Using a Formal Description Technique to Model Aspects of a Global Air Traffic Telecommunications Network

J. H. Andrews
N. A. Day
Dept. of Computer Science, University of British Columbia
Vancouver, BC, Canada V6T 1Z4
tel (604)822-3061 fax (604)822-5485
{jandrews, day}@cs.ubc.ca

J. J. Joyce
Hughes Aircraft of Canada Limited
#200 - 13575 Commerce Parkway
Richmond, BC, Canada V6V 2L1
tel (604)279-5721 fax (604)279-5982
jjoyce@ccgate.hac.com

Abstract

Aspects of a draft version of the Aeronautical Telecommunications Network (ATN) Standards and Recommended Practices (SARPs) under development by ISO-compliant committees of the International Civil Aviation Organization (ICAO) have been mathematically modelled using a formal description technique. The ATN SARPs are a specification for a global telecommunications network for air traffic control systems. A version of Harel's statecharts formalism embedded within a machine readable typed predicate logic has been used as a formal description technique to construct this mathematical model. Our model has been 'typechecked' to partially validate the internal consistency of the

specification. The work described in this paper has already uncovered some problems in the draft SARPs, and will provide a basis for follow-on efforts to apply formal analysis methods such as model-checking and symbolic execution to aspects of the ATN SARPs. The success of this approach suggests that typed predicate logic is useful as a syntactic and semantic foundation for specialized Formal Description Techniques (FDTs).

Keywords

Formal description techniques, practical experience, extensions of FDTs, state transition systems, typed predicate logic, OSI application layer, verification and validation.

1 INTRODUCTION

This paper describes the modelling of aspects of the Aeronautical Telecommunications Network (ATN) using the formalism known as 'statecharts' (Harel, 1987) and predicate logic. This effort was performed by workers at Hughes Aircraft of Canada Limited (HACL), the University of British Columbia (UBC) and the University of Victoria (UVic). It is part of the FormalWare project, jointly funded by the BC Advanced Systems Institute, HACL, and MacDonald Dettwiler.

The ATN is a global system under development which will allow aircraft and ground stations to exchange data for the purpose of air traffic control. The various software components of the ATN reside in aircraft or ground station computers, and interact with human users and with each other to perform this data exchange. The communications protocols used by the software components are defined in ICAO documents referred to as Standards and Recommended Practices (SARPs) (SARP, 1996).

This modelling effort consisted of writing textual descriptions of components of the ATN using a formal description technique. There were two goals for this effort: first, to help validate that the SARPs protocols are safe (do not lead to deadlocks or livelocks, for instance); and second, to provide a formal description of the SARPs which can potentially act as a basis for validating implementations of the ATN. The first phase of the effort consisted of writing and typechecking an extensive draft of the model, and doing some informal validation; some problems in the draft SARPs were identified as a result of this work. This paper reports on the first phase, which was done in November and December 1996.

Among the more novel aspects of this work is the use of typed, predicate logic as a foundation for a more specialized formal description technique, namely a version of Harel's statecharts formalism. Although the 'semantic embedding' of specialized notations within typed, predicate logic is reasonably well-known by formal methods researchers, the effort reported in this paper provides some evidence of the practical benefits of this approach in addition to the more

theoretical benefits such as clarifying the semantics of specialized notations. By placing statecharts within a general-purpose environment, we were also able to integrate parts of the specification written in predicate logic itself.

This paper is organized as follows. Section 2 gives background on the tools and methodologies used. Section 3 explains the overall strategy for the modelling effort. Section 4 describes the simplifying assumptions that were made in creating the model to 'abstract out' implementation details. Section 5 discusses the assumptions that had to be made in order to deal with problems identified in the draft SARPs. Section 6 presents the results of the effort. Section 7 discusses the effort planned for future phases of the project. Finally, in Section 8 we review some lessons learned from this effort with respect to the use of formal description techniques.

