Formalization and Analysis of the Separation Minima for Aircraft in the North Atlantic Region

Nancy A. Day
day@cs.ubc.ca
Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC, Canada V6T 1Z4

Jeffrey J. Joyce, Gerry Pelletier
{jjoyce,gpelletier}@ccgate.hac.com
Hughes International Airspace Management Systems
13951 Bridgeport Rd,
Richmond, BC, Canada V6V 1J6

Abstract

The formalization and analysis of an air traffic control separation minima serves in this paper as an illustration of an approach that uses formal operational semantics to drive the automated analysis of specifications. This contrasts with the approach of translating one notation into the input format for an analysis tool, or hard-coding the semantics of a particular notation into the implementation of an analysis technique.

The semantic functions capture the structure of the specification and can be directly evaluated to map a notation to a rigourous mathematical foundation. This work contributes to a greater appreciation of how the structure of a specification (e.g., the organization of a table), not just the semantics, is an important input to many analysis functions. Building upon a common mathematical foundation, different notations can be combined to support an integrated approach to the analysis of a formal specification. A related issue is the importance of being able to reverse the effect of the semantic functions so that analysis results are provided to users at the same level of abstraction used in the input specifications.

The formalization of the separation minima combines the use of a tabular style of specification with predicate logic. This paper discusses how automated analysis functions were applied to the specification to check for the properties of consistency, completeness and symmetry. The benefit of doing this analysis is demonstrated by the discovery of an ambiguity in the separation minima.

1. Introduction

This paper describes work carried out at the University of British Columbia in collaboration with Hughes International Airspace Management Systems to formalize and validate a specification of the separation minima for aircraft in the North Atlantic (NAT) region. Our formalization is based on a description provided in a source document published by Transport Canada on behalf of ICAO[1]. This document describes the official North Atlantic Separation Minima as published by ICAO. This description provides guidance to air traffic controllers managing the region of oceanic airspace between Europe and North America. It is also used as the basis for the development of software based systems that support the management of the NAT region. For example, it would be used during the planning of a flight from New York to London to check whether the route is free from separation conflicts with other aircraft expected to be in the NAT region at the same time.

This collaboration has directly involved domain experts (not just formal methods experts) in the process of developing and analyzing a formal representation of a complex "real" description. The source document is an informal specification that has been scrutinized by the NATSPG (NAT Systems Planning Group) members who are ATC specialists from the NAT countries, and most of them maintain and use automated systems that implement these rules.

Our formal representation of these separation minima is given in a mixture of a tabular style of specification and a variant of higher order logic called "S" [11].

[1]This document, "Application of Separation Minima for the NAT Region" (3rd edition, effective December 1992), was published by Transport Canada on behalf the ICAO North Atlantic Systems Planning Group. ICAO is the International Civil Aviation Organization with headquarters in Montreal, Canada. This separation minima document was developed by the COMAG (Communications and ATM Automation Group), now called the CADAG (Communication, Automation and Data Link Applications Group).

Combining multiple notations makes it possible to choose the notation best suited to the various parts of the specification. The tabular style was chosen because the rules consist of complex decision logic describing predicates and functions. To unite the various parts, including environmental assumptions, in a common framework for analysis, the tables are considered a "style" of specification in the S notation. A "style" of specification includes associated semantic functions for the constructs that are introduced, which makes it possible to capture the structure of the specification and give meaning to the notation.

Once in a common environment, the specification can then be analyzed for various properties. These range from "style" independent properties, such as typechecking and symmetry, to properties particular to the notation being used. This paper describes the analysis of the completeness and consistency of the tabular specifications. Symmetry is a particularly desirable property of the separation minima, so we also describe how the same analysis mechanisms used for completeness and consistency are used to do this check. The most significant result of the analysis was the discovery of two tables in the specification with inconsistencies, where, for the same scenario, the specification indicated two different amounts of aircraft separation.

The formal specification and related analysis results can be found on-line at http://www.cs.ubc.ca/spider/day/Research/SeparationMinima/SeparationMinima.html .

This work tests the doctoral thesis hypothesis of the first author, Day. This hypothesis is that explicit definitions of the operational semantics of a notation can be used directly in the analysis and that this method retains the domain knowledge captured by the structure of the specification, which can be exploited in analysis. This structure can be used to help convert the specification to a finite model and to determine the correct level of abstraction for presentation of the results of analysis. This paper outlines the framework for using operational semantics for analysis. We have implemented this framework, including the analysis techniques described in this paper in a tool called Fusion. The overall goal of this work is to make it possible to perform fast, automatic, lightweight checks to streamline the validation of specifications. The automated checks do not provide absolute assurance, but they isolate details that can be reviewed independently. The individual analysis checks described here usually took about 1 second of execution time on a Pentium-120 with 16 MB running Linux.

2. Related Work

2.1 Notation

Since the separation minima is a specification of combinations of conditions that produce different outcomes, a tabular style of specification seemed suitable. Previous successful efforts of using tables provide a good precedent for the readability of a tabular style of specification. These efforts include the AND/OR tables of the TCAS II project[12] and the Software Cost Reduction (SCR) notation used in the A-7 aircraft Operational Flight Program[9]. Initially, we considered using either AND/OR tables, or the style of tables presented by Parnas [16]. SCR tables are typically for system specifications that involve "modes" of operation and the separation minima does not have this characteristic.

An AND/OR table consists of a series of rows labeled by predicates. The columns to the right of the label contain "T" for true, "F" for false, or "." for "don't care". The cell is meant to represent the case where the condition given by the label is true or false. A "don't care" value means that the cell could contain either true or false. The table represents a predicate that is true if the conjunction of the cells in any column results in true.

A difficulty with AND/OR tables is that they only represent predicates. In the separation rules, sets of conditions are used to describe cases for different return values of functions.

The other approach considered was the tabular style presented by Parnas [16] which allows for the grouping of related conditions along a row. Grouping is achieved by allowing each different argument of the predicate (or function) represented by the table to have its own dimension. Hence, this style is best suited for capturing functions of a small number of dimensions which is not the case for the tables we expected to construct in our formal representation of the NAT separation minima.

