mode class evolves when external events occur, that is, when the value of monitored variable changes. The new value of the mode class (i.e. the new mode) depends on the old value (the old mode) and the current event. For example, Table 1 defines a mode class of the SMS called MasterMode as a function of two monitored variables MASS and LAS representing the two main switches controlling the SMS. The second column of the table contains events which occur when either switch is moved to a new position. Syntactically, events are expressions of the form @T(cond1) or @T(cond1) WHEN cond2 where cond1 and cond2 are two *conditions*, that is, two predicates on SCR variables. Informally, @T(cond1) is interpreted as “when cond1 becomes true” while @T(cond1) WHEN cond2 means “when cond1 becomes true provided cond2 is true”. For example the third row of Table 1 specifies that the SMS enters mode LiveInactive when Mass is switched to position Live while Las is on position Inactive.

An event table specifies how a term or a controlled variable is updated when an event occurs. The new value depends on the event and on the current value of a mode class associated with the table. For example, Table 2 specifies when the prep-sanction signal for store *i* has to be activated or cleared, according to the current mode of operation of the SMS, as specified by mode class MasterMode. The sanction signal is represented by a controlled variable PrepSanction[i]. In all modes except LiveActive, the signal must be Cleared. This is indicated by the three first rows of Table 2; the expression @T(Inmode) denotes the entry of the system in the mode indicated in the first column. For example,

Table 2. Event Table

Mode	Events	
Unpowered	@T(Inmode)	False
Safe	@T(Inmode)	False
LiveInactive	@T(Inmode)	False
LiveActive	@T(EJinit=On) or @T(Time – Last(@T(Fire[i]))) >10ms or @T(not StoreSelected[i])	@T(Inmode) WHEN StoreSelected[i] or @T(StoreSelected[i])
PrepSanction[i]	Cleared	Active

the first row of Table 2 says that PrepSanction[i] has to be turned off as soon as MasterMode becomes equal to Unpowered. The fourth row specifies the behaviour of PrepSanction[i] in mode LiveActive. The sanction must be activated when a weapon on store *i* is selected or on entry in mode LiveActive if the store has been previously selected. The signal must be cleared when emergency jettison is initiated, when the store is deselected, or 10ms after the selected weapons on store *i* have been fired. Timing is specified using an extra monitored variable Time. The expression of the form Last(event) is interpreted as the value of Time at the last occurrence of event. This manner of specifying real-time requirements in SCR is inspired from reference 8.

Table 3. Condition Table

Mode	Conditions	
Unpowered	True	False
Safe	True	False
LiveInactive	Select[i] = None	Select[i] ≠ None
LiveActive	Select[i] = None	Select[i] ≠ None
StoreSelected[i]	False	True

Condition tables define the value of a term or controlled variable as a function of a mode class and various conditions. For example, Table 3 defines the term StoreSelected[i] as a function of a monitored variable Select[i] and of the current mode. Select[i] represents the current status of store *i* as programmed by the pilot(s). As previously, each row of the table corresponds to a mode; each column specifies the value of StoreSelected[i] according to different conditions. The columns are to be read as a disjunction of cases. For example, the second column of the table indicates that StoreSelected[i] is false when MasterMode is Unpowered or Safe or if Select[i] is equal to None in the two other modes.

Together with a collection of tables of the three above types, a complete SCR specification includes additional information such as the type and initial value of the diverse variables. Semantically, the whole set of tables defines a state machine in which transitions are triggered when some of the monitored variables change value (ref. 5). Each variable of the specification corresponds to a state component and each table specifies how and when a single component is updated. For example, in a specification including tables 1-3, a state of the system includes the three components MasterMode, StoreSelected[i] and PrepSanction[i]. When the monitored variable Mass changes from On to Live, the event @T(Mass=Live) occurs. This causes the mode class MasterMode to be updated as specified in Table 1 and the value of MasterMode becomes either LiveActive or LiveInactive. As a result, the value of StoreSelected[i] is modified as defined in table 3. This may cause event @T(StoreSelected[i]) to occur and the value of PrepSanction set to Active. Further events may then be generated and new variables be updated and so on. Hence, an input event, i.e. a spontaneous change in the monitored variables, causes a chain of reactions whereby internal events are triggered and the non-monitored variables are updated one by one.