2 TOOLS AND METHODOLOGIES USED

This section describes the tools and methodologies used in the modelling effort. The statecharts formalism (Harel, 1987) was used to describe the system in terms of parallel state decomposition and state-transition diagrams. 'S' (Joyce, 1994) is a formal description notation which we used to express statechart descriptions as well as other parts of the specification that are more suitably described in predicate logic. 'Fuss' is a typechecking tool for S specifications.

There were a number of factors which contributed to our not using more commonly-known tools such as the Concurrency Workbench (Cleaveland, 1989) or those available for Estelle, LOTOS and SDL (Turner, 1993). First, the specification that we were working from is a combination of text in paragraphs and an informal state transition model given in tables, with explicitly-named states, making the statecharts notation particularly suitable. Second, we wanted to do model checking, rather than simply discrete event simulation, to demonstrate properties of our formal models. Third, many of the available tools for model checking work from the system as a single finite state machine. The nature of the ATN means that the system expressed as a flat finite state machine would have millions of states. We did not want to exclude the possibility that the structure of a hierarchical specification can be exploited to reduce the size of the state space in analysis. Finally, the conditions under which state changes take place in the ATN are relatively complex, and we needed a general logical notation to allow us to express them naturally and accurately. This is where it was advantageous to use predicate logic itself.

Since we did not find this particular constellation of needs to be met by any one tool, we felt that it was most advantageous to us to use and extend a set of tools and methodologies with which members of the team had expertise. Analysis methods such as model checking are under development within this framework.

2.1 Statecharts

Statecharts are described by their inventor, David Harel, as a 'visual formalism' (Harel, 1987). There is precedent in the air traffic control industry for using such formalisms; TCAS II (Traffic Alert and Collision Avoidance System) was formally specified using the Requirements State Machine Language, a notation which is closely related to statecharts (Leveson, 1994).

In the statecharts formalism, a system is described in terms of states and transitions between those states. In this sense, it is like the 'state transition diagram' formalism. However, a statechart state can be more than a state in a state transition diagram.

A statechart state is either a 'basic state', an 'AND-state', or an 'OR-state'. Basic states correspond to the states of a state transition diagram. AND-states represent parallel composition and OR-states represent hierarchical state transition diagrams.

Although there are advantages to graphical representations of statecharts, especially for presentation, we decided to produce the initial model in a textual form, for reasons including portability and ease of integration.

Unfortunately, a number of semantically different versions of statecharts have emerged since the original informal description of the semantics of statecharts given by Harel. For this work, we have used a particular version of this formalism which has a formal semantics defined in a machine-readable format (Day, 1993). Whereas the behaviour of some statechart based tools are not clearly specified, these explicit semantics are used to directly initialize general-purpose analysis tools.

2.2 S - a machine readable notation based on typed, predicate logic

We used the formal description notation called S to represent statecharts textually. This made it possible to integrate the statechart parts of the model easily with predicate logic, as a means of specifying the details of complex state transitions, and with the text-based approach to requirements management followed at HACL. Through the use of parameterization, we were able to reduce the size of the specification and make it easily extensible without complicating the semantics of statecharts.

S allows us to declare elements such as types, constants, functions and predicates that are left 'uninterpreted'. This contrasts with software (as well as some 'simulation-oriented' formal description techniques) where ultimately everything must be refined, either manually or by means of a compiler, into bits and executable machine code.

Like several other formal description notations, S is based on typed predicate logic. However, in contrast to such notations as Z (which requires the use of an intermediate mark-up notation or other means of handling the specialized symbols and graphical presentation format of Z), it uses a more readable syntax for non-formal methods experts, which emphasizes letters and punctuation characters rather than symbolic characters. It also tends to use common English words (like 'function' and 'select') as keywords, rather than the technical terms (like 'lambda' and 'epsilon') used in other specification languages. Furthermore, the ASCII based syntax of S simplifies the mechanics of integrating formal descriptions into engineering documentation in contrast to notations which involve specialized symbols or graphical presentation formats.