2.2 Analysis

In the TCAS II specification, a table can be used to describe the condition for taking a transition in a state machine. In the completeness and consistency analysis carried out by Heimdahl and Leveson[6], the specification is considered complete if a transition is always enabled from a state. It is consistent if the specification is deterministic, i.e., if no two transitions can be enabled at the same time.

In the TCAS II AND/OR tables the cells in the rows can contain only true, false, or “don’t care”. In the analysis, a Boolean variable is associated with each row label. The meaning of each cell in the row is the condition of whether this Boolean variable is true or false. This allows for an efficient implementation using Binary Decision Diagrams (BDDs) [2]. Completeness analysis checks that the disjunction of the columns of all the tables used to describe transitions from a given state is a tautology. Consistency analysis checks that there is no overlap in the conditions between multiple tables describing transitions from the same state, i.e., the conjunction of the meaning of two tables is a contradiction. Checking if the BDD representation of an expression is a tautology or a contradiction takes constant time.

A difficulty with AND/OR tables is that related conditions such as “x < 280” and “x > 450” are listed on separate rows and therefore the structure of the table does not capture the relationship between these terms. Related conditions are associated with different Boolean variables. This can result in the analysis producing false negatives. For example, it might return a bogus result indicating that no table covers the case where both the conditions “x<280” and “x>450” are true. The tool created by Heimdahl and Leveson catches false negatives with respect to enumerated types, but not those arising from the use of mathematical functions. They are investigating linking their analysis with a theorem prover [6].

Although SCR tables are not applicable to the separation minima, their analysis techniques are relevant. Heitmeyer, Jeffords and Labaw [7,8] describe work on checking the completeness and consistency of condition tables given in the SCR notation. They also define completeness as a coverage property - that the disjunction of the conditions in a row is a tautology. Currently, they limit themselves to conditions ranging over Boolean values or those that have been converted by hand to Boolean variables. Expressions involving relations are also converted manually into Boolean variables. They are working on techniques to reason about conditions involving mathematical functions. Their analysis of the condition tables for the Operational Flight Program of the US Navy’s A-7 aircraft found 17 legitimate errors in 36 tables with a total of 98 rows. Two false errors were found due to their strictly Boolean interpretation of the specification. Given the manual encoding to Boolean variables, these results must have been mapped by hand back to the correct level of abstraction for interpretation.

Both of these previous examples are control-oriented systems. In systems that contain a great deal of data complexity, such as these separation minima, it becomes more important to capture and utilize the relationships among data values.

Our work is similar to that carried out by Owre, Rushby, and Shankar where they have added a table construct to the PVS theorem prover[14,15]. The theorem prover is used to address some of the deficiencies of a strictly BDD-based approach. They follow the same approach as us of semantically embedding decision tables within higher order logic. The checks for completeness and consistency are carried out by proving type correctness conditions for the tables. This requires minimal theorem proving effort when the tables are complete and consistent but it appears that some effort is required to extract the cases not covered or the inconsistent cases.

Our effort documented here also attempts to address the difficulties of a strictly BDD-based approach while staying within the realm of lightweight, fully automatic techniques. In particular, we show how the structure of the table often can be utilized to eliminate the need for a more heavyweight tool such as a theorem prover. Our approach to executing the semantic definitions using symbolic functional evaluation as is done in functional programming languages also differentiates this work. Finally we demonstrate the value of including environmental constraints in the analysis process and present a simple approach for dealing with quantification to make this possible.

3. Specification Notation

At the beginning of this project, the first author was presented with an interpretation of the separation minima expressed as pseudo code (a draft documented dated 20 Sept 95). This interpretation was created by a third party to provide software developers with an algorithmic interpretation of the English text and diagrams contained in the NAT region separation minima specification. This pseudo code imposed an order of evaluation on the conditions as well as other implementation details. The imperative programming style of specification used involves

					Default
A.FlightLevel	_ <= 280	.	_ > 450	_ > 450	
B.FlightLevel	.	_ <= 280	_ > 450	_ > 450	
IsSupersonic (A)	.	.	_ = T	.	
IsSupersonic (B)	.	.	.	_ = T	
VerticalSeparationRequired (A,B)	**1000**	**1000**	**4000**	**4000**	**2000**

Figure 1: Vertical Separation

assignments of default values to variables followed by if-then-else statements to modify these variables, as well as procedure calls. Most of the conditions of the if-then-else statements were expressed in terms of English phrases.

Our goal became to formalize the separation minima using a notation that did not impose implementation constraints and that was amenable to analysis so we could determine which cases were being covered by the default values. This effort also required sorting out the variety of English phrases used to describe various conditions to yield a "dictionary" of primitives that were then introduced in the formal representation as uninterpreted functions and predicates.

Our review of previous work using tabular specifications led to the use of a variation of AND/OR tables. This variation allows related conditions to be captured within a row in a style closer to the idea of decision tables given in the structured analysis methodology of DeMarco [3]. A row isolates one dimension of the decision and the columns relate the different dimensions to produce a case. A table can also represent functions through the addition of a row of return values.

Figure 1 is an example of our tabular style of specification; this particular table specifies the minimum vertical separation (in feet) that must exist between two aircraft for them to be considered separated in the NAT region. The name of the function and the arguments to the function are given in the last row of the table which gives the return values of the function. Except for the last row of the table, the label of a row (the leftmost column) is an expression. The cells of the row are predicates that can be applied to this label to produce the condition that the cell represents. The parameter of the predicate is given by the "_" in the cell. A "." means "don't care", i.e., the predicate is always true. Then, as with AND/OR tables, the conjunction of the cells in any column is the case where the function returns the value in the last row for that column.

A semantically equivalent representation in S of the function given in Figure 1 is:

```
VerticalSeparationRequired(A:flight, B:flight) :=
        if (A.FlightLevel <= 280) then 1000
        else if (B.FlightLevel <= 280) then 1000
        else if ( (A.FlightLevel > 450) AND
                  (B.FlightLevel > 450) AND
                  (IsSupersonic (A) = T) )  then 4000
        else if ( (A.FlightLevel > 450) AND
                  (B.FlightLevel > 450) AND
                  (IsSupersonic (B) = T) ) then 4000
        else 2000;
```

The arguments A and B represent flights. In S, the "dot" notation as used in the expression "A.FlightLevel" is merely syntactic sugar for function application. "A.FlightLevel" is interpreted by an S parser as "FlightLevel (A)" to allow for the representation of static information about an item in the familiar "record" type of notation.