SCR relies on the so-called synchrony hypothesis: events and variable updates are assumed to take no time and to all occur simultaneously. The global transitions of the state machine are defined as the combination of the changes to individual variables defined by the different tables. Such a global transition is triggered by a single input event and is considered to be instantaneous.

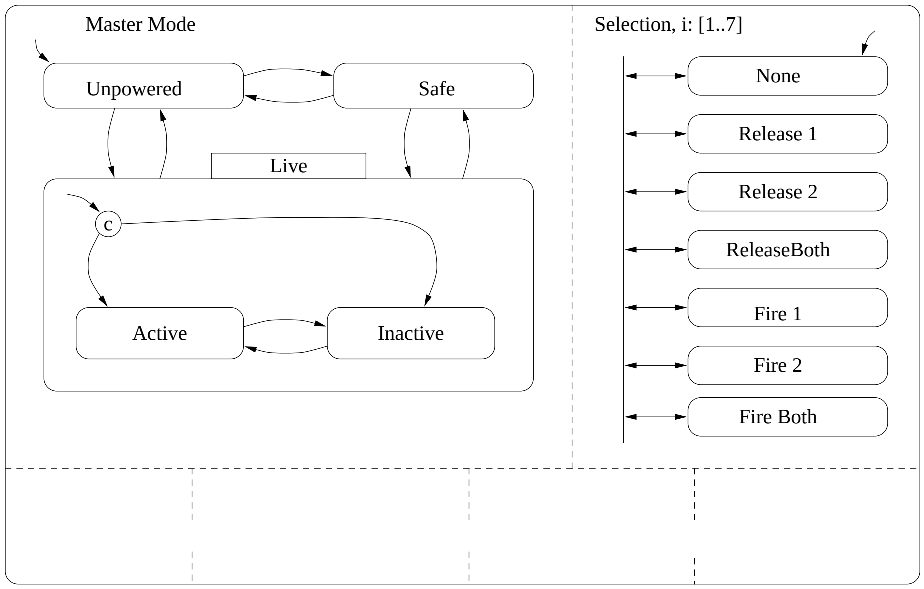

Figure 3. RSML Diagram

Consistency of SCR Specifications

Static analysis can be applied to SCR specifications in order to detect errors. The checks cover proper syntax and type correctness as well as the verification of more complex properties. SCR specifications are said to be *consistent* if the three following properties are satisfied (ref. 5):

- There is no circular dependency between the different tables/variables.
- All mode transition tables, event tables and condition tables are deterministic.
- All condition tables specify total functions.

These conditions ensure that the state machine associated with a set of tables is well defined and deterministic. Informally, the first condition implies that a single input event cannot cause an infinite chain of internal reactions. The second condition ensures that the state-machine is deterministic. The third property guarantees that the new value of a variable associated with a condition table is defined in all cases. In any state, the response of the system to any input event is completely specified and there is only one possible resulting state. The value of all the non-monitored variables in the next state is fully determined.

The verification of determinism and of the completeness of condition tables relies on predicate logic manipulations and tautology checking (ref. 5). For example, looking at the

$$\neg(\text{Select[i]=None} \wedge \text{Select[i]} \neq \text{None})$$

$$\text{Select[i]=None} \vee \text{Select[i]} \neq \text{None}$$

fourth row of Table 3, one has to check that the two formulas below are tautologies:
This ensures that the two conditions in the row are disjoint (first formula) and that all possible cases are covered (second formula). The static analysis techniques used to check that event and mode transition tables are deterministic are very similar and essentially check that certain events in a table cannot occur simultaneously (ref. 5). Tools are available to perform the verification (ref. 6).

RSML Notation

RSML (Requirement State Machine Language) is presented in reference 1. The notation is a variant of Harel's statecharts (refs. 10-11) and relies on the same notion of hierarchical state machine. The main elements of RSML are state-transition diagrams of the form shown in Figure 3 and a mixed textual/tabular notation for defining the transitions. In addition, an RSML document includes the declarations and types of input and output variables, the definition of macros and functions, the description of interfaces, etc. (ref. 1).

