# Experiences using Statecharts for a System Requirements Specification*

N.G. Leveson, M. Heimdahl, H. Hildreth, J. Reese, R. Ortega
University of California, Irvine
Irvine, CA 92717

## Abstract

*This paper describes some lessons learned and issues raised while building a system requirements specification for a real aircraft collision avoidance system using statecharts. Some enhancements to statecharts were necessary to model the complete system and a few notational changes were made to improve reviewability.*

## 1 Introduction

The problem of safety has gained importance as new computer applications have increasingly included control of systems where the consequences of failure may involve danger to human life, property, and the environment. Although system safety engineers have developed procedures to deal with hazards in physical systems, the introduction of computer control has created new and unsolved problems both for system engineers and for software engineers. Our basic approach to enhancing safety in computer-controlled systems is to apply special software hazard analysis and hazard control techniques throughout software development, linking them to the system hazard analysis and control procedures that are being performed in parallel by the system engineers at the system level. The approach relies heavily on formal modeling and analysis procedures.

In the past, the Irvine Safety Research Group has developed techniques for identification of software-related hazards [13], semantic analysis of software requirements specifications [9], software design techniques to enhance safety [11, 2], and verification of safety [12, 10]. Although many of these techniques are being used on industrial projects around the world, the set of techniques has been developed in isolation so they are difficult to use together and do not cover some important problem areas. Our current goal is to design a consistent, integrated approach to ensuring software safety throughout system development. The methodology is being developed while using a real avionics system (an aircraft collision avoidance system called TCAS II) as a testbed for immediate evaluation and feedback. The testbed will provide information about the practicality, feasibility, and usability of the methodology in parallel with its development. Since the TCAS II system is real, feedback is being provided by the Federal Aviation Administration (FAA) and other application experts (including airframe manufacturers and their subcontractors and pilot groups) outside the research community. Serious consideration is being given to how the methodology will interface with typical software development processes and what types of tools and support environments are needed to make it practical and efficient to use.

This paper describes our experiences and lessons learned while attempting to build a state-based black-box requirements model of the testbed system. We chose the statecharts modeling language [5] for two reasons: Our safety analysis techniques require a state-based, formally-defined specification language, and the abstraction and graphical capabilities of statecharts provide advantages over purely symbolic specifications. In order to enhance analyzability and understandability, we limited the Statechart features that we used, e.g., history was not used. Furthermore, in the process of building the model, we found it necessary to change the Statechart notation slightly and to enhance the communication features. We also identified some organization and style issues for such a specification.

Once the specification is completed, we will be developing analysis algorithms to find errors in the requirements model based on the formal criteria we have previously identified [9], special system and software

*This work has been partially supported by NSF Grant CCR-8718001, NASA Grant NAG-1-668, and NSF CER Grant DCR-8521398.

TH0388-9/91/0000/0031$01.00 © 1991 IEEE

hazard analysis techniques to identify software-related system hazards, procedures for generating standard system engineering analyses such as fault trees and failure modes and effects analyses directly from the model, and generating test data directly from the formal requirements specification.

## 2 The Problem

TCAS is a family of airborne devices that function independently of the ground-based air traffic control (ATC) system to provide collision avoidance protection for a broad spectrum of aircraft types (commercial aircraft and larger commuter and business aircraft) in order to reduce the risk of midair collisions. TCAS I provides proximity warning to assist the pilot in the visual sighting of intruder aircraft and is intended for use by smaller commuter aircraft and by general aviation aircraft. TCAS II provides traffic advisories and resolution advisories (recommended escape maneuvers) in a vertical direction to avoid conflicting aircraft. TCAS III will add resolution advisories in a horizontal direction.

TCAS makes use of radar beacon transponders routinely carried by aircraft for ATC purposes. The extent of protection provided by TCAS depends upon the type of transponder on the target aircraft; no protection is provided against aircraft that do not have an operating transponder.

Development of aircraft collision avoidance systems started over 20 years ago. In 1981, the FAA decided to develop and implement TCAS II, and a minimal operational standards (MOPS) document was produced using a combination of English and pseudocode. Since that date, changes have been made approximately yearly to the MOPS to fix errors or improve the specification. In 1989, the FAA required that TCAS II be installed on airline aircraft with more than 30 seats by December 1991 and on airline aircraft with 10 to 30 seats by 1995. The FAA can extend the first deadline to 1993 if they choose.

Because of perceived deficiencies in the MOPS and the difficulty of certifying aircraft with this document, an effort was begun in 1990 to provide a requirements document for TCAS II. An industry/government committee started writing an English language specification. At the same time, we were attempting to build a formal requirements specification and safety analysis for the FAA. In January 1991, the industry/government committee decided to stop working on their English specification and to adopt our formal specification as the official one.

