EXPERIENCES FROM SPECIFYING THE TCAS II REQUIREMENTS USING RSML

Mats P.E. Heimdahl, University of Minnesota, Minneapolis, MN
Nancy G. Leveson, Massachusetts Institute of Technology, Cambridge, MA
Jon Damon Reese, Safeware Engineering Corporation, Seattle, WA

Introduction

TCAS II (Traffic alert and Collision Avoidance System II) is an avionics system required on all commercial aircraft with more than 30 passengers. In 1990, FAA deemed the Minimal Operational Performance Standard (MOPS) for TCAS II, expressed in plain English and low-level pseudocode, unacceptable as a basis for government certification. To correct this problem, a high-level software requirements specification (SRS) was developed by reverse engineering the pseudocode and capturing the required behavior using RSML (Requirements State Machine Language), a requirements language based on hierarchical communicating finite state machines [1]. The Irvine Safety Research Group, in cooperation with industrial and government representatives, specified the requirements between 1990 and 1992 [1].

In this paper we provide an overview of this project and the specification technique we used. In the following section we discuss the TCAS II project. Next, we cover some desirable properties of a high-level specification language and provide an overview of RSML. Finally, we share lessons learned and outline current developments.

The System—TCAS II

In 1981, the FAA decided to develop and implement TCAS II. TCAS II is an airborne device that functions independently of the ground-based air traffic control system to provide collision avoidance protection for a broad spectrum of aircraft types (commercial aircraft and larger commuter and business aircraft). To avoid threatening aircraft, TCAS II alerts the pilot of nearby traffic (traffic advisories) and, if necessary, provides recommended escape maneuvers (resolution advisories) in a vertical direction. In 1989, the FAA required that TCAS II be installed on commercial aircraft with more than 30 seats by December 1993 and on commercial aircraft with 10 to 30 seats by 1995.

At this time, the TCAS II requirements were defined by the Minimal Operational Performance Standard (MOPS) document. The MOPS was expressed using a combination of English and approximately 7,000 lines of low-level pseudocode.

Because of perceived deficiencies in this document (discussed in the next section) and the difficulty of FAA certification without high-level system or software requirements, an effort was begun in 1990 to provide such a high-level requirements document for TCAS II.

The Original Specification; The MOPS

The MOPS was expressed using both English and pseudocode, the English is used to comment and explain the code. The pseudocode is a low-level language (called E) containing only simple data types, arithmetic expressions, if statements, loop statements, and subroutines. All variables are global: There are no local variables and formal parameters but there are language constructs to indicate which variables are used and modified by a subroutine (few subroutines actually use this feature in the TCAS specification). An example of a routine

can be seen in Figure 1. The only complex data structure allowed is a "group" that provides for grouping related variables into a "data structure", that is, giving them a group name. In summary, the pseudocode lacks many desirable features of a modern programming language and, more importantly, most desirable features of a high-level specification language.

```
PROCESS Ground_level_estimation;

  IF (O.OOGROUND EQ $TRUE)
    THEN G.ZGROUND = G.ZOWN;
  ELSEIF (G.RADAROUT GT P.RADARLOST)
    THEN G.ZGROUND = -P.ZLARGE;
  ELSEIF (G.RADAROUT GT 0)
    THEN <do not update ZGROUND>}
  ELSEIF (G.RADAROUT EQ -P.ZLARGE)
    THEN
      IF (O.ZRADAR LT P.KNOWGROL)
        THEN G.ZGROUND = G.ZOWN -
O.ZRADAR;
        ELSE; <ground level unchanged>
  OTHERWISE
    IF (O.ZRADAR GT P.KNOWGROH)
      THEN G.ZGROUND = -P.ZLARGE;
      ELSE G.ZGROUND = G.ZOWN - O.ZRADAR;

END Ground_level_estimation;
```

Figure 1. A sample routine from the MOPS

The Participants

To provide a proper requirements specification, a committee of industry and government representatives (RTCA Special Committee 147, or SC-147) was formed and began to develop an English language specification. Participants in this committee included airframe and avionics manufacturers, air traffic controllers, airline representatives, pilots, and FAA representatives. At this time, Dr. Nancy Leveson was looking for a real world system she could use as a testbed for her research in safety-critical systems. Because of its size and complexity, TCAS provided a challenging experimental application of formal specification and analysis methods to a real system.