Figure 3 is a fragment of an RSML diagram from the SMS requirements. Every box in the picture represents a state and arrows represent transitions between states. As in statecharts, states can be either basic (e.g. Active or Inactive in the picture) or obtained by grouping together sub-states or by composing substates in parallel[2]. For example Live is the super-state of the basic states Active and Inactive. Parallel composition is represented graphically by dashed lines. The top-level box of the picture is the parallel composition of the sub-states MasterMode, Selection, and of other states not shown. Selection is itself an array of identical statecharts composed in parallel. An element of the array is denoted by Selection[i] and contains seven sub-states. The diagram also specifies that Unpowered and None are the *default sub-states* of Master Mode and Selection[i], respectively. The figure also contains a condition connector and a so-called transition bus; see reference 1 for details.

The two state composition operations are interpreted in the usual way (ref. 10). Exactly one of the sub-states of an OR-state is active when the OR-state is active and all the sub-states of an AND-state are active when the AND-state is active. For example, when the OR-state Live is entered, exactly one of its sub-states (i.e. Active or Inactive) is entered at the same time. When the transition from Live to Safe is taken, the system leaves the currently active sub-state of Live. On the other hand, when the top-level state of Fig. 3 is entered, both Master Mode and Selection[i] are entered simultaneously. Since these two are themselves OR-states, one of their sub-states is also entered, namely the state Unpowered for Master Mode and None for Selection[i].

The statechart labelled Master Mode specifies the different modes of operation of the SMS and corresponds closely to Table 1. The mode can be either Unpowered, Safe, or Live and in the latter case, the system can be in either one of two sub-modes: Active or Inactive.

In RSML, an arrow between two states indicates that a transition is possible between these two states and the diagram is accompanied a list of transition definitions. For each transition, the definition includes three essential components:

Transition: $\boxed{\text{Active}} \longrightarrow \boxed{\text{Cleared}}$

Location: PrepSanctionControl[i]

Trigger Event: ClockTick

Condition:

		OR			
AND	Master Mode **in state** Active	F	T	T	T
	EJinit=On	.	T	.	.
	t ≥ t(entered(Store[i]▸Fire))+10ms	.	.	T	.
	Selection[i] **in state** None	.	.	.	F

Output Actions: PrepSanctionCleared

Figure 4. Transition Definition

the trigger event, the guarding condition, and optionally the output actions. Figure 4 shows an example related to the control of the prepare sanction signal. The figure specifies when the prepare sanction signal has to be switched from active to cleared. This is very much the RSML equivalent of the second column of Table 2. The transition is identified by its source and destination states and by its location, that is, the statechart enclosing both the source and destination states. The transition is taken every time the event ClockTick occurs provided the guarding condition is true. The output action indicates that the event PrepSanctionCleared is generated when the transition is taken.

The guarding condition associated with a transition can be specified by an AND-OR table, as shown in Figure 4. AND-OR tables are intended to be readable representations of logical formulas in disjunctive normal form. For example, the table in Figure 4 specifies that the guarding condition is true either if statechart Master Mode is not in state Active, or if it is in state Active and the emergency jettison signal EJinit is On, etc. The third row contains a timeout specification: the predicate is true 10ms after the state Store[i]▸Fire, that is, the sub-state Fire of Store[i], has been entered.

The semantics of RSML is defined in reference 1. In many ways, it is similar to the semantics of SCR sketched above. An RSML specification defines a state machine whose global state (or configuration) is defined by the active states of all the statecharts components. The machine has input and output variables which corresponds to the input/output of the control system and moves from a global

[2] In the statechart terminology, states resulting from a grouping are called OR-states and states resulting from a parallel composition are called AND-states.

configuration to the next in response to input events. Input events trigger internal transitions (micro-steps) within individual statecharts. Such transitions may in turn generate internal events producing the same kind of chain reaction as in the SCR model. The reaction terminates when a stable configuration is reached, where no transition can be taken.

RSML Consistency and Completeness

The semantics of RSML is very close to certain variants of statecharts (ref. 11) but RSML is more restrictive. RSML does not allow non-deterministic behaviours and insists on a form of completeness. Various verification techniques are applied to guarantee the same key properties as in SCR: absence of infinite chain reactions, determinism, and completeness. The verification methods used are presented in reference 2.

The techniques used to check that infinite sequences of internal events cannot be generated and to ensure determinism are quite similar to those used for SCR. The situation is somewhat more complicated because several events may be generated in parallel. RSML uses interleaving: two events a and b occurring in parallel act as if they had occurred sequentially in an arbitrary order. For determinism to be preserved, one must show that the result is the same whatever the sequential order.

