S : A Machine Readable Specification Notation based on Higher Order Logic

Department of Computer Science
University of British Columbia
Vancouver, British Columbia, Canada

J. Joyce, N. Day and M. Donat

This paper is from the Proceedings of the 1994 International Meeting on Higher Order Logic Theorem Proving and its Applications, Lecture Notes in Computer Science, vol. 859, pp.285-299, Springer-Verlag.

Abstract:

This paper introduces a new notation called S which is based on higher order logic. It has been developed specifically to support the practical application of formal methods in industrial scale projects. The development of S has occurred in the context of an investigation into the possibility of using formal specification techniques in the development of a $400 million air traffic control system. We were motivated to develop this notation after reaching the conclusion that existing notations such as Z are not suitable for use in this particular project. In addition to providing an introduction to S, this paper describes a public domain software tool called ``Fuss" which has been implemented to support the use of S as a specification language.

Introduction

S is a machine readable notation developed specifically to serve as a specification language for the practical application of formal methods in industrial scale projects. The notation is based upon the formalism of higher order logic and, in this respect, shares much in common with the machine readable ``object language" of the HOL system [5]. The development of S has also been influenced by the Z notation [10]. Like Z, and unlike the HOL object language, S includes constructs for the declarations and definitions of types and constants. S also includes a specification construct called a ``template" which serves as a packaging mechanism in the same way that ``schemas" are often used in Z specifications. Although the development of S has been influenced by our desire to use S to specify software, hardware and mixed software-hardware systems, S is a ``pure" notation in the sense that it does not include any built-in concepts of hardware or software.

To support the use of S as a specification notation, we have developed a prototype implementation of a tool called ``Fuss" that checks an S specification in a manner analogous to the checking of a Z specification by a commercial tool called ``Fuzz" [11]. In particular, our tool checks for conformance to rules of syntax and typing. Additionally, we have prototyped an extension to Fuss that translates an S specification into an ML script (for HOL88). When used as input to the HOL system, the ML script generates a HOL theory consisting of declarations and definitions from the S specification. Thus, the S notation and Fuss could be used as a front-end for a formal verification methodology which involves the use of the HOL system. Similarly, we are interested in the possibility of developing extensions to Fuss that translate S specifications into the notations of other higher-order logic theorem-proving tools and perhaps even other kinds of verification tools such as model checkers.

Section 2 elaborates on the context, motivation and objectives of the development of S. Section 3 provides a brief description of S. In Section 4, we illustrate the use of S in terms of the translation of a (hypothetical) software requirement for an air traffic control system into S. Section 5 briefly describes Fuss, our prototype implementation of a tool for checking specifications written in S. This is followed by a description of a tool, implemented as an extension to Fuss, that translates S specifications into an ML script. Related work is discussed in Section 7. Section 8 briefly mentions some of our plans for the future development of Fuss and related tools. Finally, some advantages of the use of S and Fuss are considered in the conclusion.

Context, Motivation and Objectives

The development of S has occurred in the context of a project in which we are investigating the possible use of formal specification techniques in a $400 million air traffic control system. This system, called CAATS (Canadian Automated Air Traffic System), is reputed to be the largest software project in Canada. It has been under development by Hughes Aircraft of Canada since 1989 and is scheduled for delivery in late 1996. The software requirements for this system are expressed in approximately 3,100 pages of highly structured natural language. The CAATS Software Requirements Specifications (SRSs) is based on the concept of a thread, namely, ``a path through a system that connects an external event or stimulus to an output event or response" [4]. This threads-based approach differs significantly from other traditional approaches to software requirements specification such as Structured Analysis. Given that much of the software requirements specification effort for this project was nearing completion when our investigation began, we have focused our efforts on the possibility of retrofitting formal specification techniques with the existing natural language based specification documents. In this context, we have realized the importance of using a notation that supports the development of formal specifications that closely match the level of abstraction, organization, and notational conventions of a pre-existing natural language specification.