2.3 Fuss

Fuss is a typechecking program for S, roughly corresponding to 'lint' for C. S is a strongly typed specification language, in the sense that the formal and actual parameters of a function call must be of exactly the same type, with no 'typecasting' allowed. In addition, functions can take other functions as parameters, types can be declared in terms of other types, and functions can be declared as taking different patterns of types. This expressiveness makes a rich hierarchy of types available to the user. The Fuss tool checks that the user's specification is well-typed, and also implements a 'type inference' algorithm in the style of the programming language ML, which infers (wherever possible) a precise type for any object for which we have not given an explicit type.

3 MODELLING STRATEGY

The overall strategy used was to model each software entity within the ATN, and each module and 'status' state within an entity, as a statechart state. This section first describes the structure of the ATN as presented in the SARPs, and then discusses how the various aspects of that structure were modelled using statecharts and S.

3.1 Structure of the ATN

The ATN is specified in the SARPs as a set of components interacting via messages. The components are not required to be implemented as separate processes or even as separate objects at the code level, but their behaviour must be consistent with the message-passing model.

Each component we modelled is further specified as a state transition system. Each transition between states is associated with a triggering event and a condition which must be true if the transition is to be followed when the event

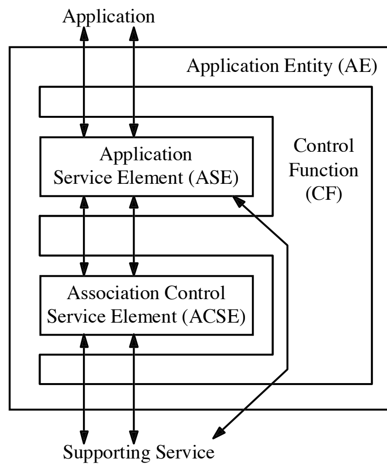

Figure 1: Internal structure of Application Entity

occurs. Each transition also has an associated action which is performed when the transition is followed. Typically, the trigger has to do with the message received and/or current variable settings, and the action is to send a message and/or change the variable settings.

The top-level components of the ATN that human users interact with are referred to simply as the *applications* or *user applications*. These do not communicate directly with each other; rather, they use a number of *Application Entities (AEs)* found in the OSI application layer to provide them with communication services. The four types of AEs modelled in the present effort are the *ADS* (Automated Dependent Surveillance), *CM* (Context Manager), *CPDLC* (Controller Pilot Data Link Communication), and *FIS* (Flight Information Service). There are two versions of each type of AE, a *ground* version which resides in ground stations and an *air* version which resides in aircraft. The AEs communicate with each other via the *supporting service*.

As shown in Figure 1, each AE in turn consists of three entities. The *Application Service Element (ASE)* performs the duty of receiving messages from the application and translating them into OSI-standard messages. The *Association Control Service Element (ACSE)* allows its AE to form associations with other ('peer') AEs. The *Control Function (CF)* mediates all communication amongst the ASE, the ACSE, the application and the supporting service. Each type of AE contains a unique type of ASE, but the CF and the ACSE are the same across all types of AEs.

The SARPs consist of on the order of 1000 pages of text, containing detailed specifications of the four types of ASEs and the CF, along with requirements on

the lower OSI layers and various less formal guidelines documents. The ACSE is described in a separate 40-page document, ISO 8650 (ISO, 1994).

3.2 Entities and Status States

The entities modelled were the CF, the ACSE, the CM and ADS ASEs, and the supporting service. The CF and the ASEs are all specified in the SARPs in terms of tables which informally describe a state transition system, as is the ACSE in its specification. The supporting service can also be expressed as a simple state transition system. Each entity was therefore modelled as a statechart OR state, and the resulting models were put together as an AND-state. The decomposition of the task into one state per entity also allowed the work to be distributed to workers and integrated more smoothly.

The CF, the ACSE, the CM ASE, and each module of the ground ADS ASE can be in one of several 'status' states (idle, associated, awaiting response, etc.) as defined in the SARPs. Each of these status states was modelled by a basic statechart state, and these basic states were put together in an OR-state to define the overall module or entity.

