# Practical Experience Applying Formal Methods to Air Traffic Management Software

Richard Yates, Jamie Andrews, Phil Gray

**Abstract.** This paper relates experiences with formal methods that are relevant to the systems engineering activities of requirements specification, design documentation, and test case generation. Specifically, this paper reviews the lessons learned from the application of formal methods to selected components of an air traffic management system. This project used experimental tools developed at the University of British Columbia: S, a formal specification tool; HPP, an HTML documentation tool; and TCG, a test case generation tool. The components experimented on are from a recently fielded system written in C++ using unimplemented pre- and post-conditions on components. The purpose of the experiment was to evaluate the usefulness of these formal methods to uncover design or logic errors in the system components and to assist in designing test cases. This experience identified some ambiguities in the original specification, evaluated the feasibility of the experimental tools we used, and identified areas in which the tools could be improved.

## INTRODUCTION

There is a widespread recognition that software engineering falls short of its goal of building robust, correct systems. There is an interest in enhancing its methods with applicable techniques from formal methods. There is a need to experiment with these new methods, and to communicate the results of the experiences, so that other organizations can more effectively approach the use of formal methods in their own application domains. This paper discusses a specific industry experience using some tools of formal methods applied to software for an Air Traffic Management system to formalize requirements, enhance design documentation, and facilitate test case generation.

Our work considered C++ components whose interfaces and function definitions were specified by comments expressing pre- and post-conditions. Our experiments formalized these assertions using a formal description notation, documented our effort using an HTML documentation tool compatible with the notation, and attempted to generate software test cases from the formal specification. As a result of our effort, we identified some ambiguities in the original informal specification, evaluated the feasibility of the experimental tools we used, and identified areas in which the tools could be improved.

Late in the project, some work was done with the Z specification notation to obtain some perspective on the experience with S.

This paper reports experiences with formal methods used in software system development, but the observations we make are relevant to any system engineering problem. All engineering disciplines require techniques to capture and analyze requirements, elaborate and analyze a design, specify interfaces, and evaluate the product. Evidence of the wide applicability of these techniques can be found in the survey article by (Clark et al. 1996) which reviews examples of formal methods for specification and verification in many different engineering domains. An informative report of a positive experience with evaluating formal methods in a nuclear reactor design is reported in (Knight et al. 1997).

The formal methods tools discussed in this paper are unreleased preliminary versions that are still under development by the FormalWARE project.

## FORMALWARE PROJECT

This work was done within the FormalWARE project. This project is a joint industry-university research initiative intended to foster the development of formal methods that are directed toward industrial needs and to encourage the use of these methods by industry in British Columbia. The research partners are Raytheon Systems Canada (formerly Hughes Aircraft of Canada Limited), MacDonald Dettwiler and Associates, The University of British Columbia, and The University of Victoria with funding from the industrial participants and the B.C. Advanced Systems Institute.

The MacDonald Dettwiler (MDA) involvement has been in the areas of component requirement specification and test case generation. Jamie Andrews, a research associate of the FormalWARE consortium at the time of this research, worked closely with MDA to help familiarize this company with:

- the specification notation: S
- the S typechecker tool: FUSS

- the test case generation tool: TCG
- the HTML documentation tool: HPP

A full description of all of the projects of the FormalWARE research consortium is at the web site: http://www.cs.ubc.ca/formalWARE/.

## AIR TRAFFIC MANAGEMENT CODE

Our experiment used samples of recently developed code for an ATM (Air Traffic Management) system and explored the usefulness of formal methods. This system used informal comments that captured pre- and post-conditions asserted to hold for each function (Meyer 1988). These assertions were manually reviewed during the original development phase. This project, which started after coding and testing was completed, examined how these assertions could be translated into S.

The examples looked at in this section come from a component that defines an airspace object. This is part of an Environmental Processing subsystem, which handles meteorological data, and other data describing the environment within which ATM operates. This subsystem is part of a larger Aeronautical Information System that supplies information used in air traffic management.