The notion of completeness used in RSML is very strong. For a basic state *S*, all the events which can trigger a transition from the state *S* are examined. RSML requires that, for all these events, there is always a transition of source *S* whose triggering condition is satisfied. RSML also requires that all the sub-states of an OR-state accept the same set of events, that is, if an event can trigger a transition in one of the sub-states, then it can trigger a transition in all the sub-states.

Comparison: Structuring Mechanisms

As can be seen from the informal presentation above, RSML and SCR are similar in many ways. Both formalisms rely on a state-based model where transitions are triggered by events. Input events initiate sequences of internal transitions which eventually terminate in a stable state. Both RSML and SCR have adopted the synchrony hypothesis: all the internal transitions take no time and the stable state is reached instantaneously. Both use tables for specifying transitions, functions, or predicates. The same forms of static verification are performed in both cases in order to ensure that the specified system is deterministic and complete. In many cases, the requirements can be translated fairly easily from one notation to the other[3]. Basic states in an RSML diagram correspond to modes or to the possible values of a term and conversely. The definition of transitions in RSML is often similar to a column of an event or mode transition table.

Although the two notations are roughly equivalent, we found parts of the SMS requirements easier to specify with RSML than with SCR. Some of the requirements are very procedural: they specify sequences of operations to execute in response to an initiation event. Figure 5 gives a simple example. The diagram contains two parallel statecharts specifying a main operation and a sub-operation. Each arrow

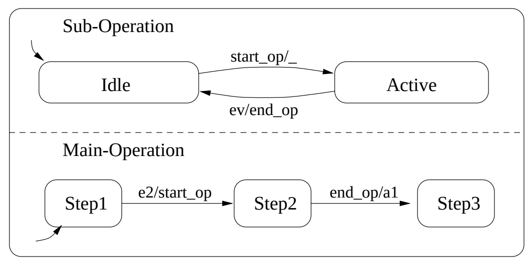

Figure 5. Operation and Suboperation

is labelled by a pair trigger_event / output_action and we ignore the triggering conditions. The main operation activates suboperation by generating event start_op then waits until suboperation terminates, as signalled by event end_op. This form of structuring requirements is very convenient for the SMS and the English language description of the system is organised in the same way.

It is difficult to preserve the structure of such requirements in SCR. A natural attempt would be to introduce two mode classes OpState and SubOpState representing the states of the two operations but this would not work because there would be a circular dependency between the two classes: The transition of SubOpState from Idle to Active is triggered by @T(OpState=Step2) and the transition of OpState from Step2 to Step3 is triggered by @T(SubOpState=Idle). There are

[3] We do not claim that there is a systematic translation procedure but in most cases, it is fairly easy to go from one notation to the other.

ways to work around this limitation but the natural structure of the requirements is lost. Another solution could be to extend SCR, for example, by allowing tables where several mode classes are defined simultaneously.

In general, SCR lacks structuring mechanisms. An SCR specification is a flat collection of tables, where states and transitions cannot be decomposed. On the other hand, RSML inherits the structuring mechanisms of statecharts based on the AND-OR decomposition. Comparing figure 3 and table 1 gives a small illustration of the difference. In RSML, the state Live is decomposed in two sub-states Active and Inactive. This is a very natural decomposition also present in the informal requirements. In the corresponding SCR specification (Tab. 1), we use two modes LiveActive and LiveInactive and the natural structure is lost. More generally, SCR has no facility for composing or constructing tables in a modular way. This may pose serious difficulties for large-scale requirements specification.

Comparison: Language Features

There are notational differences between the two formalisms: SCR is purely textual and tabular and RSML mixes graphics with text and tables. When specifying the SMS we did not find any particular advantage or disadvantage in having a more graphical rather than a purely textual notation. However certain features of RSML make it a more convenient notation. In particular the existence of arrays in RSML is useful in our case because requirements are similar for the seven stores of the system. The descriptions of SCR available in the literature do not indicate that the same facility exists. This is only a minor point and there does not seem to be any difficulty in including arrays of variables in SCR (as we have done in Tables 2 and 3).

