NASA-CR-192293

Formalizing Structured File Services for the Data Storage and Retrieval Subsystem of the Data Management System for Spacestation Freedom

GRANT
IN-82-CR
148097
P. 23

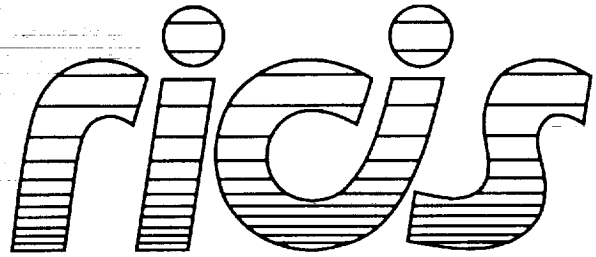
Damir A. Jamsek

Odyssey Research Associates Corporation

February 8, 1993

Cooperative Agreement NCC 9-16
Research Activity No. IR.04

NASA Johnson Space Center
Information Systems Directorate



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

FINAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

Formalizing Structured File Services for the Data Storage and Retrieval Subsystem of the Data Management System for Spacestation Freedom

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Damir A. Jamsek of Odyssey Research Associates Corporation. Dr. Charles Hardwick served as RICIS research coordinator.

Funding was provided by the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Robert B. MacDonald, Manager, Research, Education, and University Programs, Technology Development Division, Information Systems Directorate, NASA/JSC.

Formalizing Structured File Services for the Data Storage and Retrieval Subsystem of the Data Management System for Spacestation Freedom

Damir A. Jamsek
Odyssey Research Associates Corporation
301 Dates Drive
Ithaca, NY 14850

February 8, 1993

1 Introduction

This report presents a brief example of the use of formal methods techniques in the specification of a software system. The report is part of a larger effort targetted at defining a formal methods pilot project for NASA. This report's purpose is to present one possible application domain that may be used to demonstrate the effective use of formal methods techniques within the NASA environment. It is not intended to provide a tutorial on either formal methods techniques or the application being addressed. It should, however, provide an indication that the application being considered is suitable for a formal methods by showing how such a task may be started.

The particular system being addressed is the Structured File Services (SFS), which is a part of the Data Storage and Retrieval Subsystem (DSAR), which in turn is part of the Data Management System (DMS) onboard Spacestation Freedom [1]. This is a software system that is currently under development for NASA.

An informal mathematical development is presented in the section 2. Section 3 contains the same development using Penelope [23], an Ada specification and verification system. The complete text of the English version Software Requirements Specification (SRS) is reproduced in Appendix A.

2 An Informal Mathematical Model

In beginning the task of formally describing a computer system, a specifier of the system will initially develop an informal mathematical model. In developing the informal model the specifier will naturally use common intuitive notions from mathematics. Set theory and function theory will be used to describe the behavior of the SFS. Describing the system in terms of its state and how the system may transition from one state to another is also a common technique that we will use. While the use of these techniques will be informal and ad hoc, the next section will use a formal notation for describing the SFS that reflects the developments in this section.

We begin by examining the processing requirements placed on the SFS. These are captured in a series of *shalls* written in plain english text.

Shall 1 *SFS shall (1) create a structured file upon request from the software user.*

The first *shall* introduces several terms. To model these terms, we introduce several definitions.

- The set $USER$ is the set of SFS users.
- The set SFS_FILE is the set of all SFS files.
- The set $FILE\_NAME$ is the set of SFS file names.

Now a function, $create\_file$, can be defined. This function, when supplied a user and a filename, returns a valid SFS file. The domain and range of the function are defined as

- $create\_file : USER \times FILE\_NAME \rightarrow SFS\_FILE$.

As additional elements of the SFS model are defined, the behavior of the function $create\_file$ may then be defined.