Figure 1 provides an example of how assertions were captured in a header file for the C++ implementation of a simple function:

```
int assocMap (int aAssocMapNr) const;
/*
  Requires:
    ! this.isNull();
    0 <= aAssocMapNr < nrAssocMaps();
  Ensures:
    result == the number of the
              associated map
*/
```

**Figure 1. Simple C++ Function Header with Assertions as a Comment**

The above function takes as input aAssocMapNr (the index of an associated map), decodes it into a map identifier, and returns it to the caller. The purpose of the function is to provide a code lookup mechanism to provide compatibility between two subsystems that use different integer values as indexes to their set of associated airspace maps. The first line of the "requires" pre-condition states that this airspace object is not null (i.e., the object must have been initialized). The second line requires that the number of the associated map be within the range of maps currently managed by this airspace object. The "ensures" post-condition is the requirement that the returned number is the proper integer code for the associated map.

Figure 2 provides a more complex example:

```
void setAirspaceComputedBoundary(
 UTL_Status_Object& status, OUT_PARAM
 int aNrAirspaceBoundaries, IN_PARAM
 ENP_ICW_Location airspaceBoundary[]
   IN_PARAM);
/*
  Requires:
    ! this.isNull();
    airspaceBoundary is an array of
aNrAirspaceBoundaries elements
  Ensures:
    ! this.isNull();
    if (input parameter is consistent
      with the invariants &&
      aNrAirspaceBoundaries <=
           AED_MAX_AIRSPACE_
           COMPUTED_BOUNDARIES)
    then
      the object is unchanged,
      except that
        for k>=0 and
        k<aNrAirspaceBoundaries
          airspaceComputedBoundary(k)
          == airspaceBoundary[k] &&
        nrAirspaceBoundaries ==
          aNrAirspaceBoundaries;
      status == UTL_SUCCESS;
    else
      this == old this;
      status ==
        UTL_BAD_VALUE_FOR_OPERATION;
*/
```

**Figure 2. More Complex C++ Function Header with Assertions as a Comment**

The above function takes an airspace boundary data structure, airspaceBoundary[], with aNrAirspaceBoundaries number of elements in it and updates the airspace object. The pre-condition requires that the airspace object has been initialized and that the last two parameters of the function interface are consistent. The post-condition ensures that if a set of invariants is met and the function is being called with a legitimate number for aNrAirspaceBoundaries, then the object will be updated with changes to data visible only through the airspaceComputedBoundary and nrAirspaceBoundaries services, otherwise the object is unchanged. In either case, an appropriate status is set.

## SPECIFICATIONS USING THE S NOTATION

The specifications were written in the S notation. This is a formalism based on typed predicate logic, an extension to the propositional logic which is well-known to most engineers and software developers. It uses the familiar logical operators (OR, AND, NOT, IF…THEN). It is typed in the sense that each identifier must belong to a specific type just as is the case in a strongly typed programming language. S was conceived as an alternative to the Z specification notation and is intended to overcome some of the practical limitations of using Z. A description of S can be found in (Joyce et al. 94).

FUSS is a typechecker that parses S specifications to check for conformance to the syntax and typing rules of the notation in a manner analogous to the checking performed by a compiler for a strongly typed programming language to ensure that the specification was syntactically correct and well typed. The use of FUSS to check S specifications is analogous to the use of FUZZ to check Z specifications.