Figure 2 illustrates this top-level state decomposition with an example system. Dashed boxes within the large box represent the substates of the overall AND-state; the internal state transition structure of these substates has not been illustrated. The example consists of an air and ground CM AEs, the supporting service over which they communicate, and an 'environment' state Env which will be used to stand in for the applications using the AEs.

Most of the components have parameterized names to eliminate duplicate specifications of the same behaviour. This is similar to the use of procedure calls

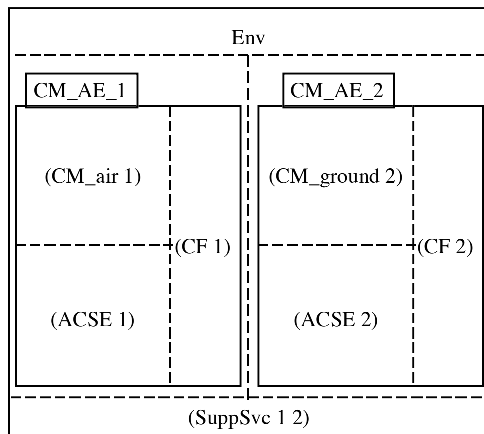

Figure 2: Statechart structure of example system

in a programming language. Thus, (CF 1) is the name of the CF state belonging to the first AE, and (SuppSvc 1 2) is the name of a supporting service connecting the first and second AEs. The text just gives a specification for (CF i) and refers to (CF 1), (CF 2), etc. without requiring new text to be written.

3.3 Transitions

Each module or entity in the SARPs is in one particular status state at any given time. It makes transitions between one status state and another depending on messages it has received, which are modelled as events, and on the results of tests that it makes. Each of these transitions was modelled by a transition in the statechart of the module or entity.

A large number of transitions is given in the SARPs for each entity, so most of the modelling effort went into formally defining these transitions.

3.4 Shared declarations

A file, called ‘sc.s’ and found in Figure 3, of shared declarations and definitions supports the modelling of statecharts in general. In sc.s, types are defined for statecharts, state names, transitions, and transition names. State names are declared separately from the state specifications. Messages are modelled as events. sc.s also defines term constructors which can be used to build up a statechart definition from basic states and transitions.

To support the task of modelling the SARPs, another S file named ‘atncommon.s’ was developed to contain declarations for the state names of the top-level ATN entities, and also for the ISO-standardized message types (e.g., ‘A-ASSOCIATE request’) used by all modules.

These two S files are ‘included’ by the S files containing the statecharts models in much the same way that a ‘.h’ file may be included by a software module written in C. They provided a common foundation for the half dozen individuals directly involved in the authoring of statechart models, allowing them to work with considerable independence during the initial phase of this project.

4 SIMPLIFYING ASSUMPTIONS MADE

One of the primary benefits of creating a formal model of a software/hardware system is that we can focus on high-level aspects of the system that we are interested in studying, and ‘abstract out’ implementation details. This process of abstraction consists largely of making simplifying assumptions about the system in order to clearly isolate the aspects we are modelling.

```
%% Type declarations

%% Basic types
: stateName;
: event;
: simpleEvent;  %% Used for "messages"
: action;
: transName;

%% Transition type
: trans == transName # stateName # event # action # stateName;

%% Statechart type
: sc := OrState :stateName :stateName :(sc)list :(trans)list
      | AndState :stateName :(sc)list
      | BasicState :stateName;

%% Constructor declarations

%% Expressions
InState: stateName -> bool;

%% Events
En: stateName -> event;         %% Entering a state
Ex: stateName -> event;         %% Exiting a state
Ev: simpleEvent;                %% Atomic event name
EvCond: event -> bool -> event; %% Event and condition
(_ And_e _): event -> event -> event;   %% Both events
(_ Or_e _): event -> event -> event;    %% One or the other event
Tm: event -> num -> event;      %% Event at given time
%% Receipt of message with data from stateName
(:A) Receive: stateName -> simpleEvent -> A -> event;

%% Actions
No_action: action;              %% Null action
Gen: simpleEvent -> action;     %% Generate message
(:A) (_ Asn _): A -> A -> action;       %% Assign var a value
Both : action -> action -> action;      %% Do both actions
%% Broadcast of message with data to all substates of stateName
(:A) Send: stateName -> simpleEvent -> A -> action;
```