The SFS files are part of the state of the SFS system. As the system executes and user requests are serviced, this state will change. To describe these changes in state, various components, as well as the overall state, are given names. We call the entire state, sfs_state. The SFS files are called sfs_files and are a component of sfs_state. Also the SFS file names are sfs_file_names and are also a component of the state.

The operation of creating a file upon user request will certainly change the state of the SFS. Before defining this change in state, it is worthwhile to note that the SFS, later in the SRS, specifies other user services that may be requested. These correspond to the input DDFS_SFS specified in the SRS as input to the SFS. Let $DDFS\_SFS \equiv \{create, open, read, write, delete, dir\}$ be this set of requests. The set $DDFS\_SFS$ captures the fact that these various requests exist, but does not indicate how these requests may effect the system.

Now, for some DDFS to SFS request, $create \in DDFS\_SFS$, we can specify some change in the sfs_state. That is, for some, $user \in USER$, and $file\_name \in FILE\_NAME$,

- a new $file \in SFS\_FILE$ should be created,
- the state component sfs_files should be updated, and
- the state component sfs_files_names should be updated.

Additional, elements of the model must be defined in order to complete the definition of the response to a $create \in DDFS\_SFS$ request. We continue by examining *shall* number 2 in the SRS.

Shall 2 *SFS shall (2) establish a dictionary describing the structured file so that it can manage the updating of the file.*

It seems clear that an SFS dictionary of some sort should be included as part of the state, call it sfs_dict. This dictionary will provide information about SFS files, so we index it by SFS file names. When looking up the entry for a particular file, by name, we would expect in return an entry for that file which describes various aspects of the file. A function

- $lookup : SFS\_DICT \times FILE\_NAME \rightarrow DICT\_ENTRY$

serves this purpose.

When files change, it may be necessary to modify their dictionary entries. A function *update* is used to update dictionary entries. Given a dictionary and a new entry for a particular file, a new dictionary, appropriately updated will be returned. The domain and range for this function is given as
$update : SFS\_DICT \times FILE\_NAME \times DICT\_ENTRY \rightarrow SFS\_DICT$

The relationship between *lookup* and *update* is that *lookup* will always return the appropriate entry in an updated dictionary. This realtionship is formally defined in the next section.

Continuing with the *shalls* from the SRS.

Shall 3 *The dictionary shall (3) contain the number of records in the file, the record size, the record structure, the archive threshhold (prescribed percentage capacity threshhold for archiving the file), an archive/circular indicator (indicating whether the file should be archived or overwritten), and access information (user identification, password, read onlt access, write access, delete access, archive rights, etc.).*

This *shall* defines various elements of a dictionary entry. The following functions on a dictionary entry are introduced.

- $nrec : DICT\_ENTRY \rightarrow \mathcal{N}$
- $recsize : DICT\_ENTRY \rightarrow \mathcal{N}$
- $recstruct : DICT\_ENTRY \rightarrow \mathcal{N}$
- $archive : DICT\_ENTRY \rightarrow \mathcal{B}$
- $access : DICT\_ENTRY \rightarrow ACCESS\_INFO$

where $\mathcal{N} \equiv \{0, 1, 2, \ldots\}$ and $\mathcal{B} \equiv \{true, false\}$

An element of the set $ACCESS\_INFO$ is a set of access rights granted to users for a particular file. These access rights are defined by the set $ACCESS\_RIGHTS \equiv \{read, write, delete, execute\}$. So, we may define $ACCESS\_INFO$ as the set of all subsets of $ACCESS\_RIGHTS$, or equivalently, $ACCESS\_INFO \equiv \mathcal{P}(ACCESS\_RIGHTS)$.

To this point, not much more than the naming of various components of the model has taken place. In addition to naming components, we have also indicated some structure and nature of each of the components. As development of the model proceeds this structure would be more fully defined and eventually yield a complete mathematical model of the intended structure of the entire system. The structure would then be suitable for various analysis, refinement and documentation purposes. This short example only serves to give a flavor for a style of development possible.