In addition to standard, "built-in" predicates such as "<=" and ">", the formalization also involved the introduction of uninterpreted types and constants. An uninterpreted constant has a type but no definition. For example, the following S declarations introduce an uninterpreted type[2] ("flight"), an uninterpreted function ("FlightLevel") and an uninterpreted predicate ("IsSupersonic"):

```
:flight;
FlightLevel : flight -> num;
IsSupersonic : flight -> bool;
```

[2] Uninterpreted types are analogous to "basic types" in a Z specification[17].

The use of uninterpreted terms allowed us to phrase the specification in domain terminology and to match the level of abstraction appropriate for this specification.

The table for vertical separation in Figure 1 specifies a function. In the case of a predicate (i.e., a function that returns a Boolean value) the bottom row of return values can be omitted and the cases designated by the columns are assumed to return true for the predicate. Any other cases are assumed to return false.

These tables also allowed us to match the modular nature of the decomposition of the separation minima. For example, longitudinal separation between two aircraft that are both turbojet depends on the current airspace (MNPS or WATRS[3]) of the aircraft. These cases are given in separate tables.

Since we were working within the S environment, we were also able to use definitions of functions in predicate logic rather than tables in some cases. These function could reference tables and tables could reference functions defined in S. This illustrated the benefits of being able to combine multiple notations.

4. Formalization Process

Figure 2 illustrates our formalization and analysis process. An important element in gaining industry acceptance of any formal method is the form in which the specification is presented for review to non formal methods experts. We chose to use HTML so that changes to the specification could be quickly viewed by all authors and so that cross references in the document between the use and definition of terms could be given using hyperlinks. This made it possible to give supplementary text to describe the formal tables. The top-down presentation given by this document also has advantages over the bottom-up order (i.e. declaring or defining terms before they are used) which is expected by analysis tools. However, it was also necessary to have a version of the specification that could be input to the analysis tools. To eliminate the difficulty of having to maintain versions of the specification existing in different forms, we created one document that is a mixture of HTML and formal notation. We used a preprocessor that produces the specification in pure HTML (using HTML tables for the formal tables) and automatically generates links from references to the declarations and definitions of terms[4]. It also produces a separate file containing only the representation in formal notation in the correct order which is used as input to the analysis tool.

The first draft of the formal specification was created by the first author based on a pseudo code representation of the separation minima. The first step in the formalization process was to determine the primitives of the specification and introduce these as uninterpreted functions and predicates in S. The pseudo code is modular so parts of it can be matched to individual tables. For each table, the relevant "inputs" were determined and used as the labels for the table rows. Columns in each table were then created as given by the logical combinations of these inputs. Typechecking was used to validate the first draft of our formal representation of the separation minima.

The first draft of the specification was handed off to the third author who is the domain expert. The only explanation of the tables that he was given was one paragraph of text with an example at the beginning of the document. From this, he edited the HTML version of the document and supplied the first author with a revised draft of the specification. This included smaller changes, such as terminology, and more meaningful changes, such as clarification of ambiguous phrases identified by the first author, such as, "a portion of the routes of both aircraft are within OR above OR below MNPS airspace". The correct interpretation of this phrase is that each aircraft is considered independently with respect to MNPS airspace, as opposed to both aircraft having to be within MNPS airspace or both having to be above MNPS airspace, etc. Relationships between various primitive terms were also identified.

[3] MNPS is Mininum Navigational Performance Specification. WATRS is West Atlantic Route System.

[4] This preprocesser was developed in the course of this work but is a separate tool that could be used for various specification notations. Information is available at http://www.cs.ubc.ca/spider/day/Research/hpp.html .