Initially we experimented with Z for capturing CAATS software requirements in a mathematical notation. In a preliminary study, we translated a very small fragment of the software requirements specification for CAATS into approximately 60 pages of liberally documented Z. Although we found that there was a very good match in style between Z specification and the threads-based approach used in the CAATS SRSs, we soon became disenchanted with Z for several reasons. Given that Z involves many non-ASCII characters and a graphical presentation format (where components of the specification are presented as combinations of text and various kinds of box-like shapes), it is not possible to directly create and edit a Z specification using a standard text editor. Instead, one must typically create and edit a Z specification indirectly in terms of a sequence of text formatting commands which, upon processing by a suitable text formatter, will produce the desired Z specification. We found this particularly cumbersome -- and, in our opinion, an approach based on Z would be difficult to retrofit into an existing software process. We also found that the semantics of Z can be difficult to explain to others who do not have prior experience with formal methods -- and sometimes, we even found it difficult within our team to reach agreement on the meaning of certain Z fragments.

We also considered some alternatives to Z. For instance, we briefly considered the use of the HOL object language (a formulation of higher order logic) and the HOL system. Although we felt that a HOL approach, given its higher-order logic foundation, offers the advantage of semantics that is easier to understand and explain to others, we also felt that it would be unreasonable to propose a practical specification methodology for use in an industrial context that would depend on the use of the HOL system as a tool for checking specifications. Since the HOL object language does not include constructs for the declaration and definitions of types and constants, one must also become familiar with

the HOL meta-language -- namely, ML -- to create formal specifications that can be checked by the HOL system. Not only does this place the additional burden of understanding of ML onto the shoulders of the specifier, but moreover, the distinction between the meta and object languages of HOL are frequently a source of confusion for new users of the HOL system.

Given that we were unable to find a notation and supporting tools that would suit our investigation into possible application of formal specification techniques in CAATS, we were motivated to develop our own notation and tools. Our development of S and Fuss was based upon a number of objectives including:

- the notation must be based entirely on ASCII characters so that it can be edited directly using standard text editors and incorporated into other documents without processing;

- the semantics must be relatively simple so that specifications can be understood by users who do not necessarily have expertise in formal methods;

- the notation must be supported by an easy-to-use software tool that automatically checks a specification with respect to rules of syntax and typing.

In addition to these main objectives, we have incorporated a number of features into S that we found useful in our efforts to formally specify selected aspects of CAATS. Some of these features are described in the next section of this paper.

A Brief Description of S

S can be described as a superset of the HOL object language in the sense that expressions of the HOL object language are also expressions of S. There are a few minor syntactic differences. For example, the reserved words ``function", ``forall", ``exists" and ``select" are used in S instead of ``\\", ``!", ``?" and ``@" for lambda expressions, universal quantification, existential quantification and Hilbert's choice operator respectively. The notation ``if A then B else C " is used in S instead of ``A => B | C" for conditional expressions. The type operator for Cartesian product in S is ``*" instead of ``#".

The S notation also includes constructs for the declaration and definition of types and constants. In this respect, S is similar to Z which also includes constructs for the declaration and definition of types and constants. However, the inclusion of these constructs in S represents a significant difference between S and the HOL object language which does not provide constructs for the declaration of types and constants. In the case of HOL, constructs for the declaration and definition of types and constants are part of the HOL meta-language -- in particular, they are functions defined in the ML programming language which is used to implement the HOL system. We have included constructs for the declaration and definition of types and constants in S because it is essential to have such constructs to write formal specifications --- and we desire the notation to be ``stand-alone"', that is, independent of any meta-language or supporting software tool.

Declaration and Definition of Types

A type declaration in S may be used to introduce one or more new types. For example, the paragraph,

```
: flight_plan, aerodrome, customs_office;
```

is an example of a type declaration paragraph. This example introduces three new types, ``flight_plan", ``aerodrome" and ``customs_office". Types introduced by means of a type declaration paragraph are ``uninterpreted" in the sense that nothing is said about their composition. A type declaration paragraph may be used in an S specification in the same way that the declaration of one or more ``basic types" may be used in a Z specification to achieve a desired level of abstraction. A type declaration paragraph may also be used to introduced parameterized types. For example, the type declaration,

```
: (tyvar) set;
```

introduces a parameterized type, ``set", which is parameterized by a single type variable, ``tyvar".