Given the start of an informal mathematical model, a system designer may decide to record his design formally. The next section addresses the same issues as this section, only in the framework of a formal mathematical notation.

3 Developing a Formal Mathematical Model

The previous section developed a model of the SFS using a relatively informal style of mathematical presentation based on commonly understood mathematical notions of set and function theory. This section addresses essentially the same portions of the specification but uses a formal mathematical notation. The notation used is taken from the Penelope Ada specification and verification system developed at Odyssey Research Associates [23]. It is a system based on the Larch style of specification and is similar in its mathematical style to systems such as EHDM, LSL, PVS. That is, its specifications are algebraic in nature. Penelope has the additional ability to directly address issues of Ada code specification and verification. The details of Penelope are not presented here and the specification is presented simply as a demonstration of the feasibility of applying mechanical means to the specification and verification of systems such as the DSAR SFS.

In the following treatment a basic style of algebraic specification is used to capture the meaning of statements in the SRS. English text provides an informal description of each component of the specification. In the following treatment the Larch construct called a *trait* is used to encapsulate a particular theory. That is, a set of related facts which form a logical structure and from which further facts may be deduced. The algebraic form of specification in Larch uses *sorts* and *operators* to define logical theories. Sorts and collections of objects. Operators define functions from sorts into sorts.

In the previous section, sets were used to describe various aspects of the model. The notion of a set needs to be formally defined by introducing operators and sorts which capture the usual behavior of sets. In this case, *empty*, and *member* are alternate notations for the usual notions $\{\}$ and $\in$, respectively.

```
--| trait Set is
--|  introduces
--|   empty: -> Set;
--|   insert: Element, Set -> Set;
--|   member: Element, Set -> Bool;
< ... additional set operations omitted ...>
--|  asserts
--|    Set generated by empty, insert
--|    Set partitioned by member
--|   axioms: forall  [e:Element, e1:Element, s:Set]
--|    not_member_empty: (not member(e, empty()));
--|    member_insert: (member(e, insert(e1, s))=
                              ((e=e1) or
                               member(e, s)));
< ... axiomatization of additional set operations omitted ...>
--|   end axioms;
```

Additional set manipulations constructs such as set union and intersection may be similarly defined. Basic mathematical notions such as sets, sequences, lists, and stacks, would normally be provided in a collection of theories gathered in a library and available for use by the specifier. This theory of sets is developed in a generalized form and later used (and reused) in several specialized forms.

The following trait defines a general theory of dictionaries that will be used subsequently. A dictionary is some object to which the operations *update* and *lookup* may be applied and in which update and lookup have the appropriate relation to each other. In addition, the operation *defined* may be used to check the definedness of a particular index.

```
--| trait Dictionary is
--|  introduces
```

```
--|   empty: -> Dict;
--|   update: Index, Ent, Dict -> Dict;
--|   lookup: Index, Dict -> Ent;
--|   defined: Index, Dict -> Bool;
--|  asserts
--|    Dict generated by empty, update
--|    Dict partitioned by lookup
--|   axioms: forall  [i1, i2:Index, e1, e2:Ent, d1, d2:Dict]
--|    not_defined: (not defined(i1, empty()));
--|    defined: (defined(i2, update(i1, e1, d1))=
                    ((i1=i2) or
                     defined(i2, d1)));
--|    lookup: (lookup(i2, update(i1, e1, d1))
  =
    (if (i1=i2)  then e1 else lookup(i2, d1)));
--|   end axioms;
--|   lemmas: forall  [i1, i2:Index, e1, e2:Ent, d1, d2:Dict]
--|    lookup_same: (lookup(i1, update(i1, e1, d1))=e1);
--|    proof:
--!    BY synthesis of FORALL
--!    BY synthesis of FORALL
--!    BY synthesis of FORALL
--!    BY lookup in trait Dictionary
--!    substituting for left
--!    BY synthesis of TRUE
--|   end lemmas;
```

While this theory is developed in general in terms of Index, Entry, Dict, etc. The use of the theory will be in terms of file names and SFS files, as in Section 2. In fact, it may eventually be reused in various specialized forms.