More importantly, the notions of events used by RSML and SCR are quite different. In SCR, state variables are the essential primitive notion. Events result from a change in the value of state variables. An input event can be interpreted as the observation of a spontaneous state transition in the external environment. An internal event is the observation of a change in a non-monitored variable. In RSML, as in statecharts, there is a clear distinction between events and states. RSML events are the cause, not the observation, of transitions between states. The RSML notion of events is better suited to many aspects of the SMS. For example, in order to examine safety properties it is important to be able to describe hardware components and their failure modes. We have to model the relays used in the store subsystems. Since a hardware failure intuitively matches the notion of events used by RSML, the modelling in RSML is easy. It is possible to model hardware components and their failure modes in SCR but this may lead to unnatural specifications. Typically, one would introduce an extra monitored variable to indicate the current failure mode of a component. In general, the SCR view of the world, where the behaviour of the environment is modelled by state variables only, is too restrictive. Certain physical phenomena are described more naturally as events in the RSML sense (i.e. a clock tick, the reception of a message, etc.).

Another important difference between the two notations is the model they use to represent communications with the environment. The issue is related to the event model. In SCR, all communication is via the monitored and controlled variables. As all other SCR variables, the monitored and controlled variables are components of the states. In any state of the system all the monitored variables and the all controlled variables have a value. This model is adequate for representing input or output such as switch positions or hardware discretes. On the other hand, this model is not very convenient for message-based communications. In RSML, all communication is via message passing. Special SEND and RECEIVE events denote the reception or emission of messages from the environment and a global transition in the RSML model is always caused by a RECEIVE event. There are also input variables but all activity must be triggered by events. Unlike in SCR, there is no straightforward way to specify that an action must take place when a property of the input variables becomes true. This leads to some redundancy: in our RSML specification, the master arm system switch is represented by an input variable called Mass but we also use a separate event to indicate that Mass has changed. In summary, the communication model of SCR is better for modelling inputs which have a value at all time (e.g. a signal from a switch) while the communication model of RSML is better for modelling intermittent input (e.g. messages). As the SMS shows, both forms of communication can exist for the same system. It would be desirable for RSML and SCR to be extended to cover both communication models.

The RSML notion of completeness requires the specification to be more explicit than in SCR. In RSML, one must state when and under which conditions various events can be ignored. This is specified by transitions from a state to itself[4]. In SCR, mode transition and event tables specify when modes or the value of a variable have to change. If no event in the table occurs then the mode or the variable does not change. The circumstances under which no change of value takes place are left implicit. Only the active transitions are described. It is not clear which of the two options is the best. Although making all requirements explicit is usually considered to be good for safety, the RSML notion of completeness may be too strong. It is reasonable to require an answer to be explicitly defined in all configurations for all input events, but RSML completeness also applies to internally generated events. For example, for the diagram of Figure 5 to be complete, one has to specify that there is a transition from state Active to itself triggered by event start_op. This transition states that start_op must be ignored in state Active but start_op cannot occur in state Active anyway[5].

Finally, we found that both notations share some limitations. In both formalisms, it is assumed that only one input event can occur at any time. This is often an essential assumption to ensure determinism in SCR. For example, Table 1 is not consistent if both Mass and Las can change at the same time. However, it may be hard to justify this assumption in practice. It would be better to be able to specify explicitly how to handle simultaneous events, for example, by indicating that an event has priority other another or by giving synchronization conditions.

We found it difficult to specify built-in-test (BIT) procedures. The SMS uses three forms of BIT: one is activated at power-up, one is initiated by the pilot, and one is performed continuously. The continuous BIT does not fit well in the reactive model used by both notations. There does not seem to be an easy way to specify the procedure in either SCR or RSML.

[4] Although, some RSML papers discuss identity transitions which seem close to the default "no-change" of SCR.

[5] Provided MainOperation is the only statechart which can generate event start_op.

Comparison: Analysis Techniques

With both notations, system requirements can be subjected to different verification techniques in order to establish consistency and completeness. In both cases, specifications are consistent if there is at most one possible response to any input event and if there is no infinite chain reaction to a single input event. As regards completeness, RSML and SCR adopt different points of view. In SCR, completeness is a property of condition tables only. A condition table is complete if it defines a total function. As mentioned previously, RSML-completeness is a much stronger property. The techniques used by both notations to show consistency and completeness are quite similar and rely to a large extent on tautology checking (refs. 2, 5).