This has been an interesting challenge with some partially conflicting goals. The specification must be formal enough for us to use as a basis for a safety analysis. It must also be readable enough for non-computer experts to read and review and be usable for both building and certifying TCAS II systems. During the development of the TCAS requirements document, we have learned some lessons about building a formal model of a large and complex system, and we have discovered several desirable properties of a state-based requirements language that are lacking in the most common languages of this type. The rest of this paper describes the lessons learned about formal modeling and specification.

## 3 Organization and Style

During the development of the requirements specification of the TCAS system, it became clear that certain organizational properties and a certain style of the state machine was desirable from a cognitive point of view. Assuming that our first attempt to build a usable model would be successful turned out to be naive on our part. We found it necessary to throw out models and start completely from scratch several times in order to increase the understandability, specifiability, and readability of the specification.

For example, we first decomposed the system requirements partly from a functional viewpoint (e.g., aircraft surveillance, tracking, advisory generation), primarily because all existing descriptions of the system used this approach. Most requirements specification languages used in industry involve functional decomposition. This approach led to a model that was much more complex than necessary and also difficult to understand and review. When we changed our specification to a pure blackbox behavioral model of the external process being controlled, application experts decided that it was easy to understand and to find errors in our blackbox model.

Even after we had settled on our final modeling approach, we still found the need to make significant changes in the basic structure of our model as we learned more about the system being specified. In this section we cover organizational properties or issues that we have discovered are important in a readable state-based specification document.

Although the idea of blackbox software requirements specification based only on external objects and values is not a new one [7], in the process of building the TCAS specification we came to understand better *why* it simplified the specification process and

enhanced reviewability. The goal of most control systems is to maintain some relationships between the input to the system ($\mathcal{I}_s$) and the output from the system ($\mathcal{O}_s$) in the face of disturbances ($\mathcal{D}$) in the process (figure 1). In the case of TCAS, the goal is to maintain a minimum separation between aircraft. That is, at any time, the relationship between components in the system must satisfy a function $F$ from the current system state, time, and external or internal events to a new system state. Theoretically, this function can be defined using only the true values describing the component states. However, at any time, the controller has only measured values of the component states (which may be subject to time lags[1] or measurement inaccuracies) and must use these measured values to infer the true conditions in the process and to output some corrective actions ($\mathcal{O}$) to maintain $F$.

Thus the controller (in our case, a computer) has to compute some function $f$ (representing our knowledge about the true function $F$) where $f$ is based on:

1. the inferred process state based on measurements of the controlled variables,
2. the past process conditions that were measured and inferred,
3. past corrective actions (outputs from the software), and
4. prediction of future states of the controlled process (in TCAS, these are predictions about aircraft positions)

in order to generate the corrective actions (or current outputs) needed to maintain $F$. It should be noted that all the information about the process has to be inferred from the indirect information provided by the input to the software ($\mathcal{I}$). In order to compute the corrective actions, the controller ($C$) uses a model of the process it is controlling. We call this the internal model — it is the view or model of the process that the software maintains and uses, and it is, of necessity, a simplification or abstraction of the system function $F$. This internal model may be quite simple, perhaps only a few variables, or it may need to be very elaborate, depending on the complexity of the system. Our current TCAS model has three components: our own aircraft, the other aircraft, and the ground-based sensors (see figure 2).

We found that the best way to specify the software requirements in terms of ease of specification and especially reviewability for errors was in terms of this

[1] Time lags are delays in the system caused by the reaction time of the sensors, actuators, and the actual process.

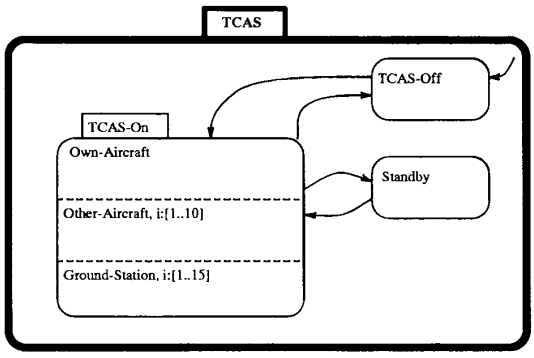


Figure 2: The model of the TCAS component

internal model. The dynamic behavior of $f$ is specified through the state changes required in the model as information is received about the state of the external world. The internal model is iteratively fine-tuned during requirements specification development to mimic our understanding of the process with sufficient detail to achieve the desired control performance.

It is the relationship between changes in the real world process state and the internal software model of the current process state that must be reviewed in a requirements review and therefore must be explicit in the requirements specification in order to make it reviewable. Errors in the requirements specification represent mismatches between this internal model of the process and the real process. When the description of the required controller behavior gets too far away from the real-world process (and, in fact, includes functional decomposition), it becomes increasingly difficult to validate the relationship between internal model and external process.