Initially, the Irvine Safety Research group developed a requirements document purely for research purposes, but it was later adopted by RTCA Special Committee 147 for their specification of the TCAS system requirements.

The requirements were expressed using RSML. The group developed RSML concurrently with the TCAS II requirements, and the experiences and feedback gathered during this effort was continually used to refine and improve the language. A brief description of RSML can be found in the next section. A detailed discussion about the reasons for developing RSML is included in [1]. In short, no other language satisfied our demands for readability and usability combined with formality.

The Solution – RSML

During the initial stages of the project, we reviewed many common specification methods. Unfortunately, we could not find any language and method that satisfied our needs.

The first step in designing a specification language or modeling method is to determine goals and criteria for the language. This section describes general design criteria for such a requirements specification language as well as a short overview of the language used to specify TCAS—RSML.

The Goals of RSML

We identified several criteria that were important with respect to our goals and that we believe apply in general to this type of specification language (a partial list is shown in Figure 2).

- Black-box
- Minimal
- Semantically simple
- Coherent, consistent, and concise
- Unambiguous and formal
- Readable, reviewable, and usable by application experts and developers
- Flexible notations
- Readability given priority over writability
- User needs given priority over personal preferences

Figure 2. Design criteria for the language.

The first criterion is that the language specifies blackbox behavior of the software only and does not include internal design information in the specification.

Two other criteria are minimality and simplicity. Minimality implies that the specification should contain only the information needed by the developers and analysts. Otherwise, time is wasted in specifying things that are not used. Many of the popular real-time requirements specification languages include facilities that are not strictly necessary. The problem with the "kitchen sink" approach is that the specification language becomes unnecessarily complex and the specification process becomes unnecessarily tedious and time-consuming.

Related to the minimality and simplicity criteria are coherency, consistency, and conciseness. Other specification languages for reactive systems, e.g., Statemate [2], Hatley/Pirbhai [3], and Ward/Mellor [4] include a variety of diverse models, some of which are not formally defined. Our goal was to specify all the required information using one formally defined modeling language based on one underlying state-machine model. We also wanted our language to represent information as economically as possible while still maintaining readability.

Because of our goal to provide a safety analysis of the specification, the language must be unambiguous and the underlying model must have a mathematical foundation. At the same time, the requirements specification must be readable, reviewable, and usable. In some respects, these criteria may be conflicting but it is possible to satisfy both. The specification must be unambiguous and translatable into a formal foundation, but it need not itself include arcane mathematical symbols that are unfamiliar to the application experts and software developers. We spent considerable time and energy developing a notation that was readable yet maintained the underlying formal state-machine model. This notation has graphical, symbolic, and tabular aspects depending on which was best for specifying a particular type of information [5].

Because readability and writability are often conflicting goals, we chose readability in cases where a conflict existed: The added investment in constructing the requirements specification pays off in terms of discovering more requirements-level errors.

As mentioned earlier, we received continual industry feedback on our specification effort. One of the advantages of the feedback was to help us overcome our individual preferences. When devising the specification language, we usually had ourselves in mind as the user. However, our familiarity with certain notations, especially mathematical notations such as predicate calculus, hid their weaknesses. Our first attempts at devising our language, therefore, were failures: the notation was clear to us but not to others. The feedback from a diverse group of users helped us to evaluate the evolving specification language more objectively. The resulting language as well as specification was by all measures very successful.

RSML Overview

RSML is a state-based requirements specification language suitable for the specification of reactive systems. RSML includes several features developed by Harel for Statecharts [2, 6], for example, superstates, AND decomposition, broadcast communication, and conditional connectives. In addition, RSML has some unique syntactic and semantic features that were developed to enhance readability, reviewability, analyzability, and the ability to handle complex systems. A complete description of RSML is provided in [1]. This section contains only a brief overview of the language.