For further analysis, in order, for example, to verify that safety requirements are satisfied, a similar range of techniques can be applied to RSML and SCR. References 12 and 13 apply model-checking to SCR specifications by translating the notation in a format usable by existing model-checkers. A similar verification approach can be used for RSML specification (ref. 14).

The two formalisms (or some variants of them) can specify real-time requirements but the state-models and the verification techniques used do not cover these real-time aspects. The machine-based models used both by SCR and RSML are already fairly complex and adding real-time to these automata may pose significant problems.

Conclusion

RSML and SCR are two requirements specification formalisms for complex control systems. A particularly interesting aspect of both notations is the use of tables to improve readability. No doubt this is a key advantage of SCR and RSML over other formal notations. For some aspects of the SMS, we found RSML more comfortable than SCR because of the clear separation between events and states and because of better structuring mechanisms.

The two formalisms provide similar means of analysing specifications for determinism, consistency, and some form of completeness. The underlying state models used in both cases are very close and the specifications written in both notations are of a similar nature. One can thus expect that a similar sort of requirements errors will be detected in both cases.

In addition to the verification for internal consistency and for completeness, further analysis will require more sophisticated approaches such as model-checking, state-exploration, and theorem proving. Work already exists in these areas, especially in translating specifications to different formats recognisable by model-checkers. At the present time, SCR and RSML can be mostly applied to purely reactive control systems. Support for important aspects of avionics systems such as real-time is not as well developed.

Acknowledgements

The authors are grateful to Axel van Lamsweerde for his careful reading of this paper and his helpful comments as well as to Jonathan Draper for his continuous support to this work.

References

1. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specifications for Process-Control Systems. *IEEE Trans. on Software Engineering*, 20(9):684-707, September 1994.
2. M. Heimdahl and N. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Trans. on Software Engineering*,22(6):363-377, June 1996.
3. N. Leveson, *Safeware: System Safety and Computers.* Addison-Wesley, 1995.
4. K. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and their Applications. *IEEE Trans. on Software Engineering*, 6(1):2-12, January 1980.
5. C. Heitmeyer, R. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Trans. on Software Engineering and Methodology.* 5(3):231-261, July 1996.
6. C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A Toolset for Specifying and Analyzing Requirements. *In proc. COMPASS'95.* pp. 109-122, 1995.
7. D. Parnas and J. Madey. Functional Documentation for Computer Systems. *Science of Computer Programming.*25(1):41-61, October 1995.
8. P.-J. Courtois and D. Parnas. Documentation for Safety Critical Software. *In proc. 15th Intl. Conf. on Software Engineering.* pp. 315-323, 1993.
9. A. J. van Schouwen, D. Parnas, and J. Madey. Documentation of Requirements for Computer Systems, *In RE'93*, pp. 198-207,1993.
10. D. Harel. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*. 19:231-274, 1987.
11. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology*. 5(4):293-333, October 1996.
12. J. Atlee and J. Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Trans. on Software Engineering*, 19(1):24-40, January 1993.
13. R. Bharadwaj and C. Heitmeyer. Model-Checking Complete Requirements Specifications using Abstraction. Report NRL/MR/5540-97-7999, Naval Research Lab., November 1997.
14. R. Anderson et al. Model Checking Large Software Specifications. *SIGSOFT'96*, October 1996.

Biographies

Bruno Dutertre, Computer Scientist, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, telephone – (650) 859-2717, facsimile – (650) 859-2844, e-mail – bruno@csl.sri.com.

Bruno Dutertre received a DEA (Master) and a PhD from the University of Rennes 1, France. His main research interests are formal methods, logic and theorem proving, and their application to high-integrity systems.

Victoria Stavridou, Senior Computer Scientist, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, telephone – (650) 859-4590, facsimile – (650) 859-2844, e-mail – victoria@csl.sri.com.

Victoria Stavridou holds a BSc on Electronic Computer Systems and an MSc in the Assessment of Computer Aided Logic Design, both from the University of Salford , UK, as well as a PhD on Equational Specification and Verification of Digital Systems from the University of Manchester. Her research interests include safety-critical systems, formal methods, and dependability. She has written extensively in these areas.