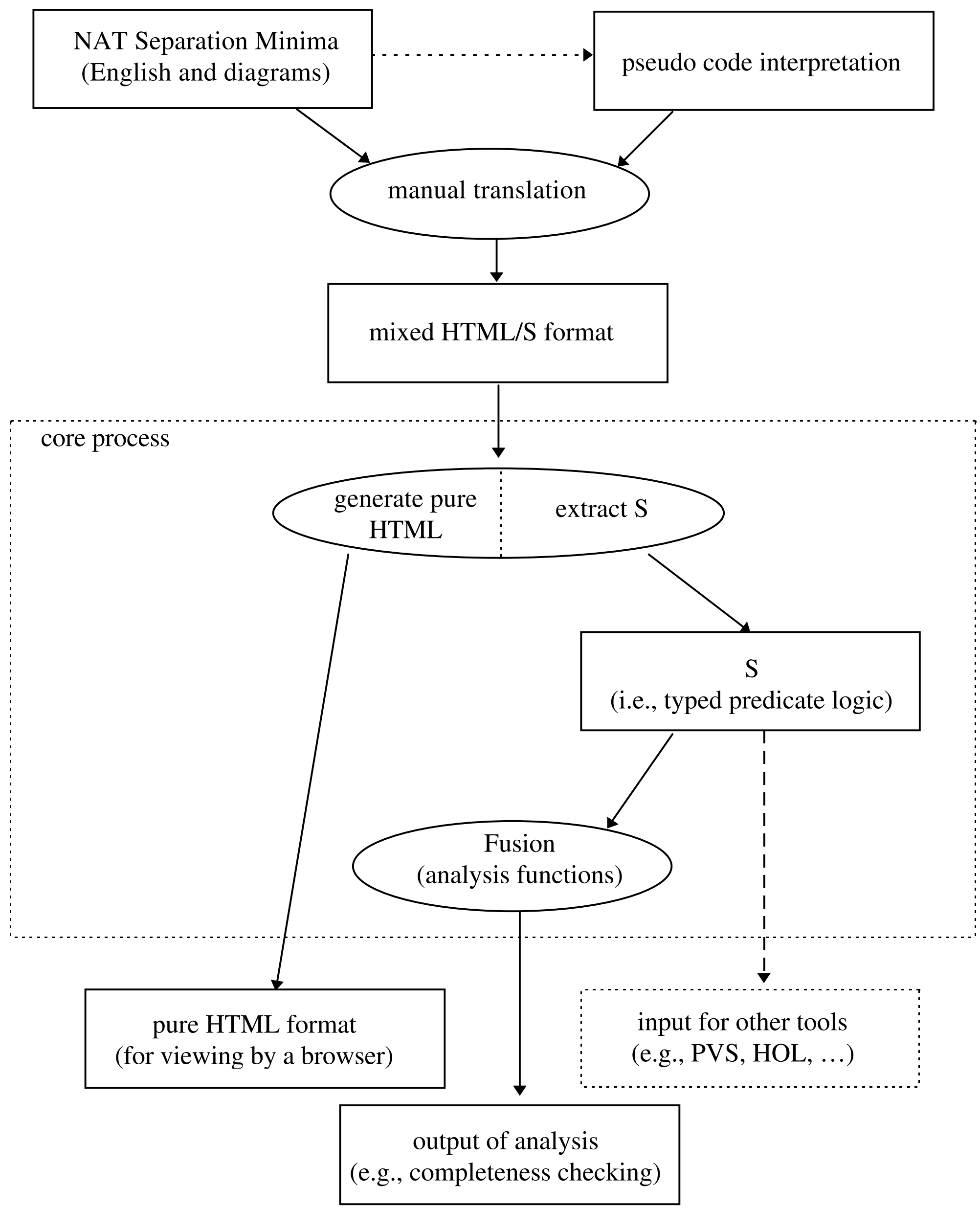

Figure 2: Process Overview

The most notable absence from the pseudo code was the top level requirement stating what separation means. Two aircraft are separated if they satisfy the separation minima for at least one dimension, i.e. vertical, lateral, or longitudinal. The criteria for each of these is given in different units. The top level requirement as stated in S is given in Figure 3. In this requirement, "ABS" takes the absolute value of its argument. Vertical separation is measured in feet. Lateral separation is measured in miles (or equivalently in degrees of latitude). Longitudinal separation is measured in minutes. Two aircraft on opposing tracks cannot be considered longitudinally separated during a certain range of time when the aircraft are close to crossing. Vertical or lateral separation must exist during that time.

```
AreSeparated(A:flight,B:flight,t:time) :=
   /* A and B are vertically separated based on flight level */
   (ABS(A.FlightLevel- B.FlightLevel) > VerticalSeparationRequired(A,B))
   OR
   /* A and B are laterally separated based on either position in degrees of latitude or position in miles */
   (if (LatitudeEquivalent(A,B)) then
          (ABS(A.LateralPositionInDegrees - B.LateralPositionInDegrees) >LateralSeparationRequiredInDegrees
(A,B))
    else
     (ABS(A.LateralPositionInMiles - B.LateralPositionInMiles) > LateralSeparationRequiredInMiles (A,B)))
   OR
   /* A and B are longitudinally separated based on time, depending on whether the two flights are in the
      approximate same or opposite direction */
   (if (AngularDifferenceGreaterThan90Degrees(A.RouteSegment,B.RouteSegment)) then
      /* opposite direction */
      NOT (WithinOppDirNoLongSepPeriod(A,B,t))
   else    /* same direction */
      ABS(A.TimeAtPosition - B.TimeAtPosition) >LongSameDirSepRequired(A,B));
```

Figure 3: Top level Specification of Separation

The addition of the top level requirement pointed out that the proper distinction had not made between minima for aircraft on opposing tracks and those on same direction tracks. The requirement on aircraft flying the same direction is a minimum number of minutes of separation. The requirement for aircraft flying in opposing directions is that some other form of separation must exist during the time period when the aircraft cross. This change mainly affected the statement of the top level requirement.

The resulting specification consisted of 15 tables, 16 definitions in S, and 47 uninterpreted constants. The largest table consisted of 8 rows and 6 columns.

5. Analysis

Our goal was to analyze the completeness, consistency, and symmetry of the tables in the separation minima specification. Completeness checking automatically determines the cases that are covered by the default column, or if no default is given, the cases that are not covered in the table. Consistency analysis returns pairs of columns with different return values that both include a set of conditions that can be true at the same time. Symmetry analysis determines if the table has the same meaning when its arguments (the pair of flights) are given in the opposite order.

All of these forms of analysis are based on the possible combinations of entries in the rows of the table. They cannot determine if some aspect of the decision given by the table has been omitted.

The framework for specification and analysis proposed in the thesis work of the first author is illustrated in Figure 4. Requirements specifications, possibly given in multiple notations are placed within a common logical framework using an embedding that closely matches the original notation and does not lose the structure of the specification. Semantic functions define the meaning of the embedded notation in logic. The semantic functions also indicate explicit join points for how multiple notations fit together, such as a predicate table being used to describe the condition on a transition in a statechart [5] similar to what is done in RSML (Requirements State Machine Language) [12].The keywords used in the embedding and their associated semantic functions are called a "style" of specification in S.

These semantic functions are executable in the sense that they map a structured specification, such as a table, into an expression in logic. The expression in logic is called the semantic representation in the diagram. One method of executing these semantics is to use rewrite rules within a theorem prover. However, this is a more general mechanism than is needed for functions known to be executable. Drawing on techniques from functional programming language implementations, Fusion includes a symbolic

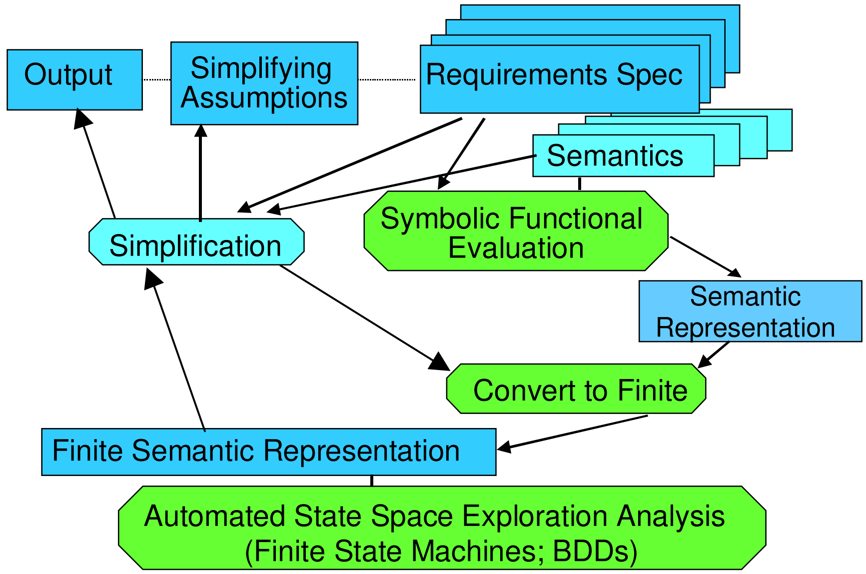

Figure 4: Specification and Analysis Framework

functional evaluator which can execute statements in S, stopping when it reaches uninterpreted constants. This engine carries out evaluation in place for efficiency, as is done in the implementation of a functional programming language.