Figure 3: ‘sc.s’ declarations for statecharts in S

In our case, we were primarily interested in studying issues to do with the sequences of messages sent between the various ATN entities. We wanted to examine whether the message protocol as defined in the SARPs is safe (that it does not lead to deadlocks or livelocks, that it is complete and consistent, and so on). We also wanted to provide a formal definition of the high-level structure of the ATN and its protocols, which could be used as a basis for developing and testing the actual software. The simplifying assumptions we made reflect this focus:

- We assumed that the supported service was stable and error-free.
- We assumed that the translation of the data by the various entities did not affect the safety properties of interest to us, and therefore did not need to be modelled.

- Timers are often specified in the SARPs for such purposes as timing out dropped connections. We modelled timeouts of timers as messages sent from the environment, which could be sent at any time, rather than actually modelling time. For instance, a statement in the SARPs which specifies that a timeout will occur 30 seconds after a particular event will be modelled more generally as a timeout that could occur at any time.

Note that the simplifying assumptions were made not because we felt unable to manage the extra details; rather, they were made because in our judgement the extra details were not relevant to the properties we are trying to validate. We may discover that we cannot validate some property because some necessary detail is missing. If this does happen, we will then add the missing detail to the model. However, as long as our simplifying assumptions hold, then any property derivable from the formal description should also be true for the SARPs.

5 DISAMBIGUATING ASSUMPTIONS AND PROBLEMS WITH THE SARPS

In contrast to simplifying assumptions as just discussed, we also found it necessary to make additional assumptions which we have called 'disambiguating assumptions'. From a logical point of view, these assumptions are less 'safe' than simplifying assumptions in that we are not merely shaving away irrelevant detail. With disambiguating assumptions, we are adding necessary detail to the model that may or may not have been intended by the authors of the SARPs.

The effort reported here has revealed some ambiguities and lack of clarity concerning the handling of error conditions (for instance, when messages are received out of sequence) in the draft SARPs. The SARPs give somewhat ambiguous recommendations about what to do in a given error situation. For example, in the specification of the CF, the only substantial passage concerning the behaviour of the CF when a message is received out of sequence is the following: 'The error handling shall result in the association being aborted, if one exists, and a notification being given to the Application user.' This passage makes no mention of the fact that the CF must inform three different entities (the ASE, the ACSE, and the peer AE) of the abort of the connection, it does not describe the sequence or format for these messages, and it does not specify how notification of the abort is to be given to the Application user.

Because the SARPs were ambiguous, the people writing the formal specifications were not able to come up with a model which corresponded unambiguously to the SARPs. Implementations would have the same difficulty. It is observed by experts in software safety that software intensive systems often perform well when operating under normal conditions, but not when operating under unusual or error conditions. There is the potential in the SARPs that

protocol errors which go undetected during validation will cause silent aborts of connections, error cascades, or similar problems. For instance, when the ACSE detects a malformed message, it is supposed to send an abort request both to its user and to its peer; but as soon as the CF passes on the first abort message, it goes into a state in which all subsequent abort messages from the ACSE are treated as protocol errors. Because of this, there is the possibility of a further error report from the CF, and the possibility that the peer will not know about the abort of the association. This in turn will at least cause other error conditions, and may have more serious consequences such as error cascades.

Hence we have had to make some disambiguating assumptions about what the SARPs mean. Our group developed an interim strategy for dealing with error conditions to allow the development of the formal model to continue. Since the modelling effort was done, a new version of the SARPs was released around December, 1996. The particular problem noted in the last paragraph remains in the latest version; we have communicated our concerns to the ICAO committee responsible for the development of the SARPs. When these problems are resolved, it should be reasonably straightforward to modify our model based on the resolutions provided.