Although this type of specification is easier to review and simplifies finding errors, it does make the process of going from requirements to standard software designs based on functional decomposition a larger and perhaps more difficult step. It may turn out that very different software designs result naturally from such a "process-model-oriented" specification, and these designs may have distinct advantages for this type of reactive or control software.

In the process of building our model, other organizational and style issues surfaced with respect to the use of hierarchical abstraction and the decision to express something as a state or as a variable.

In our first attempts to build a model of the software requirements, we included unrestricted use of hierarchy in the statecharts. Although at first we thought that this would aid in understanding the spec-

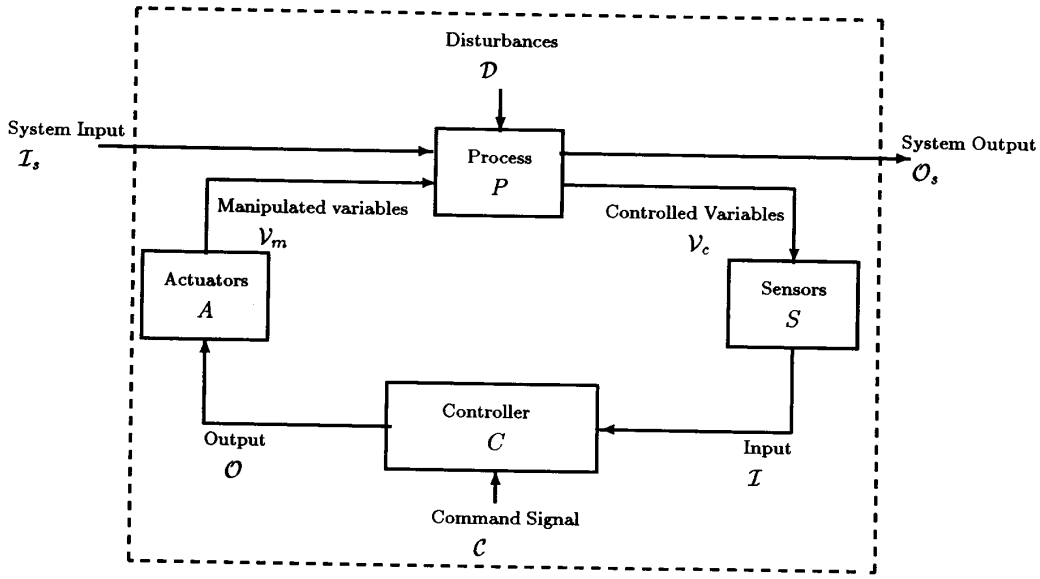


Figure 1: A basic process control model

ification, we discovered that in some cases it had the opposite effect.

The use of clustering (grouping states into superstates) indeed made the specification more readable. On the other hand, the use of abstraction, a type of information hiding that allows showing only the superstate and not the component substates, often had an undesirable affect on readability. One of the main advantages of such abstraction is that lower level information, i.e., substates, can be hidden from the reader and in that way the system is presented in digestible chunks. Our first attempts hid as much information as possible at every level. As a result, transition predicates that "crossed" levels (referred to nonvisible states) became very difficult to understand. We discovered that providg context to every transition was extremely important. In the TCAS specification, this was achieved by trying to display all information about a particular component in one statechart always visible to the reader. This "road map" provides the context necessary to easily understand the transitions. We call this concept of always providing as much information about the structure as possible "information exposure". The difficulty of providing adequate information exposure increases as the complexity of the model being built increases.

Information exposure appears to be related to the concepts of coupling and cohesion that have been identified for program design [16]. In a design, it is desirable to maximize cohesion (i.e., intramodule information association) and to minimize coupling (i.e., intermodule information sharing). Information exposure is directly related to the level of coupling; high coupling complicated the problems of providing adequate information exposure. Although it is not clear how much control the specifier has over cohesion and coupling when modeling based on physical components and values (rather than functional decomposition), the physical separation of the components may naturally limit the sharing of information to well-defined interfaces.