Separately, clues given in the structure of the specification can be used to help determine simplifications to map the semantic representation into a finite model that can be efficiently represented using BDDs and analyzed using automatic means. Sometimes the simplification implied by the structure is not a valid abstraction. If the tool can not determine the validity of the simplifications these are stated to the user as "assumptions". This serves to the reduce the review process to one of evaluating isolated assumptions.

The results of the analysis map the simplifications back into the terms of the specification for the user to examine.

We regard tables to be a "style" of specification in the S language where the textual specification of the table in S is structurally close to the tabular presentation. Figure 5 shows the S representation of the table for "VerticalSeparationRequired" given in Figure 1. Our preprocessor turns this representation into an HTML table.

The keyword "Row" is a semantic definition that substitutes the label of each row into the list of predicates. The "_" describing the parameter of the predicate in the HTML version is given by a lambda variable. "DC" is a predicate that always returns the value true. This takes the place of the "." in cells in the HTML representation. "TRUE" is a predicate that states that its parameter must have the value "true". The result of applying the "Row" function is a list of elements with Boolean values.

The keyword "Table" gives meaning to the table, matching the conjunctions of values in the columns with return values. These semantic functions are defined in S and given in Appendix A.

```
VerticalSeparationRequired (A,B) := Table
        [Row (A.FlightLevel)    [(\x.x <= 280);   DC;      (\x. x > 450); (\x.x > 450)];
         Row (B.FlightLevel)    [  DC;       (\x.x <= 280); (\x. x > 450); (\x.x > 450)];
         Row (IsSupersonic (A)) [  DC;            DC;        TRUE;          DC      ];
         Row (IsSupersonic (B)) [  DC;            DC;        DC;           TRUE     ] ]
    [1000;1000;4000;4000;2000];
```

Figure 5: VerticalSeparationRequired Table in S

A.FlightLevel	_<=280	(280 < __) AND (_ <= 450)	450 < __

Figure 6: Example Row

Having related conditions in a row uses the structure of the table to show how the user views the possible values of the row label. For example, in the table found in Figure 1 for vertical separation, the flight level of each aircraft is only important in how it compares to flight levels 280 and 450 for the purposes of the function given by the table. Therefore the analysis can assume that the specifier would like the analysis results given in these terms as well. This is in keeping with our goal of returning results at the same level of abstraction as the specification.

The simplification engine uses the entries in a row to partition the state space for the aspect of the problem given by the label of that row. This partition can then be encoded in Boolean variables just as an enumerated type can be encoded for automatic verification (as seen in [1,10]). For example, if a row contained the entries found in Figure 6, there are three conditions given by the cells in this row:

A.FlightLevel <= 280
(280 < A.FlightLevel) AND (A.FlightLevel <=450)
450 < A. FlightLevel

The flight levels of A have been divided into three ranges. To represent these ranges, we need two Boolean variables, say a1 and a2. We can use the encoding given in Figure 7. The use of structure eliminates the need for the more heavyweight reasoning of a theorem prover in this instance.

This encoding results in one leftover possible encoding for the Boolean variables of a1=true and a2=true. Since this does not correspond to any possible real case, we add this to the final expression to check in the analysis to ensure that this artifact does produce an extraneous result.

This technique does make the assumption that the partition the user provides is complete and consistent. This is not always the case. Some readers may have noticed that for the table "VerticalSeparationRequired" in Figure 1, using only the elements in the first row leaves out the case for when the flight level is between 280 and 450. An earlier version of the tool stated these assumptions about the partition to the user. This approach isolated details of the specification to review separately. The conditions given by the partition could also be evaluated by an automated decision procedure in a theorem prover such as PVS [13].

After noting that these assumptions are incorrect (i.e., a range is missing), one possible remedy is to modify the table. However, in all cases for the tables in this specification, the predicates over numeric values consist of a comparison to a concrete value. To improve the accuracy of the analysis results, we added a simple interval checker which checks the partition of the range given by the elements of the row and adds any ranges not mentioned explicitly in the row. After this addition, one table remained with overlapping ranges in the elements of the row. The interval checker identified this case and we modified the table to separate the ranges into multiple rows and used environmental assumptions to relate the rows.

There are many times when the possible values given by the element labeling the row are known. An example is a Boolean condition such as "IsSupersonic" found in the "VerticalSeparationRequired" table. This element can take on the values true and false even though the row for it in this particular table never has a false case. The checker automatically recognizes these situations and considers both true and false as possible values in the analysis. This is applicable for any values of finite types. Enumerated types can be declared using S declarations.

A.FlightLevel <= 280	NOT(a1) AND NOT(a2)
(280 < A.FlightLevel) AND (A.FlightLevel <=450)	a1 AND NOT(a2)
450 < A. FlightLevel	NOT(a1) AND a2