6 RESULTS OF EFFORT

Component	*# of states*	*# of trans*	*# of vars*	*Prior worker knowledge*	*Worker hours*	*Lines of S text*
CF	5	96	2	Very high	24	680
ACSE	7	123	2	High	20	400
CM	13	201	2	High	44	1790
Ground ADS	22	232	1	Moderate	60	2850
SuppSvc	1	10	0	High	5	50
Misc Support					32	150
Total	48	662	7		185	5920

Figure 4: Results of effort

Figure 4 shows the results of the effort, in terms of the number of hours spent and number of lines of S text produced, according to the number of states, transitions and variables in the given component and the prior worker knowledge of the S formalism. The '# of states' in the column is the number of statechart states. It is not a measure of the complexity of the state space for analysis, but rather a rough measure of the inherent complexity of the specification. The number of hours also includes the time taken to perform static checks for completeness and consistency and integration. All workers were graduate students or faculty; 'very high' prior knowledge means knowledge of S in

```
% A normal transition for the ACSE (no conditions).
% Normally called as "inMessage.(ACSE_TRANS ...)" in
% order to emphasize message.
ACSE_TRANS i sourceState outMessage
           (destState: stateName -> stateName)
           inMessage :=
  ( (PTrans ((ACSE i).sourceState) inMessage),
    ((ACSE i).sourceState),
    (Receive (CF i) inMessage (ACSEData i)),
    (Send (CF i) outMessage (ACSEData i)),
    ((ACSE i).destState)
  );
```

Figure 5: A customizing declaration in S (from ACSE model)

particular, 'high' means knowledge of typed logic but not S in particular, and 'moderate' means only knowledge about first order logic.

Figures 5 and 6 show some sample text from the resultant specification. Figure 5 shows a typical 'customizing' declaration in S; like a declaration of an auxiliary function in a programming language, this declaration allows the rest of the specification to be more compact. The function 'ACSE_TRANS' maps five parameters, 'i', 'sourceState', 'outMessage', 'destState' and 'inMessage' to an instance of a transition denoted by a 5-tuple of the form (transition label, source state, event/condition, action, destination state). Figure 6 shows a typical section of the specification of the ACSE which lists the transitions from a particular state. In the definition of 'Transitions_From_Awaiting_AARE', 'ACSE_TRANS' is used within a let-definition to introduce a local name for a function called 'TRANS_CELL'. In the let-definition of 'TRANS_CELL', the function

```
Transitions_From_Awaiting_AARE i :=
    /* From Awaiting_AARE state (STA1) */
    let Error_Cell := (ACSE_error i Awaiting_AARE) in
    let TRANS_CELL := (ACSE_TRANS i Awaiting_AARE) in

    [ /* Making connection */
      A_ASSOCIATE_req     . Error_Cell;
      A_ASSOCIATE_rsp_pos . Error_Cell;
      A_ASSOCIATE_rsp_neg . Error_Cell;
      P_CONNECT_ind       . Error_Cell;
      P_CONNECT_cnf_pos   .
          (TRANS_CELL A_ASSOCIATE_cnf_pos Associated);
      P_CONNECT_cnf_neg   .
          (TRANS_CELL A_ASSOCIATE_cnf_neg Idle);
      /* Releasing connection normally */
      A_RELEASE_req       . Error_Cell;
      A_RELEASE_rsp_pos   . Error_Cell;
      A_RELEASE_rsp_neg   . Error_Cell;
      P_RELEASE_ind       . Error_Cell;
      P_RELEASE_cnf_pos   . Error_Cell;
      P_RELEASE_cnf_neg   . Error_Cell;
      /* Releasing connection abnormally */
      A_ABORT_req         . (TRANS_CELL P_U_ABORT_req   Idle);
      P_U_ABORT_ind       . (TRANS_CELL A_ABORT_ind     Idle);
      P_P_ABORT_ind       . (TRANS_CELL A_P_ABORT_ind   Idle)
    ];
```