The informal specification of Figure 1 was translated into an S specification as:

```
forall x.
forall aAssocMapNr.
If NOT ((x.isNull) (at_time 1))
   AND (0 <= aAssocMapNr)
   AND (aAssocMapNr <
((x.nrAssocMaps) (at_time 1)))
then
    (is_map_ID_for aAssocMapNr
((x.assocMap) aAssocMapNr) (at_time
2)));
```

**Figure 3. S Formalization of the Simple C++ Function**

There are several things to note about this syntax:

- The use of "forall" provides a simple, readable text to identify the universal quantifier. This use of a format that does not require special fonts is one of the most useful features of S. Z specification notation has a rather forbidding appearance to the non-mathematically inclined. Unlike Z, S permits the construction of very English-like specifications. Furthermore, these specifications can easily be included within ASCII-based documents such as comments within code files or as text within the project design documentation.
- S is a specification notation based on a declarative paradigm. Our example is written in C++ code based on a procedural paradigm where program steps make sequential changes over time. Consequently, we need a way to represent the passage of time in S. Figure 4 shows how we do this by defining a new type "time" and a new construct "at_time" to allow us to represent successive times of interest during the execution of the program:

```
:time;
at_time: num -> time;
```

**Figure 4. Specification of the Type "time" and the Function**

- The element "x" stands for our airspace object. The expression "x.isNull" stands for the "isNull" method applied to the object to return a boolean value. Figure 5 shows how this method was specified in S:

```
isNull: AED_Arspc_Object -> time -> bool;
```

**Figure 5. Specification of the "isNull" Method**

  The above declaration of "isNull" says that this function first takes a value of the type "airspace object" followed by a value of type "time" and yields a value of type "boolean". The FUSS typechecker uses this definition to check the expression: "(x.isNull) (at_time 1)".
- Because the function "assocMap" is a function within an AED_Arspc_Object being called at a particular time, we give it two additional parameters: the object it belongs to and the time it is being called. This could be written as "(assocMap x aAssocMapNr (at_time 2))" where x is the object. Or, equivalently, it could be written as "((x.assocMap) aAssocMapNr (at_time 2))" which helps point out the fact that "x" is the object being handled. Similarly, a call to the isNull function could be written as "(isNull x (at_time 1))" or "((x.isNull) (at_time 1))".
- The post-condition ("result == the number of the associated map") is an example of a problem in formalizing an informal specification and has implications for generating test cases from a specification. This assertion has no representation within the coded C++ functions of the original airspace component. A new function "is_map_ID_for" was created to express the required property. This creates problems later if test cases are to be generated from this specification since this is not an executable function of the C++ code. Any test using this new function would be "white box" testing (i.e. testing that requires the tester to have knowledge of a property of the system under test that is not available from the C++ functions of the original airspace component).
- The S notation has no special formatting rules. The indentations seen above are those chosen by the authors for readability, and is not a style imposed by S.

The informal specification of Figure 2 was translated into an S specification as:

```
forall x j.
forall <parameters-list>
  <invariants-list>.
If NOT ((x.isNull) (at_time 1))
  AND ((array_size airspaceBoundary)
     == aNrAirspaceBoundaries)
then
  if
    invariants_met(<invariants-list>)
```

```
  AND
   (x.setAirspaceComputedBoundary)
   (status, aNrAirspaceBoundaries,
  airspaceBoundary) (at_time 1)
  AND(aNrAirspaceBoundaries <=
     AED_MAX_AIRSPACE_
     COMPUTED_BOUNDARIES)
  then
     <complicated-statement-
      asserting-the-object-
      has-not-changed>
    AND ((j<0) OR
      (j>=aNrAirspaceBoundaries)
      OR
        ((array_element
       ((x.airspaceComputedBoundary)
        (at_time 2)) j) ==
           (array_element
              airspaceBoundary j))
    AND ((x.nrAirspaceBoundaries)
          (at_time 2) ==
             aNrAirspaceBoundaries)
    AND ((value_of status
        (at_time 2)) == UTL_SUCCESS
 else
   <complicated-statement-
    asserting-the-object-
    has-not-changed>
   AND ((value_of status
      (at_time 2)) ==
        UTL_BAD_VALUE_FOR_OPERATION)
```

**Figure 6. An Abbreviated S Formalization of the more Complex Function Specification**

There are several things to note about Figure 6:

- The use of angle brackets represents places where large chunks of specification were removed to make the above more readable.
- In building specifications, we found that in dealing with functions with large numbers of parameters we had to repeat lists of variables. There was no "macro" facility that let us substitute a simple identifier for these long lists. This made our specifications quite long. In fact, the actual specification was 101 lines long, not the 19 lines shown above.
- The predicate "invariants_met" is a 96 line long definition. What this demonstrates is that the tool requires the building up of a specification infrastructure. This infrastructure can be quite large. For the airspace component, some 160 lines of informal specifications led to 1300 lines of formal specifications. The larger the component, the more this specification infrastructure size can be amortized. But a system with more components will not generally gain as much from this kind of amortization effect since most specifications are local.
- A comparable translation of the informal specification into Z has a significantly smaller size, roughly 60% of the size of the S specification. The size difference comes from two facts: Z supports a schema facility which helps modularize and reuse elements of the specification, and the Z specification was done after we had gained some skill in formal specifications. This second factor is hard to quantify, but our assessment is that it is fair to say that S is not as compact as Z. On the other hand, Z uses a standard formatting that requires special tools and creates layouts that are longer and require far more blank space on the page than S. Also, it uses a notation that is highly mathematical and opaque to most software engineers.
- The Z notation uses an apostrophe decoration on a variable to indicate the implicit passage of time. In Figure 6 the last three lines could be represented as status' = UTL_BAD_VALUE_FOR_OPERATION where the apostrophe captures the (at_time 2) indication. Clearly, Z is more condensed. It could be argued that Z is more opaque since it requires an understanding of the notational convention.

## HTML DOCUMENTATION TOOL

The HPP tool (developed by Nancy Day at the University of B.C.) was used to generate on-line documentation for the specification. It is able to automatically generate an HTML document from a specification. The usefulness of the hyperlinked approach to specifications should be obvious: users are able to jump to the definition of a construct. This is extremely useful.

This tool lets users include a definition using the following construct:

```
<HPPTAG s="name-of-construct",
[
put the specification here
]>
```

**Figure 7. HPP Tag**

The tool will automatically link all other references to "name-of-construct" back to this definition.

We found this to be an excellent idea for documentation. Unfortunately, our other project documentation is not currently in HTML format, so the overall product documentation is not compatible with this the use of this tool at this time. But this is definitely the right idea for the future.

## TEST CASE GENERATION

The TCG (Test Case Generation) is an experimental tool, developed by Michael Donat of the University of B.C. and described in (Donat 97). It is designed to work from S specifications to produce test cases.

The automated generation of test cases offers the potentially significant advantages of reducing test generation effort, permitting tailoring of test coverage schemes, and ensuring consistency of test cases with the system specifications. The strength of TCG's approach is that test case generation is based on logical calculation rather than heuristics.

We encountered problems using TCG on subsets of our specification because we used a prototype version of the tool. Some of our problems are traceable to the fact that we created specifications with large numbers of universally quantified variables using a specification style that differed from Donat's approach. We intend to re-evaluate this tool as it evolves.

## LESSONS LEARNED

There were a number of useful lessons learned from this research project.

Our experiment in formalizing specifications for functions did help uncover some cases where the natural language of the informal specification was unclear. In this regard, formal methods is helpful because it adds rigor to the process. But it did not uncover any major errors in the manually generated specifications. Our experience was inadequate to properly judge the cost-effectiveness of this approach. But in our estimation, the costs of formalization are rather high and the benefits somewhat limited in the context of the current experiment. While this technology is promising, serious industrial use will depend on the development of commercial grade tool sets to aid the working engineer.

Working with a formal specification notation is a highly-skilled task. Despite the efforts to create S as a language which is English-like in its expressions, there was a rather large learning curve to overcome. A significant portion of the time spent was involved in understanding the constructs of the notation. This was the case despite the fact that the engineer working on the task has many years experience programming and a background in formal logic. The syntax and semantics of formal specification notations are significantly different from that of procedural programming languages and, in the case of S, are closer to the those of functional programming languages such as ML.