A simple finite-state machine is composed of *states* connected by *transitions* (see Figure 3). *Default* or start states are

signified by states where a connecting transition has no source. In Figure 3, state *Within-Limits* is the start state. Transitions define how to get from one state to another.

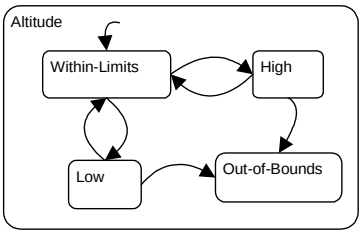

Figure 3. A Basic State Machine Example.

Superstates

In RSML (and Statecharts), states may be grouped into *superstates* (see Figure 4). Such groupings reduce the number of transitions by allowing transitions to and from the superstate rather than requiring explicit transitions to and from all of the grouped states (*substates*). Superstates can be entered in two ways. First, the transition to the superstate may end at the superstate's border (transition T_1 in Figure 4). In this case, a default state must be specified within the superstate. In the example, state *Climb* is entered upon taking transition T_1. Alternatively, the transition may be made to a particular state inside the superstate (transition T_2 in Figure 4). The same superstate may have transitions ending at the border and at any number of the inner states. The superstate may be exited in two ways (transitions T_3 and T_4 in Figure 4). Analogous to transitions into the superstate, transitions out of the superstate may originate from the border or from an inner state.

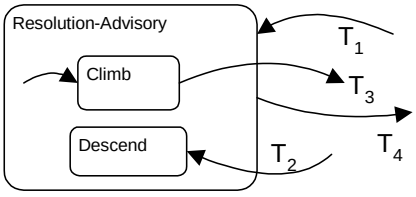

Figure 4. A Superstate Example.

AND Decomposition

One of the most important innovations in Statecharts is what Harel calls the *parallel state*[1], which contains two or more state machines separated by dashed borders (Figure 5). When the parallel state *S* is entered, *each* of the state machines *A*, *B*, *C*, and *D* within it is entered. All state machines are exited when *any* transition is taken out of the parallel state. The use of parallel states greatly reduces the size of the specification.

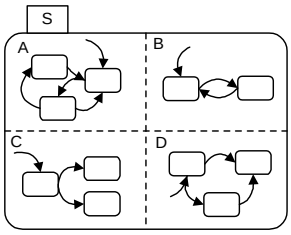

Figure 5. The Parallel State.

Naturally, the TCAS II specification is significantly more complex than the small examples above. To illustrate, we have included a part of the TCAS II specification.

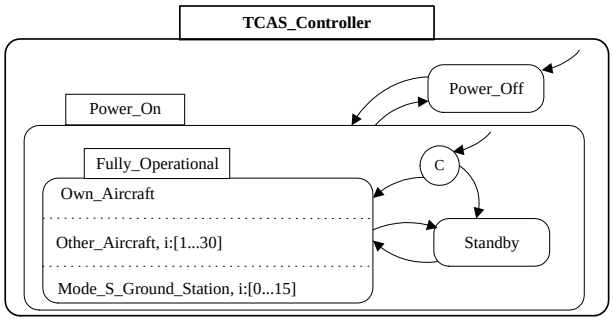

Figure 6. The Highest Level of TCAS.

The highest level TCAS state machine is shown in Figure 6. At this level, TCAS is either on or off; if it is on, it may be either fully operational or in standby mode. In the case of the TCAS logic, the states of three types of process components are modeled: our own aircraft, other aircraft, and mode-S ground radar stations. Each of the three subcomponents of TCAS is elaborated in more detailed RSML models.

[1] Parallel states are also known as "orthogonal products'", "product states'", and "AND states'".