A type definition may also be used to introduce a new type. Unlike a type declaration, the definition of a type describes its composition in terms of a set of constructors. These

constructors are implicitly declared as S constants. The constructors of a type definition are functions that map zero or more objects to objects of the new type. For example,

```
: ty := c1 | c2 :num | c3 :bool :bool*num | c4 :ty;
```

defines a new type ``ty'' in terms of four constructors, ``c1'', ``c2'', ``c3'' and ``c4''.

The type definition may be recursive -- that is, the new type may appear on the right-hand side of the type definition provided that at least one of the branches does not contain a reference to the new type. The limitations on recursive type definitions are the same as those found in HOL.

A type definition may also be parameterized by one or more more type variables. For example,

```
: (tyvar) list := NULL | CONS :tyvar :(tyvar)list;
```

is a recursive type definition parameterized by a type variable, ``tyvar''.

In addition to type declarations and type definitions, S provides a construct for the introduction of type abbreviations. In this case, a new type is not declared or defined -- instead, a new name is introduced which can be used in place of potentially more complicated type expression. For example,

```
: ty == num -> bool;
```

results in the introduction of ``ty'' as an abbreviation for ``num -> bool''.

Declaration and Definition of Constants

A constant declaration in S, for example,

```
c : num;
```

may be used to introduce one or more new constants. In this case, a new constant, ``c'', has been declared as a constant with type ``num''. Constants may also be declared to be infix constants by surrounding the name of the new constant with underscores as illustrated, for instance, by the following declaration:

```
(_ is_greater_than _) : num -> num -> bool;
```

Another way to introduce constants is by means of a constant definition. A constant definition is equivalent to the declaration of a new constant and the introduction of a definitional axiom for the new constant. For instance, the following constant definition,

```
plustwo n := n + 2;
```

is an example of a simple, non-recursive constant definition. They may be recursively defined in terms of a recursively defined type. Constant definitions, along with constant declarations, may also be parameterized by type variables. This is illustrated by the following definition,

```
(:tyvar) length NULL := 0 |
        length (CONS (x:tyvar) s) := 1 + length s;
```

which defines a new constant, ``length", in terms of a recursively defined type, ``list", whose two constructors are ``NULL" and ``CONS". The recursive definition of ``length" is parameterized by a type variable, ``tyvar". This definition also illustrates the use of pattern matching in constant definitions.

We consider the explicit declaration of type variables in declarations and definitions to be useful as a mechanism for discouraging a specifier from blindly parameterizing a declaration or definition by an excessive number of type parameters.

Templates

The S notation also includes a construct called a template which we have found to be particularly useful as a packaging mechanism when specifying software systems. The notion of a template was originally inspired by the idea of a schema in Z. But aside from the possibility of using S to write Z-like specifications, the template construct is a solution to a problem often seen in formal specifications of large systems where logical expressions grow to unreadable proportions because of long lists of parameters associated with various operators (e.g., functions and predicates).

A template can be described as a predicate with ``implicit parameters". These parameters are ``implicit" in the sense that they do not appear as actual parameters in an expression when the template is referenced. For instance, a template ``A" with two implicit parameters, ``x" and ``y", is referenced as just ``A" instead of ``A(x,y)". The names of the implicit parameters of a template are given in the definition of the template together with zero or more Boolean expressions that specify constraints on these implicit parameters and/or global constants. For example,

```
A := x,y:num; x < y;
```

is a definition of template ``A" with the constraint that ``x" is less than ``y".

A template may be referenced in any expression that follows the definition of the template. A template reference may appear anywhere that a Boolean expression may appear. For example, the expression,

```
forall y. (y = 0) ==> exists x. A
```

expresses the arithmetic fact that every natural number except zero is greater than some other natural number. In the above expression, ``x" and ``y" refer to the implicit parameters of template ``A".

Another example is the following template definition which includes a reference to template ``A":

B := z:bool; A ==> z; x > 9;

This template definition consists of a single variable declaration and two constraint expressions. The first of these two constraint expressions contains a reference to template ``A".