The trait *Dictionary* not only introduces some new concepts related to dictionaries but also contains the proof of a property *lookup_same* which says that after updating a dictionary entry, looking it up will return the correct entry. Proofs of properties that follow from basic definitions can be considered a form of design verification.

The SRS specifies various items as input to the SFS. One of these is a DDFS to SFS message or data item. The DDFS_SFS trait defines the possi-

ble DDFS to SFS operations, *create*, *open*, *read*, *write*, *dir*. The behavior of these operations is not yet defined.

```
--| trait DDFS_SFS is
--|  introduces
--|   sort DDFS_SFS is enumeration (create, open, read, write, dir)
```

The enumeration construct defines a particular sort and its individual elements much as in section 2 we used the set notation, $DDFS\_SFS \equiv \{create, open, read, write, delete, dir\}$ to define DDFS_SFS.

In response to the DDFS to SFS operation a message from the SFS to the DDFS is required. The trait SFS_DDFS specifies some messages from the SFS to the DDFS in response to the DDFS to SFS operations. Again, the enumeration construct is used to define the elements of the sort SFS_DDFS.

```
--| trait SFS_DDFS is
--|  introduces
--|   sort SFS_DDFS is enumeration (create_ok, create_error, ...)
<... more SFS_DDFS message specifications elided ...>
```

Access rights to files are granted to users of the SFS. These access rights, *read*, *write*, and *delete*, are defined in the next trait.

```
--| trait AccessRights is
--|  introduces
--|   sort AccessRight is enumeration (read, write, delete)
```

Various information, beyond simply access rights, must be maintained for files in the system. The trait AccessInfo defines what this information is and how it can be manipulated. The trait AccessInfo makes use of previously defined traits for Sets and AccessRights.

```
--| trait AccessInfo is
--|  includes (AccessRights)
--|  includes (Set)(AccessRight for Element,
                   UserAccessRights for Set)
--|  includes (Set)(UserAccessInfo for Element,
                   AccessInfo for Set)
--|  introduces
--|   sort UserAccessInfo is tuple (User, Password, UserAccessRights)
```

The trait AccessInfo shows how previously defined concepts can now be effectively used to build up more complex theories. In this case, the trait AccessRights is included in its original form. Because we would also like to manipulate sets of AccessRights, we include the trait Set, renaming the general sort Element with the specific sort AccessRights. Similarly, for UserAccessInfo, the trait Set is included. The tuple construct defines the sort UserAccessInfo to be a triple, the first element taken from the sort User, the second from the sort Password, and the last from the sort UserAccessRights.

The SFS Dictionary is a specialization of the general theory of dictionaries defined in the trait Dictionary. It is specialized to the case where entries are file information and the indexes are file names. Additionally, other file information operators are defined which describe the types of file information being retained in the dictionary.

```
--| trait SFS_DICTIONARY is
--|  includes (Dictionary)(Filename for Index,
                          FileInfo for Ent,
                          SFS_Dict for Dict)
--|  includes (AccessInfo)
--|  introduces
--|   rec_count: FileInfo -> Int;
--|   rec_size: FileInfo -> Int;
--|   rec_structure: FileInfo -> RecStruc;
--|   arch_thresh: FileInfo -> Int;
--|   arch_circ: FileInfo -> Bool;
--|   access_info: FileInfo -> AccessInfo;
```