Another program design principle that seemed to apply to our specification was the notion of *uses hierarchies* [15]. A statechart component A can be said to use another component B if a state of B is used in the transition predicates describing A. For example, the component RA in Other-Aircraft (figure 3) uses information about both ground stations and own aircraft. Parnas has pointed out [15] that it is undesirable to have loops in the uses hierarchy of program modules, i.e., A uses B and B uses A either directly or indirectly, since it tends to result in software where either everything works or nothing works. In a specification document where there are loops in the uses hierarchy of the components of the internal model, the behavior of the components included in a loop is not well defined until they are all defined in detail, which complicates specification construction and makes it more difficult to understand individual component behav-

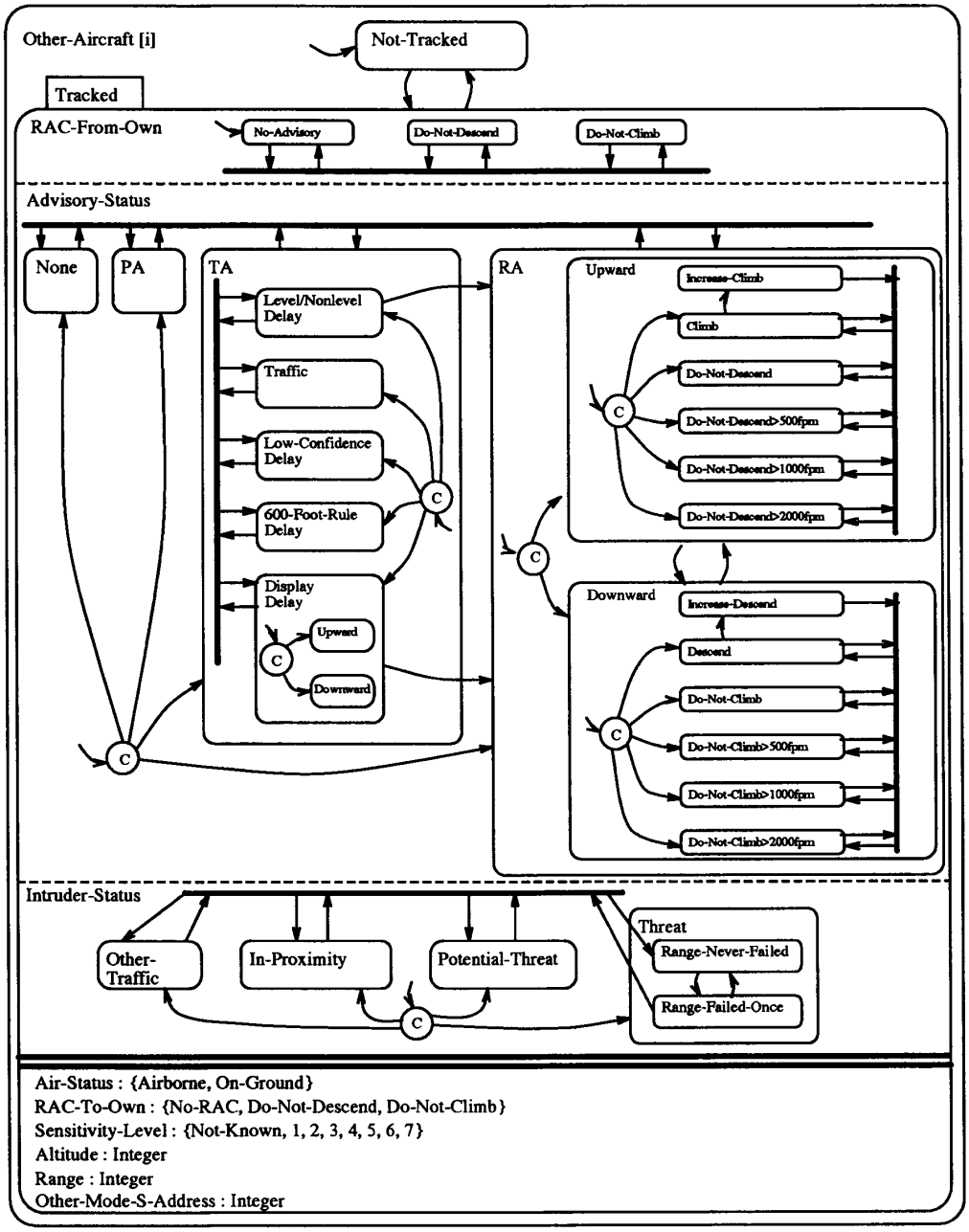


Figure 3: Our internal model of Other-Aircraft

ior.

Another issue that arose during our specification efforts involved the use of state variables. Although the statecharts notation allows the use of variables, no rules are given for making a decision about whether to model something as a state or as a variable. The use of variables is, in many instances, unavoidable. For example, it is usually impractical to model the notion of aircraft altitude as a set of states. But in other instances, such as intruder sensitivity level (see figure 3), a choice was necessary. Inclusion as a state makes the specification more understandable in that more is included in the pictorial representation of the model, yet it can lead to more states and thus decreased readability. We decided to represent as state variables all entities that change value only as a direct result of external events since the transition specifications for these variables were trivial. In our graphical notation, the variables associated with a component are listed below two solid lines at the bottom of the statechart modeling that component (see figure 3).