Figure 7: Example Encoding

```
>%include minima.s
Including rules.s
>%comp VerticalSeparationRequired
VerticalSeparationRequired is:
(
 (Table
  [
   ((Row (FlightLevel A))
     [(\x.(x <= 280));DC;(\x.(x > 450));(\x.(x > 450))]);
   ((Row (FlightLevel B))
     [DC;(\x.(x <= 280));(\x.(x > 450));(\x.(x > 450))])
   ;((Row (IsSupersonic A)) [DC;DC;TRUE;DC]);
   ((Row (IsSupersonic B)) [DC;DC;DC;TRUE])])
 [1000;1000;4000;4000;2000])

Invoking interval checker...

Interval checker partitions the range into:
((FlightLevel A) > 450)
((280 < (FlightLevel A)) AND ((FlightLevel A) <= 450))
((FlightLevel A) <= 280)

Invoking interval checker...

Interval checker partitions the range into:
((FlightLevel B) > 450)
((280 < (FlightLevel B)) AND ((FlightLevel B) <= 450))
((FlightLevel B) <= 280)

The following cases
yield the default value of 2000
Case 1
Row 1 : ((280 < (FlightLevel A)) AND ((FlightLevel A) <= 450))
Row 2 : ((280 < (FlightLevel B)) AND ((FlightLevel B) <= 450))
Row 3 : DC
Row 4 : DC

Case 2
Row 1 : ((FlightLevel A) > 450)
Row 2 : ((280 < (FlightLevel B)) AND ((FlightLevel B) <= 450))
Row 3 : DC
Row 4 : DC

Case 3
Row 1 : ((280 < (FlightLevel A)) AND ((FlightLevel A) <= 450))
Row 2 : ((FlightLevel B) > 450)
Row 3 : DC
Row 4 : DC

Case 4
Row 1 : ((FlightLevel A) > 450)
Row 2 : ((FlightLevel B) > 450)
Row 3 : ((IsSupersonic A) = F)
Row 4 : ((IsSupersonic B) = F)


Stats for VerticalSeparationRequired completeness checking:
Number of cases identified: 4
Total time: 1 sec
 --------------------------------------------------------------
>
```

Figure 8: Completeness Checker Output

5.1 Completeness Checking

Checking the completeness of a table means determining if all possible cases are covered by the specification. The meaning of a column is the conjunction of the predicates in the row cells. For the tables, completeness checking involves checking whether the expression denoting the disjunction of the columns is a tautology. By applying the semantic function for the meaning of a predicate table and using the Boolean encodings identified by the simplification engine described above, we can evaluate this check efficiently.

Figure 8 shows a Fusion session in which the completeness analysis function “comp” is applied to the table “VerticalSeparationRequired”. Every line shown in Figure 2 is generated by Fusion except for the two user commands which appear on the lines that begin with the user prompt, “>”. The first command “%include minima.s” causes the S representation of the separation minima to be parsed and typechecked. The second command “%comp VerticalSeparationRequired” causes the completeness analysis function to be applied to the table “VerticalSeparationRequired”.

The completeness analysis function first invokes the interval checker for the first two rows. It correctly determines the missing range for the possible values of the “FlightLevel” of the aircraft.

The completeness analysis function then generates a list of the cases covered by the default column. The analysis reveals four of these cases. In order to minimize the amount of output generated by the checker, the results are given in an approximation of minimal sum-of-products form. This can be seen in the use of “don’t care” (“DC”) values for some of the rows. It was important that these results were given in terms of the unexpanded row label in cases where the row label was an application of a function defined elsewhere. This meant the reviewer could easily match the cases given in the results to the original table in HTML after substituting the row label into the blank.

In reviewing this session output, the domain expert would decide whether the default value, 2000, is appropriate for the listed cases. The maximum number of default cases revealed by the completeness analysis for this specification was 50 - for a table of eight rows and six columns. A table is not necessarily flawed because it has default cases — but it is important for the default cases to be enumerated and reviewed by a domain expert. This kind of analysis would be performed by some means in a disciplined system development process. However, the use of an automated completeness analysis function, as illustrated here, streamlines and systematizes the review process by enumerating the default cases explicitly. A possible method of evaluating these would be to iteratively examine a single case, determine whether it is an error or not, and then add it, likely in a generalized format (i.e., with "don't care" values in some of the cells) to the table. This approach would mean that the default cases would gradually be fully specified. Thus, the use of Fusion for this purpose can also be seen as a way to measure the quality of a specification.

In Heimdahl and Leveson's work [6], they are able to draw conclusions about the overall completeness of a specification by referring to a functional definition of the semantics. For this specification, we can ask whether the completeness of individual tables ensures the completeness of the overall specification. Given that the tables represent functions, if the other parts of the specification (including the uninterpreted functions) represent total functions then we can conclude that the specification is complete, assuming the scope of each table is complete.

5.2 Environmental Assumptions

In reviewing the output of the completeness checker, our domain expert pointed out that some of the cases produced were impossible. These were situations where the rows within a table were related to each other. For example, an aircraft cannot satisfy both of the constraints "InCruiseClimb" and "IsLevel" at the same moment.

These constraints are information about the physical limitations of the items involved in the specification. They can be considered assumptions about the environment. We documented these in S using expressions such as:

forall (F:flight).
  mutually_exclusive(InCruiseClimb F, IsLevel F)

where "mutually_exclusive" is defined to mean only one of its arguments can be true.

These expressions are evaluated using the symbolic functional evaluator as for the tables. However in order to reduce these expressions to the terms used in the tables, we substituted any existing items of the correct type as the parameter in the "forall" expression. Most of the tables involve two flights, A and B. The above environmental assumption would evaluate to:

mutually_exclusive(InCruiseClimb A, IsLevel A)
AND
mutually_exclusive(InCruiseClimb B, IsLevel B)

Substitutions determined by the simplification engine for the table, along with some additional Boolean variables (since all environmental assumptions are not relevant to every table) are used to encode the environmental assumption.

The addition of environmental constraints slightly changed the method of evaluating the completeness of a table. Instead of checking whether the meaning of the table was a tautology, we had to check whether the environment conjoined with the negation of the meaning of the table was a contradiction.

In the output, the environmental assumptions are not listed. They are existentially quantified out of the results.

5.3 Consistency Checking

Consistency checking involves comparing each column of a table to all other columns within the table that have a different return value to see if the cases denoted by the columns overlap. The same evaluation

```
>cons otherSameDirLongSep env
otherSameDirLongSep is:
(
 (Table
   [((Row (ReportedOverCommonPoint (A , B))) [TRUE;DC]);
    ((Row (SameOrDivergingTracks (A , B))) [TRUE;DC]);
    ((Row ((AllOf [A;B]) (\x.((IsOnRoute Routes3) x))))
      [DC;TRUE])]) [15;20;30])

Columns 1 and 2 conflict in the following:
Case 1
Row 1 : ((ReportedOverCommonPoint (A , B)) = T)
Row 2 : ((SameOrDivergingTracks (A , B)) = T)
Row 3 : (((AllOf [A;B]) (\x.((IsOnRoute Routes3) x))) = T)


Stats for otherSameDirLongSep consistency checking:
Number of cases identified: 1
Total time: 0 sec
 ------------------------------------------------------------
>
```

Figure 9: Output of Consistency Checker

and simplification process used for completeness checking is used for this analysis. However, here we check whether the conjunction of the meaning of the two columns is a contradiction. If the result is a contradiction, the checker indicates the two columns involved and lists the case(s) where they overlap in the same form as the output for completeness checking.