The SFS state information can now be defined by a trait, *SFS_STATE*. The components of the *SFS_State* defined in this trait are *SFS_Dict* and *SFS_Files*. There is a distinguished state called *sfs_initial_state* about which the axiom *sfs_initial_state_dict* states that "the initial state contains an empty dictionary". In such a way, information about the desired initial state of the system and any other states in the system may be recorded. For example, the intent of the predicate *sfs_valid_state* is to specify which states in the system are valid.

```
--| trait SFS_STATE is
```

```
--|  includes (SFS_DICTIONARY)
--|  introduces
--|   sort SFS_State is tuple (SFS_Dict, SFS_Files);
--|   sfs_initial_state: -> SFS_State;
--|   sfs_valid_state: SFS_State -> Bool;
--|   axioms:
--|    sfs_initial_state_dict: (sfs_dict(sfs_initial_state())=
                                empty());
<... definition of sfs_valid_state omitted ...>
--|   end axioms;
```

The SFS state information and the messages to and from SFS and DFSS can now be incorporated into the trait SFS which will define the processing requirements for SFS. The function *sfs_operation* when axiomatized will define how a particular user can operate on a particular file with a particular operation in a given state of the system. The result is the new state of the system. The function *sfs_result* defines the resultant message when the operation is attempted.

```
--| trait SFS is
--|  includes (SFS_STATE)
--|  includes (SFS_DDFS)
--|  includes (DDFS_SFS)(Operation for DDFS_SFS)
--|  introduces
--|   sfs_operation: User, Filename, Operation,
                          SFS_State -> SFS_State;
--|   sfs_result: User, Filename, Operation,
                        SFS_State -> SFS_DDFS;
<... axioms for operations omitted ...>
```

Using the mathematical model developed so far, which culminated in the trait SFS, a specification of an Ada package can be written. This specification begins to define the actual Ada constructs, types, objects and functions which will eventually realize the SFS.

```
--| with trait SFS ;
package SFS is
```

```
  --| based on SFS_State
type sfs_filename is private ;
  --| based on Filename;
type sfs_username is private ;
  --| based on User;
type sfs_ddfs_message is private ;
  --| based on SFS_DDFS;
function sfs_create_file(user : in sfs_username;
                         file : in sfs_filename)
              return sfs_ddfs_message;
--| where
--|      Global SFS : IN OUT;
--|      in sfs_valid_state(SFS);
--|      out sfs_valid_state(SFS);
--|      out (SFS = sfs_operation(user, file, create(),in SFS));
--|      return sfs_result(user, file, create(),in SFS);
--| end where;

<... remaining SFS functions, procedures, objects are omitted ...>
end SFS;
```

The Ada package SFS has associated with it a state, SFS, which given the annotation above, is based on the sort *SFS_State* in the sort SFS. The behavior of function and procedure in the package SFS will be partially defined in terms of their effect on this state.

The types introduced are *private* to the package being defined. That is, their actual implementation is not important to the user of the package. However, we note that they are *based on* elements of the mathematical description of the SFS in sort SFS. From this, it is possible to deduce aspects of their behavior.

The single function *sfs_create_file* is also specified in terms of the mathematical objects in the specification. Paraphrased, the specification reads something like: "The function *sfs_create_file* modifies the state of package SFS. On entry to the function the state of SFS must be valid and on exit the new state of SFS must be valid. The new value of the state is defined by the *sfs_operation* function when the desired operation is *create* and the *user* is returned the value of the *sfs_result* function."

This package specification may now serve two purposes.

1. It provides a clear description of the desired program in a manner which is independent of the actual code design of the software. That is, it is based solely on the mathematical description of the design. A designer and developer of the actual code may refer to the mathematical description of the objects being coded whenever a question arises as to the specification's intent.

2. Developed code may be verified against the mathematical description of the code to provide a greater assurance in the correctness of the developed code.

4 Conclusions and Recommendations

This example is meant to show a possible style of specification which may be applied to the SFS software component. The formal specification would provide clear, unambiguous specifications which would be suitable for various kinds of analysis, both mechanical and human.

The presentation was necessarily short and incomplete. Its intention was to provide an insight into the appropriateness of formal methods techniques, as well as the suitability of the DMS DSAR function as a target application.