By means of a simple syntactic transformation called ``template expansion", any S specification can be converted into an equivalent S specification where all of the template definitions have been replaced by constant definitions, and references to the defined templates replaced by references to these constants. For example, the above template definitions would be transformed into the following two constant definitions,

A(x,y) := x < y;

B(x,y,z) := ((x < y) ==> z) / (x > 9);

and expressions such as,

forall y. (y = 0) ==> exists x. A

forall x y z. A B

would be transformed by means of template expansion into the following expressions:

forall y. (y = 0) ==> exists x. x < y

forall x y z. (x < y) (((x < y) ==> z) / (x > 9))

Thus, the template construct of S can be seen as mechanism for defining constants where the parameters are not given explicitly. Although not very interesting in theoretical terms, this construct has considerable practical value when used in the formal specification of large systems. For instance, the template construct allows a top level specification of a requirement, as illustrated by our example in Section 4, to be expressed in a manner that is not cluttered with parameters.

Another Z-like feature of S is that templates can be ``decorated" with one or more decoration symbols -- namely, ``'", ``!" and ``?". (The use of``!" and ``?" in S is entirely different than the use of these symbols in the HOL object language where they are used

as quantifiers. In S, as in Z, the decoration of a name with one of these symbols is merely a naming convention.) The decoration of a template reference has the effect, upon template expansion, of decorating each of the implicit parameters of the template and each of the template references that appear in the definition of the decorated template reference. For example, an expression such as,

forall x y. A B'

where the reference to ``B" has been decorated with a single ``'" would result, upon template expansion, in the following expression:

forall x y. (x < y) (((x' < y') ==> z') / (x' > 9))

where x', y' and z' are free.

Names of Types, Constants and Templates

Another useful feature of S is the ability to use phrases such as ``this is a very long name" as the names of types, constants and templates. With the exception of a small set of standard mathematical and logical symbols such as ``+", ``=" and ``<", a name containing characters that are neither upper/lower case letters, digits nor the underscore character must be enclosed inside a pair of matching double quotes. For example:

the successor of three := 4;

There are several reasons why we allow names in S to be arbitrary strings of characters. In part, this is a reflection of the fact that arbitrary strings are often used as names in the natural language specification of a large software system. For example, names such as,

<operator enter filed flight plan>

[filed flight plan]

\Validate IFR/CVFR Flight Plan Route\

are all examples of names that appear in the CAATS software requirements specification. In addition to the separation of words with whitespace characters, the above examples include characters such as ``<", ``>", ``[", ``]" and ``\" which have special significance with respect to naming conventions for CAATS documentation. For example, a string enclosed inside a matching pair of angle brackets, ``<" and ``>", indicates a reference to an entry in the project data dictionary. Given this use of arbitrary strings in natural language specifications, we have allowed arbitrary strings to be used as names in S specifications so that a greater correspondence between the natural language specification and its formalization in S can be achieved.

Post-Fix Function Application

A software specification frequently involves the representation of data objects with multiple attributes. For instance, the example in the next section of this paper involves the representation of flight plans in a hypothetical air traffic control system where a flight plan has a number of attributes including the names of the departure and destination aerodromes. As illustrated in the next section, these attributes can be represented formally in S as functions. For instance, we can introduce a function,

<departure aerodrome>: flight_plan -> aerodrome;

that maps a given flight plan to an aerodrome. Although the application of such functions to objects can be expressed in terms of simple juxtaposition,

<departure aerodrome> new_flight_plan

we have found that a specification can be made more readable by using a post-fix form of function application where the operator appears after the operand and is separated from the operand by a dot:

new_flight_plan.<departure aerodrome>

This post-fix form of function application suggests the idea of a record type with a field called ``<departure aerodrome>" -- or in the case of an object-oriented approach, the informal idea of an ``flight_plan" object with ``<departure aerodrome>" as one of its attributes. Mathematically, this syntax is nothing more than post-fix application of a function to a value, but this syntax is very helpful when attempting to draw upon the intuition of specification readers familiar with these more established kinds of specification and programming notations.

Example

In this section, we illustrate the use of S as a notation for the formalization of a software requirement for a hypothetical air traffic control system. Our example is motivated by our familiarity with the CAATS software requirements specification -- however, this fragment is not intended to be a representation of any requirement in the CAATS software requirements specification.

Consider the following natural language specification of a software requirement for part of the system responsible for processing flight plans as they are entered into the system.