The format of the informal specifications was not conducive to a simple formal specification. Functions had many parameters and the informal specifications often referred to the internal state of the component rather than to any externally observable condition. We demonstrated that S could formalize these specifications, but it required some rather extensive definitions and we were left with more complicated specifications than we would have liked. In retrospect it would have helped to have used a more extensive and complex substructure of definitions. For example, Z has a schema construct that permits the modularization of a specification through schema inclusion. An equivalent technique has not yet been developed for S.

Formalization does not remove all sources of errors. Since the formalized specification is checked by FUSS for only conformance to syntax and type rules, there is still room for error in the specification. FUSS, like many other formal methods tools, has no underlying model of domain knowledge against which the correctness of the specification can be checked, so it is possible to make erroneous specifications. (An analogy is that a clean compile of a software component only guarantees that the program is syntactically correct, not that it is defect free.) For example, in our use of the "at_time" function to express sequential dependencies in our specification, there was no mechanism to ensure that the actual time values were correct. There was at least one place where a large section of specifications was meant to be tagged as applying at time "3". But because of a cut-and-paste error in building this part of the specification, some values were left at "2". FUSS was unable to catch this kind of error since it only understands the notational formalities, not the intentions about the sequential timing model in the mind of the person making the specification. In our use of Z with a theorem prover we were able to show, for very simple specifications that used an underlying mathematical representation, that the domain knowledge of the mathematical toolkit helped move us a slight distance forward in handling this problem of "domain specific knowledge" with formal specifications.

Issues of system and component specification had implications for automatic test case generation. If the assertions for a function can be expressed in terms of properties available within the specification, then "black box" test cases can be generated. However, if an assertion cannot be expressed with properties of the system, then a new function has to be invented. The value of this function can be checked during testing only by a "white box" testing technique. In Figure 3 the "is_map_ID_for" is a function that can be checked by a test if there is a way to look inside the airspace object and ensure that the a number returned by the "assocMap" function is in fact the valid map ID for the number aAssocMapNr.

The S specification notation can be improved in the following areas:

- While S provides some support for writing readable specifications, there is still much that needs to be done. Our experience was that the volume of specification required to express fairly simple requirements was surprisingly large. Some of this concern might be addressed with more experience with S which leads us to insights on how to write more concise specifications. Some must come from our recognition that formal specifications necessarily must make explicit details that are implicit in less formal specifications. And, finally, some of the concern is expected to disappear when we have enough experience to show that there are large chunks of the formal specification which could be reused in other projects because they are industry-specific or domain-specific elements reusable in similar systems.

- Some of the notational techniques used by S are difficult for the non-specialist to master. The S notation is based on lambda calculus which has non-obvious differences for programmers who are used to a procedural programming style of notation such as C++. The goal of supporting English-like specifications is laudatory, but the underlying syntax of S is a stumbling block for an untrained engineer. Our subsequent experience with Z was that despite its forbidding notational appearance, it placed no more real difficulty than S and in some ways was in fact a more effective specification notation.
- Overloading of types is not permitted. Consequently the common mathematical operators cannot be used when different numerical types are specified. Removal of this restriction would help in making the “dot notation” style of S more useful since it would permit the same name to be used for methods for multiple types of objects.

An industrial equivalent of FUSS requires some enhancements:

- A more helpful user interface then the command line paradigm. A significant element of the success of the menu-oriented windows interface is that it relieves the user of remembering many commands. The ability to use interface forms relieves the user of remembering the complexities of command syntax. In particular, a context-sensitive editor that identifies errors upon input would reduce the number of edit-run-debug cycles required by the current loosely integrated command line interface toolset paradigm.
- Better localization of specification errors needs. FUSS displays errors using the parsed version of the specification. This creates problems for the user in identifying what portion of his specification is the source of the error.