A formalization of the DSAR requirements appears to be feasible and would result in a clear specification of its functionality. The application of formal methods techniques to this software system does not seem to be a high risk endeavor. Therefore it would seem a likely choice for a pilot project in formal methods at NASA.

The main areas that the formalization would address include:

- Input, output, and state data format specifications,
- Functional description of user interfaces to the DSAR,
- Interaction of functional components of the DSAR.

The most difficult issues to be addressed in a formalization of the requirements should not be overlooked.

- There are concurrency and multiprocessing requirements placed on the software. An appropriate technique for mathematically describing concurrency must be selected or developed.

- The interface to operating system services must be formalized. This would entail formalizing at least a portion of the Ada/OS runtime environment.

Finally, it should be made clear what the intention of the formalization effort is:

- To provide a clear, concise specification of the system that can be used by system designers in further developing the system, and to serve as a reference point when ambiguities or conflicts arise in the system design and implementation.

- To provide a formal model of the system from which properties of the system may be formally verified. In this case, what those properties are still needs to be determined.

- To provide a basis for the formal design and verification of the actual implementation of the system. This would entail code verification in an effort to increase assurance in the final software product.

For each of these goals, a formal treatment of the software requirements for DSAR will yield a software system with a higher degree of assurance in its correctness.

A Software Requirements Specification

The following software requirements specification is taken from the *Data Management System (DMS) Data Storage and Retrieval (DSAR) Software Requirements Specification (SRS)*. It the the section that addresses processing requirements for the Structured File Services (SFS) component of the DSAR. It is presented here unedited and in its entirety.

A.1 4.1.2.1.1 INPUTS

Name	Description	Source
DDFS-SFS	The distributed structured file request from a DMS service user or a directory list request	DDFS
DFTSS-SFS	Response from an archive request	DFTSS
ST.FILE-SFS	Input Structured File	ST.FILE

A.2 4.1.2.1.2 PROCESSING

SFS shall (1) create a structured file upon request from the software user. SFS shall (2) establish a dictionary describing the structured file so that it can manage the updating of the file. The dictionary shall (3) contain the number of records in the file, the record size, the record structure, the archive threshhold (prescribed percentage capacity threshhold for archiving the file), an archive/circular indicator (indicating whether the file should be archived or overwritten), and access information (user identification, password, read onlt access, write access, delete access, archive rights, etc.). SFS shall (4) inform the requestor that the file has been created.

The SFS shall (5) accept write requests for the structured file. If the file size has exceeded the threshhold, SFS shall (6) automatically request that the file be transferred to the archive via the DMS File Transfer Service or begin overwriting the file file depending on the archive/circular indicator. If a file is to be archived, the SFS shall (7) ensure that no structured file processing interferes with archiving the file.Then, SFS shall (8) write the data to a 2KB buffer in main memory. When the buffer is full, SFS shall (9)

write the contents to the mass storage device (MSD). SFS shall (10) notify the requestor of the actions taken.

SFS shall (11) accept requests to open, close, read, or delete a structured file, or provide a directory list. A directory list is a file containing the list of files and the attributes of files contained in a designated directory. SFS shall (12) provide access to structured files via a key to be: record number, time, date, date and time, or a range of dates and/or times. SFS shall (14) permit access to individual field(s) within a record. SFS shall (15) provide for on-demand read requests. SFS shall (16) provide a locking mechanism to protect the integrity of the data. If a read or write request for a locked file is received, SFS shall (17) not honor the request and shall (18) inform the requestor that the file is locked.

SFS shall allow multiple concurrent access to centralized structured files. SFS shall (20) provide for distributed access to structured file services from SDPs and MPACs on core and payload networks. The interface to SFS shall (21) be such that the requestor need not know where the structured files are located or the detail of the detail of the mechanisms used to perform the structured file services.

For all of the requests described above, SFS shall (22) determine if the requestor has authorization to access the structured file in the manner requested. If access is not authorized, the SFS shall (23) deny access and inform the requestor of that fact. SFS shall (24) maintain a log of denied acesses and report denied access to predesignated MPAC.