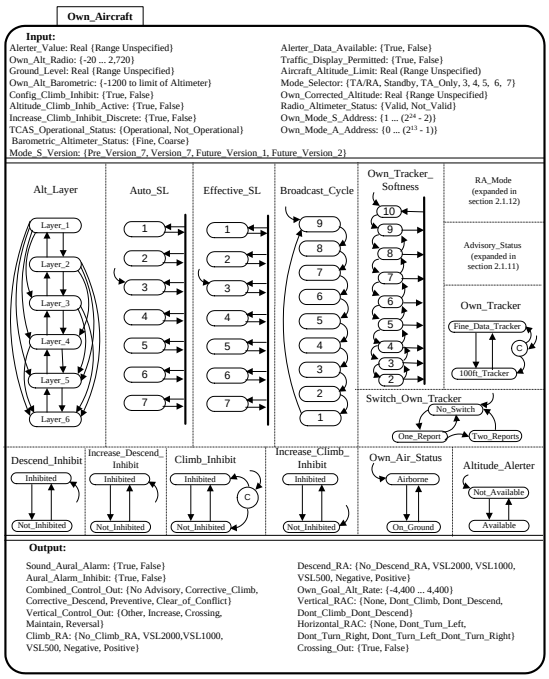

Figure 7. The Model of Own Aircraft

Figure 7 shows the expanded Own-Aircraft portion of the TCAS model. The top portion of the diagram lists variables that represent inputs to TCAS from the sensors that provide information about the state of Own-Aircraft. The bottom portion of the diagram lists variables that represent outputs to TCAS actuators. The middle portion of the diagram represents the parts of the derived Own-Aircraft state necessary for the evaluation of the TCAS control function. Again, for a detailed discussion the reader is referred to [1].

Transition Definitions

Transition definitions in RSML contain five parts: (1) the identification (the source and destination of the transition), (2) the location, (3) the triggering event, (4) the guarding condition, and (5) the output action. The identification, location, and triggering event are the only required parts.

Transitions are taken upon the occurrence of the *trigger event*, provided that the guarding condition is true. The *guarding condition* defines preconditions on the transition and is specified using AND/OR tables, described below. *Output actions* identify events that are generated when the transition is taken. These newly generated events may now trigger transitions elsewhere in the state machine. This *event propagation mechanism* is used to sequence and synchronize the execution of the parallel state machines in the model.

AND/OR Tables

Many state-based languages use standard logic notation to describe the guarding conditions on the transitions [6,8]. Our TCAS external reviewers, however, did not find this notation natural or reviewable. Instead, we decided to use a tabular representation of disjunctive normal form that we call AND/OR tables (see Figure 8 for a transition from the TCAS II requirements).

The far-left column (the wide column) of the AND/OR table lists the logical phrases in the condition. Each of the other columns represents a conjunction of those phrases and contains the logical values of the expressions (a column denotes the logical AND of the phrases in the wide column). A column evaluates to true if all of its elements are true. A dot denotes “don’t care”. The collection of columns represents a disjunction (indicated by the OR above the columns). Thus, if one of the columns is true, then the guarding condition (the table) evaluates to true.

Experience using RSML

We used this language to develop an experimental specification of TCAS II. This initial specification was well-liked by the RTCA TCAS II Committee. The members felt that adopting a formal approach would help to accurately capture the requirements and enable a shorter time to certification. The RSML language has been subsequently formally adopted for the official requirements specification.

Location: Own_Aircraft$_{s\text{-}40}$ ▷ Auto_SL$_{s\text{-}89}$
Trigger Event: Descend_Inhibit_Evaluated_Event$_{e\text{-}682}$
Condition:

				OR	
	Effective_SL$_{s\text{-}97}$ **in one of** {1, 2}	.	.	.	T
	Effective_SL$_{s\text{-}97}$ **in state** 3	.	.	T	.
	Own_Alt_Radio$_{v\text{-}43}$ ≤ 900 ft$_{(ZSL3TO2)}$	.	.	T	.
AND	Climb_Descend_Inhibit$_{m\text{-}370}$	.	T	F	.
	Own_Air_Status$_{s\text{-}141}$ **in state** Airborne	F	T	T	T
	Traffic_Display_Permitted$_{v\text{-}50}$	T	.	.	.
	Radio_Altimeter_Status$_{v\text{-}46}$ = Valid	.	.	T	T
	Own_Alt_Radio$_{v\text{-}43}$ < 1,100 ft$_{(ZSL2TO3)}$	.	.	.	T

Output Action: Auto_SL_Evaluated_Event$_{e\text{-}682}$

Figure 8. A transition definition from TCAS II.