## 4 System Specification and Communication

Because of our goal of analyzing the requirements specification for correctness and safety, our model must include the behavior of the other components of the system and the external interaction between these components and the embedded software. Note that the behavioral description of the other components represents the assumptions about the behavior of the environment within which the software will execute. This information is required by the software designer in order to minimize the effects of changes in this environment on the software design and to design software that is robust against failures in the environment. It is also needed to perform a system safety analysis. Melhart [14] has defined an External Interaction Model (EIM) that employs statecharts to describe state information for all components and adds input and output exchange declarations, rules, and mappings for these exchanges. The input/output exchange declarations essentially represent an abstract specification of the communication or interface between the components. Safety analysis techniques are defined on the combined model (1) assuming no failures to determine worst case reachability of unsafe states and (2) assuming failures to generate system fault trees directly from the model.

Although Statemate [8] in the whole includes module specification and activity specification languages that allow specification of much of the information that we need, the use of multiple models, some of which are not formally defined, makes the resulting analysis too difficult (if possible at all). Instead, we have added the information that we need to the basic statechart model. This requires a notational distinction between regular statecharts and component statecharts and the modeling of limited communication channels between components since broadcast communication (the only form of communication defined in statecharts) is not an appropriate abstraction for communication between physically distinct components (e.g., two aircraft). Between components, only directed communication over logical channels is allowed.

In TCAS, system component statecharts are denoted by thick borders; communication channels are represented by arrows between the component statecharts. Figure 4 shows the overall system model for TCAS. The system consists of several components carried on the TCAS equipped aircraft (e.g., altimeter and a pilot-controlled switch), a collection of aircraft with different collision avoidance capabilities, and a set of ground stations. The direction of communication is indicated by the arrows connecting the components.

Each communication channel is connected to a component statechart by a socket. The socket specification will contain the information about characteristics of the input and output interactions for the component. It includes information such as message types, data values and data value ranges, timetypes (e.g., continual, periodic, stimulus/response), time ranges, destination (for outputs), capacity and load (i.e., maximum number of inputs or outputs that can be handled or sent per time unit), and exceptions. Exceptions are declared for inputs only and denote some mismatch between assumed behavior and actual behavior, e.g., values out of range indicate equipment malfunction. Exception transitions must be provided within the component to define the desired behavior upon violation of these declared assumptions. The EIM model also allows for mappings and rules to be specified to describe required characteristics of the input and output exchanges. Some of these can be used to check consistency in the static model while others can be used to derive dynamic consistency-checking code for the software.

Information about inputs and outputs to components are described in a tabular format. Figure 5 shows an example of a message specification.

Finally, input and output messages need to be connected to transitions in the statechart. By introduc-

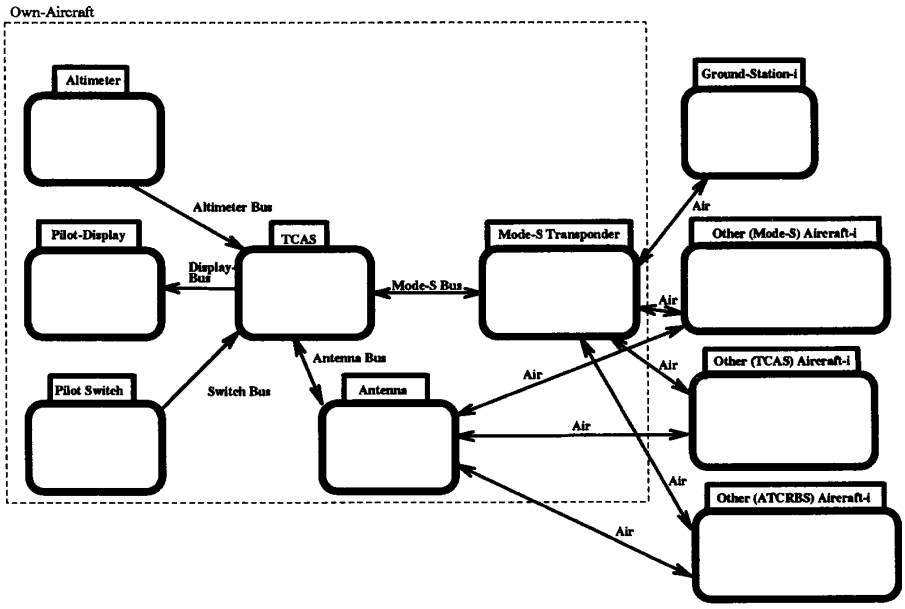


Figure 4: The configuration of the physical system

**Name:** Own-Altitude-Radar
**Message Format:** N/A

**Source:** Altimeter
**Destination:** TCAS
**Timetype:** Periodic
**Input Capacity:** One per second

**Type:** Integer
**Data Representation:** TBD
**Unit:** Feet
**Granularity:** 25
**Expected Range:** **Lower bound:** 0
**Upper bound:** 3000