The results of analyzing the separation minima revealed that two tables are inconsistent. After consulting the official specification (i.e. not the pseudo code representation), our domain expert concluded that these are cases where the specification is ambiguous.

Figure 9 shows the result of analyzing one of the tables that is inconsistent. The table "otherSameDirLongSep" specifies the number of minutes of time that must exist between two aircraft (that are not both turbojet or both supersonic) flying in the same direction for them to be considered longitudinally separated. The checker identified that, for the case where two aircraft have reported over a common navigation point, are on the same or diverging tracks, and are both on a particular set of routes that have special criteria, the table is ambiguous as to whether there should be 15 or 20 minutes of separation between them.

The second table with inconsistencies describes requirements for lateral separation[5]. This table has eight rows and four columns. This case again involves special provisions for particular routes that overlap with the more general criteria. The results clearly reveal cases in the official specification that are ambiguous as to the amount of lateral separation required between aircraft .

5.4 Symmetry Checking

Symmetry is a desirable property of this specification. It is important that the separation criteria are the same regardless of the order of the parameters (i.e., the two flights) given to the functions and predicates that the tables describe.

[5] There were actually two other tables with inconsistencies but these two tables "LateralSeparationRequiredInMiles" and "LateralSeparationRequiredInDegrees" represent the same sets of conditions, but have different return values for the functions.

```
>%sym ssOppDirNoLongSepPeriod

ssOppDirNoLongSepPeriod is:
(
      (Table [((Row (ReportedOverCommonPoint (A , B))) [TRUE;FALSE])]
   )
  [((ept (A , B)) , ((ept (A , B)) + 10));
   (((ept (A , B)) - 15) , ((ept (A , B)) + 15))])

The table is symmetric if the following condition(s) hold
(some conditions may overlap):

(
  ((ReportedOverCommonPoint ((FST (A , B)) , (SND (A , B)))) = T)
   =
  ((ReportedOverCommonPoint ((FST (B , A)) , (SND (B , A)))) = T)
  )

(
  ((ReportedOverCommonPoint ((FST (A , B)) , (SND (A , B)))) = F)
   =
  ((ReportedOverCommonPoint ((FST (B , A)) , (SND (B , A)))) = F)
  )

Total time: 1 sec
 ------------------------------------------------------------
>
```

Figure 10: Output of Symmetry Checker

To carry out symmetry checking, two versions of the table are created — one with each ordering of the parameters. The meaning of the disjunction of all columns within each table that return the same result is compared to the other table.

If these expressions are not equivalent, the symmetry checker returns constraints, that if satisfied, would mean the table is symmetrical. Figure 10 shows an example of the output of the symmetry checker applied to a simple table.

The initial results of this analysis (as seen in Figure 10) pointed out that the symmetry of a table is often dependent on the symmetry of the primitive terms used in the table. Environmental assumptions of the form,

```
forall A B.
   ReportedOverCommonPoint (A,B) =
         ReportedOverCommonPoint(B,A)
```

were added to make this analysis more accurate.

While this analysis did not reveal any errors in the specification, it did point out information about the primitive terms which might not be known by an implementor of the separation minima in software.

6. Future Work

This work is the first attempt to validate the thesis ideas of the first author. The operational semantics integrate the "style" of specification with the predicate logic environment, and are used directly in analysis to map the specification (possibly in multiple notations) into a form that can be automatically analyzed using state space exploration analysis techniques. The use of the explicit semantic definitions retains the structure of the specification for analysis so that structure can be used to help create a finite model for analysis. This was illustrated here in the use of predicate logic along with a tabular notation, and using the structure given in the rows to partition the state space. The continuation of this thesis work will look at how the techniques used in this example can be generalized for other notations and other state space exploration analysis techniques, such as model checking.

A particular issue in this work is the use of constants with semantic definitions as the keywords that capture the structure of the table, such as "Row" and "Table". This means that the notation associated with the semantics no longer has to be "lifted" from the base S notation. If the more traditional path of defining keywords like "Row" and "Table" as constructors had been chosen, any place one table references another table, the reference would need to be "wrapped" with its semantic function to refer to the meaning of the table. However, the execution of the semantic functions expands the definitions used in the specification. For the most efficient execution, evaluation must be done in place, which eliminates the original expression. This makes it difficult to return the appropriate level of abstraction in the results. In the case of the tables, the output had to be given in terms of the labels of the rows to be useful to the reviewer. We have already implemented a method of maintaining the original expression that works for the results given here. We are working on formally defining this method generally so that it is possible to retain the appropriate original expression while still taking advantage of evaluation in place.

Another important point in this work is the use of the general purpose interface language, predicate logic. State space exploration analysis techniques are geared toward model-oriented specification methods. While the form of the specification used in this example is functional, we have illustrated how data aspects of the problem can aid in the analysis. Having a general-purpose interface made it possible to integrate the functional and data aspects of the specification for analysis. Rather than taking the approach of only allowing specifications that definitely can be analyzed using finite means, we have started from general-purpose logic interface and applied particular techniques when the specification fit a particular form. This opens a door to the possibility of using formal specifications created primarily by domain experts as input to more specialized analysis techniques such as theorem proving performed by formal methods experts. It would be straightforward to convert the S representation of our tabular style of specification into input for other analysis tools such as PVS [13] and HOL [4].

Recently, a 4th edition of the document "Application of Separation Minima for the NAT Region" has been produced. This document has some significant additions including reduced vertical separation minima. It would be interesting to see if this document corrects the ambiguities found in this work through consistency checking.