Requirement 57:

``If a new flight plan has been entered and the <departure aerodrome>of the new flight plan is outside Canada and the <destination aerodrome> is inside Canada, then send the new flight plan to the Canada Customs office serving the <destination airport>".

We begin our formalization of this requirement by focusing on the high level logical structure of the above natural language sentence -- in particular, by identifying the main logical connectives of the requirement, breaking up the natural language specification into phrases, and re-assembling these phrases into an S expression based on the high level logical structure of the sentence. In this case, the natural language specification can be broken up into four separate phrases,

> ``a new flight plan has been entered"
>
> ``the <departure aerodrome> of the new flight plan is outside Canada"
>
> ``the <destination aerodrome> of the new flight plan is inside Canada"
>
> ``send the new flight plan to the Canada Customs office serving the <destination aerodrome>"

and then assembled into an expression using three logical connectives that we can define in S -- namely, NOT for logical negation, AND for logical conjunction and a ``split" connective, if ... then ..., for logical implication. This yields the following constant definition which represents the top level formalization of the above natural language requirement:

```
Requirement 57 :=
  if (
    a new flight plan has been entered
    AND the <departure aerodrome> of the new flight plan is
      outside Canada
    AND the <destination aerodrome> of the new flight plan is
      inside Canada)
```

```
  then
    send the new flight plan to the Canada Customs office serving
     the <destination aerodrome>;
```

In the above constant definition, each of the four phrases is formalized as a template reference. The second step in the formalization of this requirement is the definition of the four phrases as templates:

```
a new flight plan has been entered :=
  {
    source : sender;
    new_flight_plan : flight_plan;
    source has_sent (new_flight_plan_message(new_flight_plan))
  };

the <departure aerodrome> of the new flight
       plan is outside Canada :=
  {
    new_flight_plan : flight_plan;
    NOT(is inside Canada
          (new_flight_plan.<departure aerodrome>))
  };

the <destination aerodrome> of the new flight
       plan is inside Canada :=
  {
    new_flight_plan : flight_plan;
    is inside Canada(new_flight_plan.<destination aerodrome>)
  };

send the new flight plan to the Canada Customs office serving the
 <destination aerodrome> :=
  {
    new_flight_plan : flight_plan;
    new_flight_plan_message(new_flight_plan) is_sent_to
      (new_flight_plan.<destination aerodrome>.customs_office)
  };
```

The third step in the formalization of this hypothetical requirement is the declaration of constants and types used in the above template definitions. In particular, this involves the declaration of the following constants,

```
(_ has_sent _) : sender -> message -> bool;

(_ is_sent_to _) : message -> receiver -> bool;
```

new_flight_plan_message : flight_plan -> message;

<departure aerodrome> : flight_plan -> aerodrome;

<destination aerodrome> : flight_plan -> aerodrome;

is inside Canada : aerodrome -> bool;

customs_office : aerodrome -> receiver;

which, in turn, are based on the declaration of the following types:

: flight_plan, message, sender,
receiver, aerodrome;

Since these types and constants are likely to be used to specify other related requirements, the declaration of these types and constants would be part of the supporting infrastructure shared by various components of the formal specification.

Fuss -- A Tool for Specification in S

To support the use of S as a notation for formal specification, we have developed a prototype implementation of a tool called ``Fuss'' that checks an S specification for conformance to rules of syntax and typing. The tool may be used in batch mode (in the same way that Fuzz may be used to check a Z specification). Fuss may also be used in an interactive manner in combination with a text editor to incrementally develop an S specification. A user may interactively query Fuss about the current state of the S specification under development -- for example, a user can inquire about the type of a particular constant as determined by a previous type definition, constant declaration or constant definition.

Fuss does not currently check that certain rules of definition are satisfied by type definitions and constant definitions. However, we expect to extend Fuss to make these additional kinds of checks.

In addition to its function as a specification checker, Fuss builds an internal representation of an S specification that can be accessed through a C programming language interface. Thus, Fuss can be used as the front-end for user-defined applications that accept S specifications as input and apply some specific processing or transformation to the S specifications. An example of a user-defined application would be a tool that transforms an S specification into the specification notation of a verification tool. The `S to HOL' translator described in the next section is implemented in this manner.