After SC-147 decided to adopt the RSML specification, we received continual feedback on our effort from the full SC-147 and from a smaller ad-hoc Requirements Working Group chartered with the development of the requirements.

In March 1992, the document was delivered to SC-147 for verification. The RSML specification contained approximately 47 input and output variables, 140 states, and 170 transitions. The specification has since then undergone an extensive verification and validation effort, and is currently being maintained and extended by the Rannoch Corporation.

Lessons Learned

During the course of the project we learned several lessons, some of which are summarized in this section. The lessons generally fell into two categories (1) general lessons regarding requirements specification and (2) lessons related to the use of formal specification languages.

General Lessons Learned

Keeping the design out is hard: Initially, we had difficulty abstracting away from the design. Even when we did not look at the pseudocode, we found it difficult in the beginning to eliminate functional decomposition and flowchart-like logic, i.e., to specify the problem without trying to solve it. With practice we became better at omitting design information, but the struggle never entirely abated. The very low level of the pseudocode also made the process of abstraction more difficult as many purely implementation features, such as flags, were used extensively in the pseudocode. After the specification of the CAS logic was completed, an independent verification and validation was performed to compare the pseudocode specification and the RSML specification. The verifiers experienced the same problems that we did, and a large number of identified discrepancies between the pseudocode and the RSML specification resulted in no change to the RSML specification because they merely represented design peculiarities of the pseudocode and not requirements.

Understanding intent is difficult: Although it may be a function of the particular system we were working on, we found it impossible to derive the requirements specification strictly from the pseudocode and an accompanying English language description. Although the basic information was all there, the *intent* was missing. The rationale for the various system design tradeoffs was never recorded. Therefore, distinguishing between requirements and artifacts of the implementation was not possible

in all cases. An audit trail of decisions and the reasons why decisions were made is absolutely essential. This was not done for TCAS over the 15 years of its development.

We should have started from scratch: The final requirements specification model would have been different and much simpler if we had been starting from scratch. Because the TCAS pseudocode specification had evolved over a period of more than 15 years, the current version contains more complexity than is necessary. This is a common maintenance dilemma, and TCAS was no exception. When changes are made to design or code without backing up all the way to requirements, such problems arise and increase as time passes. For TCAS, the highest-level specification *was* the pseudocode.

Because of the necessity of building a requirements specification that matched the TCAS systems actually in use (which were certified against the pseudocode specification), our resulting model was more complicated than necessary, included more than the minimum required behavior, and was harder to understand than was strictly necessary. This was frustrating as we first built a nice, simple model and found that we had to complicate it for no better reason than that it had to match some errors or poor design decisions in the pseudocode. We believe that if a blackbox behavioral model of our type had been built originally, not only would the final specification be simpler and more understandable, but making changes without introducing errors or unnecessarily complicating the resulting requirements also would have been simplified.

Language Lessons Learned

Formal requirements are feasible: One result of this effort was a demonstration that formal specifications can be applied to complex, reactive systems and that such specifications can be readable and reviewable by application experts with a minimal knowledge of mathematics and computer science. Formal specifications are clearly usable if their design takes into consideration the training and backgrounds of those who are to read and review the specification. Some engineers working with us on the TCAS specification reported that they liked the AND/OR table description of the transition condition because it resembles the logic tables that they are used to using and that the state machines and logic tables fit the way they think about systems.