Figure 6: Typical section of ACSE model

‘ACSE_TRANS’ is partially evaluated when it is applied to two values, ‘i’ and ‘Awaiting_AARE’, as arguments for the first two of the five parameters of ‘ACSE_TRANS’. This yields a local function, ‘TRANS_CELL’ which is used to denote transitions that always originate from the state ‘Awaiting_AARE’. ‘TRANS_CELL’ is parameterized by the remaining three parameters of ‘ACSE_TRANS’, namely, ‘outMessage’, ‘destState’ and ‘inMessage’. This use of functions results in a more concise, and potentially easier to understand description. Our use of the S notation provides much the same expressiveness as a general-purpose functional programming language.

Models have been completed for the CF, the ACSE, the air and ground CM ASEs, the supporting service, and part of the ground ADS ASE, incorporating five out of the seven ADS modules defined in the SARPs. The amount of effort required to integrate any new ASEs into the model should be minimal.

We have successfully integrated the CM ASE models with the CF, ACSE, and supporting service, to the level of typechecking. The resulting statechart specification models an air CM AE (consisting of an air CM ASE, a CF, and an ACSE) talking to a ground CM AE (consisting of a ground CM ASE, a CF, and an ACSE) via the supporting service (see Figure 2). When later ASE models are developed, they should be able to be easily added to the specification and re-use much of the existing specification through parameterization.

The integrated system has passed the typechecking of the Fuss tool. This indicates that most interface errors have been eliminated, although it does not allow us to conclude that the models are completely correct.

Some static checks have been performed for the ACSE and CF. These are of two types: completeness checks and consistency checks. The completeness checks are intended to ensure that for each message received by a component of the model, there is at least one transition that will be followed regardless of global variable settings. The consistency checks are intended to ensure that not more than one transition can be followed for each combination of messages received and global variable settings. These checks were carried out by visual inspection. It would be useful to have a tool to do this analysis. Previous efforts at checking the completeness and consistency of state-based models (Heimdahl, 1996; Heitmeyer, 1996) rely on a tabular specification of the transition triggers.

7 FUTURE EFFORT

Two additional phases of the project are planned for the future. Phase 2 consists primarily of effort to adapt/develop a model checking tool as necessary to demonstrate properties of the statechart model. Phase 3 consists of effort by the team as a whole to do the more extensive validation. We also expect our model to be maintained in order to track changes in the SARPs.

At the present time, the only tool available to support the modelling effort is the Fuss typechecker. The second author's PhD thesis research examines how to analyze specifications consisting of integrated components in different notations (such as statecharts and predicate logic), and how to automatically analyze specifications at a high level of abstraction. The SARPs statechart model serves as test data for this effort; the other workers will interact with her to clear up any problems that may arise from the models.

As part of a research collaboration involving two universities and two industrial organizations, the work described in this paper is being used as the basis for a variety of research oriented investigations. Members of the FormalWare project are using, or are expected to use, this example as a case study for the development of methods and software tools for purposes such as automatic test case generation, symbolic execution and possibly code generation. In many cases, the parsing and typechecking functionality of Fuss is used as a front-end for the implementation of software tools which use S as input. This is easily achieved since Fuss is designed specifically to support user developed extensions which access the internal representation of an S specification created by Fuss.

8 LESSONS LEARNED

The work described in this paper represents the results of using an integrated approach to specifying a model. Using a general-purpose formal description notation (S) as the basis of the entire project, we built models of the components of the ATN based on the statecharts formalism, and laid the groundwork for building analysis tools using S as input. We conclude that this integrated approach has indeed been useful.

8.1 Usefulness of a general-purpose formal description notation

Using a general-purpose formal description notation has been valuable. The alternative would be to use a specialized notation for state-based applications, such as pure statecharts. Our approach allowed us to integrate a state-based formalism with predicate logic to express the complex conditions on transitions. We were also able to use uninterpreted constants to maintain a level of abstraction, and parameterization to reduce duplication.