The HPP hyperlink specification generator was useful:

- The experience was generally very positive with this tool. The only problem was a minor restriction over the name used in setting up the HPPTAG to create a named link to other sections of the document.

TCG is a research prototype that needs to be enhanced to be ready for use in industry:

- The tool needs its algorithms optimized to handle larger, more complex specifications.
- The format of the output from this tool is rather voluminous and unwieldy. More control over generating test cases and producing the documentation about the test cases is essential.

## CONCLUSIONS

Formal methods have been proposed for use in many systems engineering tasks. We need to identify the costs and benefits of using such methods in different settings, to evaluate where they are more usable and less usable, and how they can be improved. This paper has described some experiences relevant to this evaluation for the specific field of software engineering.

This project was a valuable learning experience for MacDonald Dettwiler. To achieve industrial acceptance, formal methods must bridge the gap to the working engineer. To achieve this, we need tools that are robust, supplied with an appropriate graphical user interface, designed to make fewer requirements for logical sophistication on the engineer, optimized for real world cases, and provided with good documentation and tutorial support. Formal methods are not a panacea that will guarantee robust and correct systems, but they can be a valuable aid in developing better systems. Our experience was, on the whole, a positive one.

## ACKNOWLEDGEMENTS

The authors wish to thank Jeff Joyce, Nancy Day, and Michael Donat for their helpful review of early drafts of this paper.

## REFERENCES

Andrews, Jamie, “Executing Formal Specifications by Translation to Higher Order Logic Programming”, *1997 International Conference on Theorem Proving in Higher Order Logics*, 19-22 August, 1997.

Andrews, J. H., Day, N. A., and Joyce, J. J., “Using a Formal Description Technique to Model Aspects of a Global Air Traffic Telecommunications Network”, *FORTE/PSTV ’97, the 1997 IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, November 18-21, 1997.

Clark, Edmund M, Wing, Jeannette M., et al., “Formal Methods: State of the Art and Future Directions”, ACM Computing Surveys, Vol. 28, No. 4, December 1996.

Donat, Michael R, “Automating Formal Specification-Based Testing”, Lecture Notes in Computer Science, TAPSOFT ’97: Theory and Practice of

Software Development, 7th International Joint Conference CAAP/FASE, volume 1214, edited by Michel Bidoit and Max Dauchet, Springer-Verlag, April 1997.

Joyce, Jeffrey J., Day, Nancy, and Donat, Michael R., "S: A Machine Readable Specification Notation based on Higher Order Logic", *Lecture Notes in Computer Science 859, Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, edited by T.F. Melham and J Camilleri, Springer-Verlag, 1994.

Knight, John C, DeJong, Colleen L., Gibble, Matthew S., Nakano, Luis G., "Why Are Formal Methods Not Used More Widely", Proceedings of the Fourth NASA Langley Formal Methods Workshop, September 1997.

Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

## BIOGRAPHY

**Richard Yates** has been with MacDonald Dettwiler since 1979 and has held a number of programming and technical positions. He has specialized in performance engineering. He brought to this project a background in formal logic.

**Jamie Andrews** received his Ph.D. from the University of Edinburgh in 1991. His research interests are in Prolog and formal methods. He was a postdoctoral fellow and sessional lecturer at Simon Fraser University (1991-1995) before joining the FormalWARE group as a Research Associate (1996-1997). He has recently (Fall 1997) taken a faculty position at the University of Western Ontario. He brought to this project a background in logic programming, formal methods, and specification techniques.

**Phil Gray** received his Ph.D. in Computer Science from Imperial College, London in 1971. From 1971-1981 he was a member of the scientific staff at Bell-Northern Research. From 1981 to present he has held a number of technical lead positions at MacDonald Dettwiler. He specializes in interactive systems, user interface design, and the application of human factors to user-software interfaces. He brought to this project a background in system specification and experience as the technical lead on a number of the air traffic management systems.