Readability, simplicity, and usability are extremely important: Reviews of our document for correctness by users during development made clear that specifications should include graphical, symbolic, tabular and textual notation, depending on the type of information being conveyed. For example, the graphical state machines were a great help during reviews for finding certain types of errors as were the tables for finding other errors. A language that contains only graphics or only tables or only symbolic strings is probably less useful than one in which different notational techniques are used to communicate different types of information.

Easy to be blind to the users needs: Although formal specification languages obviously have to be defined in an unambiguous and mathematical way, the syntax itself does not have to contain obscure mathematical symbols that are familiar and comfortable to neither the application expert nor the implementor of the system. Currently, formal specification languages are designed primarily by mathematicians who use a notation with which they are comfortable, but which is foreign to those who must use the language. One solution is to train hardware and software engineers to think like mathematicians while our alternative solution is to provide languages that allow the user to think about the system in the way that they have been trained in their discipline. We hypothesize that providing a model of a system that is closer to the mental model that the reviewer and implementor have of the system will aid in finding errors in the specification itself and reduce the numbers of errors that are introduced when implementing the

specification. This hypothesis, of course, still needs to be experimentally validated, although our experience provides some convincing anecdotal support.

Summary and Recent Developments

This paper has discussed an approach to specifying system requirements for real-time, reactive systems, some criteria that should be used in designing a language for such requirements, and some lessons learned while writing a system requirements specification for an aircraft collision avoidance system.

Since the completion of the project we have made advances in many areas, two of which are of particular interest to the practicing engineer: (1) Specification and traceability of intent and design rationale and (2) an improved specification language and modeling methodology.

Intent Specifications

Leveson has developed a new way of structuring specifications called *Intent Specifications* [8]. Instead of the usual hierarchical abstraction based on *what* and *how*, Intent Specifications abstract on intent or *why*. Because each level of the specification is mapped to the appropriate parts of the intent levels above and below it, traceability of design rationale and design decisions is provided from system-level requirements and constraints down to code (or physical form if the function is implemented in hardware) and vice versa.

Intent specifications integrate formal and informal aspects of system and software development. The structure is designed to facilitate the tracing of system level requirements and constraints into the design and assist in the assurance of various system properties—such as safety, security, and survivability—in the initial design as well as reduce the costs of implementing changes and reanalysis when the system is changed, as it inevitably will be. Leveson has applied these ideas to specifying control systems with shared human and computer control and to assuring safety in these systems. Leveson and Reese then demonstrated the approach's feasibility by creating a complete (800-page) intent specification (from high-level system goals down through code) for TCAS II.

SpecTRM

After the completion of the TCAS II project our method and language have evolved. A commercial version of the approach is named SpecTRM (pronounced "spectrum") [8]. The name stands for Specification Tools and Requirements Methodology. The specification language in the SpecTRM approach is called SpecTRM-RL (RL for Requirement Language).

SpecTRM-RL is our new language based on the lessons learned from the experiences specifying TCAS as well as subsequent large case studies. SpecTRM-RL improves on usability, readability, and analyzability over RSML. It supports a complete system engineering methodology, is integrated with intent specifications, and extensive tool support will be commercially available shortly.

References

[1] N.G. Leveson, M.P.E. Heimdahl, H.Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, vol-20, no-9,September 1994.

[2] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, vol-16, no-4, April 1990.

[3] D. Hatley and I. Pirbhai. *Strategies for Real Time System Specification*. Dorset House Publishing, 1987.

[4] P. Ward and S. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, 1985.

[5] M. Fitter and T.R.G. Green. When do diagrams make good computer languages? *International Journal on Man-Machine Studies*, no-11, 1979.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231—274, 1987.

[7] C.L. Heitmeyer, R. Jeffords, and B.L. Labaw. Consistency checking of SCR-style requirements specifications. *ACM Transactions on Software Engineering and Methodology*, vol-5(3):231—261, July 1996.

[8] SpecTRM: A CAD System for Digital Automation. *Proceedings of the 17th Digital Avionics Systems Conference (DASC).* Seattle, WA, 1998.