**Exceptions:**Radar altitude may drop out at levels above 2000 ft.

Figure 5: The altitude reported from the altimeter

ing three new primitives to the language describing the transition predicates, directed communication can be modeled. The trigger (input) event *Receive(Socket-Name)* evaluates to true when a message arrives at a specific socket and the two actions *Send(Socket-Name, Data)* and *Read(Socket-Name, Data)* are used to send and receive data over a specific channel. In the TCAS specification, several different types of communication need to be modeled: The communication between Own-Aircraft and an intruder is of a completely different complexity and structure than the one between the TCAS component and the Pilot-Switch.

## 5 Notation

Readability and reviewability (i.e., the ability to easily answer questions about the system being modeled) are important goals for our specification. At the same time, a formal model is required for the safety analysis. We needed to resolve these potentially contradictory goals and spent considerable time and energy in developing a readable notation that maintained the underlying state machine model.

In devising a readable notation, the following four basic rules were followed:

- *Readability is more important than writability.*

Readability and writability are sometimes conflicting goals [4]. We have chosen readability in cases where we found a conflict; the added investment in writing the requirements specification pays off in terms of discovering more requirements-level errors.

- *Supply the necessary information* [4]. An often ignored piece of information is the context within which something new is described. This is one reason why information exposure is important — it is necessary to relate new information to what has already been seen.

- *Leave out irrelevant information* [4]. If information is of limited help at a particular point, then don't include it; try to help the reader focus on what is important.

- *Try to overcome individual preferences* [3]. When devising notations, we usually had ourselves in mind as the user. However, our familiarity with certain notations hides their weaknesses. Our first attempts were therefore failures: clear to us, but not to anyone else. Adopting this rule allowed us to look more objectively at what we devised. We sought outside feedback on our notation as it was being developed from those who were going to be using the final specification.

- *Use concise notations.* Choose a notation that represents the information as economically as possible while still maintaining readability.

Our first attempt to write the definitions for the state transitions used pure predicate calculus (figure 6), as this was what we had seen in previous statecharts examples [1, 6] and it was natural to us. Our external reviewers, however, did not find it natural or reviewable and told to us to come up with something else. In fact, we found that we had difficulty in writing and reading complex predicate calculus expressions ourselves even though we were familiar and comfortable with the notation; while developing another notation, we found scoping errors in our use of logical quantifiers that were not at all obvious in the original form.

Our second attempt replaced logical phrases with English phrases and a list of English-to-logic mappings. Although this is superficially more readable, we found that annotating the logic with English did not provide an appreciable advantage in terms of the underlying complexity of the logical expressions.

The notation we finally chose is a tabular representation of disjunctive normal form (DNF) (figure 7) that we call AND/OR tables. The far left column lists the logical phrases; the other columns are disjuncts of those phrases. If *one* of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements evaluate to true. To make all these relationships clearer, we physically separated the columns, the far-left column a little more than the others. The AND/OR tables do not eliminate the need for existential and universal quantifiers; however, their scope is limited to a disjunct term, making it much easier to parse the expressions.

The AND/OR tables have been used in expert reviews of the specification several times, and the response has been positive. Some evidence of the readability and reviewability of our notation is that errors we made in representing the system were quickly discovered by the experts after only a very minimal (five minute) tutorial on our notation. Below the AND/OR tables, we added an English language description of the guarding conditions on each transition. There is also a reference to the pseudo-code in the MOPS (minimal operational performance standard) implementing the transition (see figure 7). This reference provides traceability between the requirements and the design.

As we wrote the TCAS requirements, we discovered that some of the AND/OR tables became very complicated. Also, some of the logic is repeated in several tables. We solved both problems by using macros, which are abstractions into labeled AND/OR tables or functions. For example, the macro in figure 8 evaluates to true if any column in its AND/OR table evaluates to true. These macros, for the most part, correspond to typical abstractions used by the application experts in describing the TCAS requirements and therefore add to the understandability of the specification. We did, however, try to use them sparingly in order not to provide too many levels of indirection in the specification.

As we wrote the specification and participated in reviews, we found some other minor notational changes to standard statecharts useful. The advantage of statecharts over other finite state machine models is the ability to reduce the large number of states usually required in a simple finite state machine to a conceptually manageable number. Statecharts further reduce notational "clutter" by combining similar transitions. We found it helpful to introduce some constructs that reduced clutter even more.