7. Conclusion

This paper is perhaps most notable for the example of applying formal methods to the ICAO standard for separation between aircraft over the North Atlantic. The ease with which a domain expert was able to review and edit the specification and analysis results is a favourable data point in the struggle to make formal methods acceptable to industry. Beyond documenting this collaborative effort as an instance of the industrial use of formal methods, this work illustrates:

- the integration of the tabular style into a general-purpose predicate logic environment which allowed the specification of uninterpreted functions, and environmental assumptions;
- a framework for analysis which uses the explicit operational semantics directly, allowing different notations to be combined, and making it possible to exploit the structure of a given notation in analysis and to return results at the correct level of abstraction;
- the advantages of using a presentation format in HTML and how it is possible to integrate this with a format that can be used as input to analysis tools.

Although it rests upon a solid mathematical foundation, we believe that the approach illustrated in this paper could be fully integrated into an industrial system/software engineering process without causing the domain experts to be excluded or extensively retrained in formal methods. The automated analysis provided by Fusion streamlines the manual review process by automating some of the processing that would otherwise need to be done manually.

Acknowledgments

The authors would like to thank Richard Yates of MacDonald Dettwiler for comments on an early draft of the specification. The first author would also like to acknowledge the input of her thesis committee in focusing these ideas. Reviewers for this workshop made helpful comments. The university based component of this collaborative research is supported by funding from Hughes International Airspace Management Systems, MacDonald Dettwiler, and the BC Advanced Systems Institute.

References

[1] Joanne Marie Atlee. *Automated analysis of software requirements.* PhD thesis, University of Maryland, 1992.

[2] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-25(8):677-691, August 1986.

[3] Tom DeMarco. *Structured analysis and system specification.* Yourdon Press, Englewood Cliffs, New Jersey, 1979.

[4] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* Cambridge University Press, Cambridge, 1993.

[5]David Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231-274, 1987.

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

[7] C.L. Heitmeyer and B.G. Labaw. Consistency checks for SCR-style requirements specifications. Technical Report NRL/FR/5540-93-9586, United States Naval Research Laboratory, Washington, D.C., December, 1993.

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

[9] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, 6(1):2-13, 1980.

[10] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Computer-Aided Verification: Fourth International Workshop*, 1992.

[11] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 285-299, Valletta, Malta, September, 1994.

[12] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process control-control systems. *IEEE Transactions on Software Engineering*, 20(9):684-707, September, 1994.

[13] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, pages 748-752, Saratoga, NY, 1992.

[14] Sam Owre, John Rushby, and Nataranjan Shankar. Analyzing Tabular and State-Transition Requirements Specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, April, 1996.

[15] Sam Owre, John Rushby, and Natarajan Shankar. Integration in PVS: Tables, Types, and Model Checking. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 336-383, Enschede, The Netherlands, Springer-Verlag Lecture Notes in Computer Science, Vol. 1217, April, 1997.

[16] David Lorge Parnas. Tabular representations of relations. Technical Report 260, Communications Research Laboratory, Faculty of Engineering, McMaster University, October 1992.

[17] J.M. Spivey. *Understanding Z.* Cambridge University Press, Cambridge, 1988.

Appendix A

The S notation is very similar to the syntax for the term language used in the HOL theorem prover [4]. But unlike HOL, S does not involve a meta-language as part of the specification format for declarations and definitions. Instead, the syntax for declarations and definitions is an extension of the syntax used for logical expressions. (In this respect, S more closely resembles Z and other similar formal specification notations.) For example, the symbol ":=" is used in S for a definition, e.g., "TWO := 2", in contrast to an assertion, e.g., "TWO = 2".

Another difference that will likely be noticed by readers familiar with HOL is the explicit type parameterization of constant declarations and definitions. Type parameters, if any, are given in a parenthesized list which prefixes the rest of the declaration or definition. This is illustrated in the definitions given below by the parameterization of "EveryAux" by a single type parameter, "ty".

Many of the definitions shown below are given recursively based on the recursive definition (not shown here) of the polymorphic type "list". These recursive definitions are given in a pattern matching style (similar to how recursive functions may be defined in Standard ML) with one clause for the "NIL" constructor (i.e., the non-recursive case) and another clause for the "CONS" constructor (i.e., the recursive case). Each clause in this style of S definition is separated by a "|". The functions "HD" and "TL" are standard library functions for taking the head (i.e., the first element) of a list and the tail (i.e., the rest) of a list respectively.

Type expressions of the form, ":ty1->ty2", are used in the declaration of parameters that are functions from elements of type "ty1" to elements of type "ty2". Similarly, type expressions of the form, ":(ty) list", indicate when a parameter is a list of elements of type "ty".

Lambda expressions are expressed in S notation as, "\x.E" (where E is an expression) .

The semantic definitions for the tabular notation given in the S notation are shown below.

```
(:ty)
EveryAux (NIL) (p:ty->bool) := T |
EveryAux (CONS e tl) p :=
        (p e) AND EveryAux tl p;

(:ty)
Every (p:ty->bool) l := EveryAux l p;

(:ty)
ExistsAux (NIL) (p:ty->bool) := F |
ExistsAux (CONS e tl) p := (p e) OR ExistsAux tl p;

(:ty)
Exists (p:ty->bool) l := ExistsAux l p;

(:ty) UNKNOWN : ty;

(:ty)DC := \(x:ty).T;
TRUE := \x.x=T;
FALSE := \x.x=F;

(:ty1)
RowAux2 (CONS (p:ty1->bool) tl) label :=
        CONS (p label) (RowAux2 tl label) |
RowAux2 (NIL) label := NIL;

(:ty)Row label (plist:(A->bool)list) :=
        RowAux2 plist label;

Columns t :=
  if ((HD t)=NIL) then NIL
  else CONS
    (Every (HD) t)
    (Columns (Map t (TL)));

(:ty)
TableSemAux2 (NIL) (retVals:(ty)list) :=
  if (retVals=NIL) then UNKNOWN
  else (HD retVals) |
TableSemAux2 (CONS col colList) retVals :=
  if  col
  then (HD retVals)
  else TableSemAux2 colList (TL retVals);

(:ty)
Table t (retVals:(ty)list) :=
  TableSemAux2 (Columns t) retVals;

PredicateTable t :=
  Exists (\x.x) (Columns t);
```