A future goal of the project is to extend the range of current analysis methods to integrated requirements specifications given in multiple notations and at a high level of abstraction, such as those containing uninterpreted constants. In this paper we have demonstrated that a general-purpose notation can serve as a foundation for expressing specialized notations and integrating notations. Future analysis work will take advantage of the fact the semantics of the specialized

notation can also be expressed in the same framework (Day, 1993). This means that we are not locked into a specialized notation for specification and analysis.

8.2 Usefulness of S in particular

S has been particularly appropriate as a general-purpose formal description notation because (a) it is strongly typed and has an associated typechecker, Fuss; (b) it is machine readable; (c) it is more human-readable than many more symbolic notations; and (d) its power and generality allow a good deal of flexibility in how the model components are expressed.

Another important benefit of using typed, predicate logic was the ability to build a layer of infrastructure (i.e., the S file 'atncommon.s' mentioned earlier) on top of the specialized FDT which tailors our use of statecharts specifically to the purpose of modelling aspects of the SARPs.

Finally, our choice of S as the foundation for our approach made it possible to use Fuss 'off the shelf ' for this integration task.

8.3 Classification of assumptions

This work also led to a better appreciation of the distinction between the role of 'simplifying assumptions' and other kinds of assumptions made in the development of a formal representation, such as the 'disambiguating assumptions' made to address aspects of the SARPs which were found to be ambiguous or unclear. There is a natural tendency to regard any kind of modelling simplification as something that may undermine the validity of results derived from the model. But we have used the term 'simplifying assumptions' to describe aspects of our formal representation which, in effect, increase the generality of these results rather than undermining their validity.

9 ACKNOWLEDGEMENTS

In addition to the authors, a number of individuals contributed to this work, including: from HACL, Ayman Farahat, Alec MacKay, Ofelia Moldovan, Greg Saccone and Robert Taylor; from UBC, Kendra Cooper, Michael Donat, Ken Wong; and from UVic, Dilian Gurov and Bruce Kapron. Michael Donat provided helpful comments and corrections on earlier versions of this paper.

10 REFERENCES

Cleaveland, R., Parrow, J. and Steffen, B. (1989) A Semantics-Based Verification Tool for Finite State Systems, in *Proc. 9th IFIP Symposium on Protocol Specification, Testing and Verification*, North-Holland.

Day, Nancy (1993) A model checker for statecharts, M.Sc. thesis, Department of Computer Science, University of British Columbia, Technical Report 93-35.

Harel, David (1987) Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**, 231-274.

Heimdahl, Mats P.E. and Leveson, Nancy G. (1996) Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering,* **22(6)**, 363-377.

Heitmeyer, Constance L., Jeffords, Ralph D. and Labaw, Bruce G. (1996) Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, **5(3)**, 231-261.

ISO (International Organization for Standardization) (1994) ACSE Protocol, ITU-T Rec. X.227 -- ISO/IEC 8650-1: Edition 2. Available in electronic format via anonymous FTP at URL 'ftp://ftp.stel.com/ pub/atnp2/iv/P2'.

Joyce, J., Day, N. and Donat M. (1994) S: A machine readable specification notation based on higher order logic, in *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, 285-299.

Leveson, Nancy G., Heimdahl, Mats P. E., Hildreth, Holly and Reese, Jon D. (1994) Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, **20(9),** 684-107.

SARP (1996) Aeronautical Telecommunication Network Panel. Draft. Available via anonymous FTP at URL 'ftp://ftp.stel.com/ pub/atnp2'.

Turner, K. J. (ed) (1993) Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL. Wiley.

11 BIOGRAPHIES

Jamie Andrews is an Assistant Professor in the Department of Computer Science, University of Western Ontario. This work was carried out while he was a post-doctoral fellow at the University of British Columbia (UBC) working with Paul Gilmore and Jeff Joyce.

Nancy Day is a PhD student in the Department of Computer Science at UBC. She expects to complete her dissertation near the end of 1997.

Jeff Joyce is a Research Scientist at Hughes Aircraft of Canada and an Adjunct Professor in the Department of Computer Science at UBC.