First, several sections of our statechart model are fully- or almost fully-interconnected, i.e., there is a transition from each state to nearly every other one. Showing each transitions explicitly is confusing to the eye (figure 9a). We therefore devised a

**Transition(s):** 5 ⟶ 4
6 ⟶ 4
7 ⟶ 4

**Location:** Own-Aircraft ▷ Own-Sensitivity-Level$_{66}$

**Trigger Event:**

**tr**( ¬Climb-Desc.-Inhibit$_{56}$ ∧
( ( Own-Altitude-Radar > 400 ∧
Alt.-Radar$_{11}$ **in** Active ∧
( Own-Altitude-Radar ≤ 2150 ∧
( ( ∃i:Ground-Station$_{68}$[i] **in** 2 ∧ ∀j:Ground-Station$_{68}$[j] **in** x ∧ x ≥ 2) ∨
( Pilot-Switch$_{13}$ **in** 2) ∨
( ∀i:Ground-Station$_{68}$[i] **in** x ∧ x ≥ 4 ∧ Pilot-Switch$_{13}$ **in** 0))) ∨
( ∃i:Ground-Station$_{68}$[i] **in** 4 ∧ ∀j:Ground-Station$_{68}$[j] **in** x ∧ x ≥ 4 ∧ Pilot-Switch$_{13}$ **in** 0)) ∨
( Own-Altitude-Barometric ≤ 2150 ∧ Alt.-Radar$_{11}$ **in** Not-Active)))

**Output Action:** TBD

Figure 6: Predicate calculus

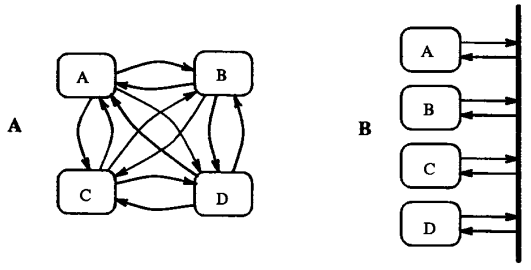


Figure 9: A fully interconnected statechart and the same statechart with the bus notation

"transition bus" (figure 9b), which expresses a fully-interconnected chart more economically. The semantics are as follows: a transition to the bus from state A and a transition from the bus to state B indicate that a transition exists from A to B. Although the number of arrows is about the same, they don't stretch all over the page.

Another problem arose with writing transition information on the arrows between states. This is fine for relatively simple transitions and relatively simple statecharts. Even marking the arrows with a short tag that identifies the transition logic elsewhere was found to complicate the graphics and make it more difficult to comprehend when the statechart was complex. Such tags are symbolic "noise"; the information is not salient, even when using supposedly mnemonic tags, and the resulting clutter is more harmful than helpful. We opted instead to put page references on each arrow, indicating where the transition logic could be found in the document (figure 10). Like many of our decisions, this was a compromise between showing important information but not too much information and reducing the number of synonyms and names that the user must remember. Paging through the document when reading transition definitions in order to view the corresponding statechart was minimized by including fold-out pages of the graphical part of the statecharts visible from anywhere in the document.

Page referencing was useful elsewhere in the document as well. No matter how concise the notational style, requirements specifications for large systems span many pages and usually contain references to other parts of the specification. We wanted to reduce redundancy while still making easily accessible all information that is needed to understand or review each part of the document. Liberal use of page references as subscripts on names (such as macros) defined elsewhere was a practical compromise (see figure 7).

**Transition(s):** 5 → 4; 6 → 4; 7 → 4

**Location:** Own-Aircraft ▷ Own-Sensitivity-Level$_{66}$

**Trigger Event:** Condition becomes

**Condition:**

| | | | *OR* | |
|---|---|---|---|---|
| | Alt.-Radar$_{40}$ **has value** | Active | Active | Not-Active |
| *AND* | *Own-Altitude-Radar* | >400 ≤2150 | >400 ≤2150 | · |
| | Own-Altitude-Barometric$_{45}$ **has value** | · | · | ≤2150 |
| | Lowest-Ground-Station **has value** | 2 | {4, 5, 6, 7} | · |
| | Mode-Selector$_{42}$ **has value** | · | 0 | · |
| | Climb-Desc.-Inhibit$_{56}$ | | | |

**Output Action:** TBD

**Comments:**

**Column 1** Own-Altitude-Radar is between 400ft and 2150ft, at least one Ground-Station has ordered TA-only, Climb and Descend are not both inhibited, and the Radio-Altimeter is working properly.

**Column 2** Own-Altitude-Radar is between 400ft and 2150ft, the Ground-Stations have all ordered a SL higher than 3, the pilot has selected Auto-SL, Climb and Descend are not both inhibited, and the Radio-Altimeter is working properly.