S and HOL

We have developed an extension to Fuss that translates an S specification into an ML file --- the meta-language of the HOL system -- which can then be used as input to the HOL system. It generates a theory consisting of declarations and definitions of types and constants corresponding to the declarations and definitions of types, constants and templates of the S specification. The first step of this transformation involves the replacement of template definitions and template references with constant definitions and constant references as described earlier in Section 3.3. The rest of the transformation is straightforward given the fact that S and the HOL object language share a common logical foundation, i.e., higher-order logic and moreover, a very similar syntax. When an S specification involves names that would be rejected by the HOL system --- for example, the use of ``<destination aerodrome>" as the name of a constant -- these are transformed algorithmically into a name that will be accepted by the HOL system.

Currently, this extension serves as a ``one-way" interface from S to HOL. We plan to eventually implement a ``HOL to S" extension to the HOL system so that the declarations and definitions of a HOL theory and its ancestors could be translated into S and dumped into an input file for Fuss. This would allow a S specification to be created on top of some existing HOL infrastructure such as a HOL library.

The transformation of an S specification into an ML file for use as input to the HOL system allows the HOL system to be used as a verification tool for an S based approach to formal specification and verification of a software system.

In CAATS, we are experimenting with the use of the HOL system as a mechanism for generating test cases for a major part of the CAATS development program called System Integration Testing (SIT). In this approach, test cases are formally derived as logical consequences of one or more software requirements. The use of HOL for this purpose, as a means of increasing the level of automation in this aspect of developing a large software system, may offer very significant benefits in terms of cost reduction over the traditional approach to test case generation which is manually intensive and prone to human error. Our initial experiments with this approach have been very encouraging and we are now examining how this technique could be developed into a production process.

Related Work

Although we regard S as an independent notation, its development is clearly related to efforts by others which may be described as efforts to combine Z with the HOL system. ICL [7] has developed an ``industrial strength'' version of the HOL system, called ProofPower, which includes a ``deep'' semantic embedding of Z in the underlying logical framework of the HOL system, namely, higher-order logic. A simpler approach taken by Bowen and Gordon [1] is based upon a ``shallow'' embedding of a subset of Z for the HOL system. Both of these approaches demonstrate the possibility of a semantic link between Z and the HOL system which allows a Z specification to be mechanically translated into a representation within the logical framework of the HOL system. One possible problem with this kind of approach is that a complete and ``safe'' Z-to-HOL translation process may depend on a substantial amount of logical infrastructure to bridge the considerable gap between the mathematical foundation of Z and the mathematical foundation of the HOL system. While it may be possible to create this infrastructure, it is unclear to us how successfully this infrastructure can be hidden to allow the HOL system to be used in a practical way to reason about a Z specification. This kind of approach may be contrasted with our scheme with translating S into input for the HOL system which only involves a tiny amount of infrastructure since both S and the HOL system share a common mathematical foundation, namely, higher-order logic.

VDM and Larch are two other formal specification languages that can be used for similar purposes as S. As with Z, VDM uses some built-in non-ASCII symbols as operators. In Larch, the user defines their own set of symbols possibly drawing from a library [12]. This means that like S, no special, non-ASCII symbols have to be used.

In both VDM and Larch the specification is usually formatted in terms of pre- and post-conditions explicitly unlike Z or S. A Larch specification is further partitioned into an interface specification, giving pre- and post-conditions for each operation, and a ``trait'', where the function symbols are defined algebraically [12]. With S, there is more flexibility to adapt the format of the specifications to existing styles when retrofitting formal specification techniques into an on-going project.

Future Work

We are now considering extensions to S and Fuss which would allow descriptive (extra-logical) information to be associated with fragments of an S specification. For instance, this may include the ability to annotate fragments of S specifications with information that records links to ``higher level'' requirements and other project information.

In addition to some of the planned refinements of Fuss already mentioned in this paper, we are interested in the possibility of providing support for the execution of an ``operational subset'' of S.