**Column 3** Own-Altitude-Barometric is below 2150ft, Climb and Descend are not both inhibited, and the Radio-Altimeter has dropped out.

**MOPS Ref.** Section 4.1.4, TRACK-OWN.Set-Index.

**Abbreviations:**

- **Lowest-Ground-Station in state x**

  **There exists an** i **such that** Ground-Station$_{89}$[i] **has value** x
  **AND**
  **For all** j Ground-Station$_{89}$[j] **has value** $\{x, \ldots, 7\}$

Figure 7: The AND/OR table

## Macro: Potential-Threat-Altitude-Test
**Definition:**

| | | *OR* | |
|---|---|---|---|
| *AND* | $\lvert Intruder\text{-}Relative\text{-}altitude \rvert < Altitude\text{-}Threshold$ | TRUE | · |
| | $Intruder\text{-}Relative\text{-}altitude\text{-}rate \geq Altitude\text{-}Rate\text{-}Threshold$ | · | FALSE |
| | $-\frac{\lvert Relative\text{-}altitude \rvert}{Relative\text{-}altitude\text{-}rate} < Time\text{-}Threshold$ | · | TRUE |

Figure 8: An example of a macro

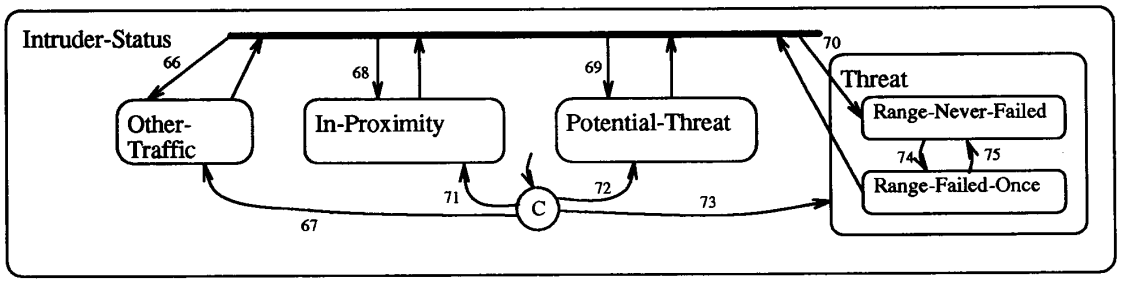


Figure 10: A statechart with page references to denote where the transitions are defined.

## 6 Conclusions

This paper has described some of the lessons learned and issues raised while building a model using statecharts of a real aircraft collision avoidance system. Having feedback from industrial experts has been invaluable in improving our specification techniques and our modeling language. Once the system requirements specification is completed, safety analysis procedures will be derived for the modeling language and evaluated using the TCAS testbed.

## References


[1] G. R. Bruns, S. L. Gerhart, I. Forman, and M. Graf. Design tecnology assessment: The statecharts approach. Technical Report STP-107-86, MCC, March 1986.

[2] S. Cha. *Design Chriteria for Safety Critical Software.* PhD thesis, University of California, Irvine, 1991. In preparation.

[3] Mike DeWalt. Personal communication during the fall of 1990.

[4] M. Fitter and T. R. G. Green. When do diagrams make good computer languages. *International Journal on Man-Machine Studies*, 11, 1979.

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

[6] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts (extended abstract). In *2nd Symposium on Logic in Computer Science*, pages 54–64, Ithaca, NY, 1987.

[7] K.L. Heninger, J.W. Kallander, J.E. Shore, and D.L. Parnas. Software Requirements for the A-7e Aircraft. Technical Report 3876, Naval Research Laboratory, Washington, D.C., November 1978.

[8] i Logix. The languages of statemate, March 1987.

[9] M.S. Jaffe, N.G. Leveson, M. Heimdahl, and B.E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transaction on Software Engineering*, 1991 (accepted for publication).

[10] N.G. Leveson. An outline of a program to enhance software safety. In W.J. Quirk, editor, *Safety of Computer Control Systems 1986 (SafeComp'86)*, Sarlat, France, October 1986. Pergamon Press.

[11] N.G. Leveson. Software Safety: What, Why, and How. *ACM Computing Surveys*, 18(2):125–164, June 1986.

[12] N.G. Leveson and J.L. Stolzy. Safety analysis of ada programs using fault trees. *IEEE Transactions on Reliability*, R-32(5):479–484, December 1983.

[13] N.G. Leveson and J.L. Stolzy. Safety analysis using petri nets. *IEEE Transaction on Software Engineering*, SE-13(3):386–397, March 1987.

[14] B.E. Melhart. *Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model.* PhD thesis, University of California, Irvine, July 1990.

[15] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Transaction on Software Engineering*, SE-5(2):128–138, March 1979.

[16] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2), 1974.