We also plan to investigate the integration of S with a state machine formalism which is better suited to modeling concurrent components. SpecCharts [9] is a graphical, hierarchical formalism similar to Statecharts [6] except that the actions take place within a state (known as a behaviour in SpecCharts). These actions are described in VHDL. For higher levels of abstraction or software systems, S would be a better language for describing the actions than VHDL. By using an executable subset of S, it would be possible to model check requirements of these specifications similar to the manner in which a model checker was created for Statecharts [3].

Another possibility is to develop extensions to S that allow decision tables in a graphical format which could be compiled into S for analysis purposes.

Conclusion

S, together with support provided by Fuss, offers several advantages over the development of formal specifications directly in the HOL object language using the HOL system as a specification checker. For instance, the Fuss tool can be used effectively to check a specification without any expertise beyond an understanding of the S notation. This contrasts with the use of the HOL system as a specification checker which requires some understanding of the HOL system meta-language in addition to an understanding of the HOL object language. Another very important practical advantage of S is the fact that the Fuss system is a relatively small C program which we expect will be easily ported to a wide variety of platforms including those often found in engineering environments such as PCs.

Similarly, S and Fuss offer several advantages over the use of Z and Z-based specification checking tools such as Fuzz. For instance, one very important advantage -- particularly, when one considers the possibility of using formal specifications in a large project involving non-academic document preparation tools -- is that S is based entirely on printable ASCII characters. An S specification can be created directly using an ordinary text editor and incorporated into other documents without any special text formatting tools. This contrasts with Z where one must typically create a machine-readable specification in terms of a sequence of text formatting commands which, after processing by a suitable text formatting tool, results in a graphical representation of the Z specification. This is a benefit for either retrofitting formal specification techniques into existing requirements documents or when starting a new project since the supporting natural language description is very important for interpreting the meaning of the formal specification.

Another advantage of S is the existence of a verification tool with a substantial user community, namely the HOL system, which can be used to reason almost directly about an S specification using the same proof mechanisms that would likely be used to reason about the same specification expressed directly in the HOL object language.

Therefore, we believe S satisfies our goal of creating a specification language for the practical application of formal methods in industrial scale projects while leaving the door open to apply current verification techniques to specifications written in this language. So far, S, and the benefits of formal methods in general, have been welcomed with interest at Hughes. We have given fragments of specifications written in S to Hughes employees with no knowledge of formal methods and they have been able to understand them. Our next step is to see how Hughes employees react to working with the notation themselves.

Acknowledgements

In addition to the authors of this paper, work was completed on this project by S. Kahan, M. Wong Cheng In, and Z. Zhu. This work has been supported by funds from the B.C. Advanced Systems Institute, Hughes Aircraft of Canada and the Natural Science and Engineering Research Council of Canada.

References

1) Jonathan Bowen and Mike Gordon. Z and HOL. Draft copy.

2) D. Craigen, S. Gerhart and T. Ralston. An International Survey of Industrial Applications of Formal Methods (2 Volumes). Technical Report #NRL/FR/5546-93-9581, Naval Research Laboratory, Washington, D.C. .

3) Nancy Day. A Model Checker for Statecharts. Technical Report 93-35, Department of Computer Science, University of British Columbia, October, 1993.

4) Michael S. Deutsch and Ronald R. Willis. Software Quality Engineering - A Total Technical and Management Approach. Prentice Hall Series in Software Engineering, Englewood Cliffs, New Jersey, 1988.

5) M. J. C. Gordon and T. F. Melham (eds.,). Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, 1993.

6) David Harel. Statecharts: A visual formalism for complex systems. Science of Computing, 8:231-274, 1987.

7) R.B. Jones. ICL ProofPower. BCS FACS FACTS, 1(1): 10 13, 1992. Series III.

8) J. Joyce and N. Day. S: A General Purpose Specification Notation. In preparation.

9) Sanjiv Narayan, Frank Vahid, and Daniel D. Gajski. System Specification with the SpecCharts Language. IEEE Design and Test of Computers, pages 6-13, December, 1992.

10) J.M. Spivey. The Z Notation: A Reference Manual. 2nd edition, Prentice-Hall, 1992.

11) J.M. Spivey. The fuzz Manual. 2nd edition, Computer Science Consultancy.

12) Jeannette M. Wing. A Specifier's Introduction to Formal Methods. Computer, 23(9):8-22, September, 1990.