SFS shall (25) accept requests for structured file services from the distributed interface for DMS service users. When the service is complete SFS shall (26) transmit the requested information or status to the requestor. If the service cannot be performed, SFS shall (27) transmit the reasons to the requestor.

SFS shall (28) honor requests for structured file services based on input priority level of the requests. SFS shall (29) provide for concurrent use of the files.

SFS shall (30) maintain a Structured File Dictionary containing a record of all structured files using information provided when a structured file is created.

SFS shall (31) use the file services of the OS/Ada RTE CSCI for interfacing to the MSD. SFS shall (32) SFS shall provide for a number of retries for reads and writes when I/O errors are encountered. The number of retries

will be provided in the detailed design.

SFS shall (33) transmit to SM a service usage notification for logging of SFS activities. SFS shall (24) maintain a log of all errors resulting errors resulting from structured file requests in an SFS Error Log. SFS shall (35) make the information in the error log available to SM on demand or periodically.

A.3 4.1.2.1.2 OUTPUTS

Name	Description	Destination
SFS-DDFS	The response from a structured file request or a file containing a directory list request	DDFS
SFS-DFTSS	Request to transfer a structured file to the archive	DFTSS
SFS-ST.FILE	Structured File Output	ST.FILE
SFS-NHSDM	Health and Status, activity data, error log.	SM NHSDM

References

[1] *Data Management System(DSAR) Data Storage and Retrieval (DSAR) Software Requirements Specification (SRS)* SY33.1I MDC H4353 IBM Specification No. 150a442 January 2, 1990

[2] ANSI. *The Programming Language Ada Reference Manual*, 1983. ANSI/MIL-STD-1815A.

[3] E.R. Anderson, B. Di Vito, and R.M. Hart. ASOS: Information Security for Real-Time Systems. In *AFCEA West Intelligence Symposium*, 1987.

[4] E. Astesiano et al. Draft formal definition of Ada. Technical report, CRAI, DDC, 1987. Deliverable 7 of the CED MAP project.

[5] H. Barringer, J. H. Cheng, and C. B Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21, 1984.

[6] Boehm, H.-J. Side-effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems* 7, 4 (October 1985).

[7] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[8] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[9] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science XIX*, pages 19–32. American Mathematical Society, 1967.

[10] R. Gerth. A sound and complete Hoare axiomatization of the Ada rendezvous. In *Proc. 9th ICALP*, Lecture Notes in Computer Science 140, Springer Verlag, New York, 1982, pp. 252–264.

[11] Joseph A. Goguen and R. M. Burstall. *Introducing Institutions*, volume 164 of *Lecture Notes in Computer Science*. Springer Verlag, 1984.

[12] Donald Good. Revised report on gypsy 2.1 (draft). Technical report, University of Texas, July 1984.

[13] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[14] David Guaspari. Formal semantics of two-tiered specifications. Technical Report 89-35, Odyssey Research Associates, September 1989. original number 17-14.

[15] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[17] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[18] S. Katz, Z. Manna. A Closer Look at Termination. Acta Informatica 5, pp 333–352 (1975)

[19] Carl Landwehr. The rs-232 software repeater problem. *Cipher: Newsletter of the IEEE technical committee on security and privacy.*

[20] D. C. Luckham et al. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, March 1979.

[21] Luckham, D.C., and Polak, W. Ada exception handling: An axiomatic approach. *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), pp. 225–233.

[22] D. C. Luckham et al. Anna: A language for annotating Ada programs. Technical Report CSL-84-261, Stanford University, 1986. Reference Manual.

[23] C. Marceau and C.D. Harper. An interactive approach to Ada verification. *Proceedings of the 12th NBS/NCSC National Computer Security Conference*, MBS/NCSC, October 1989.

[24] D. R. Musser. Abstract data type specifications in the AFFIRM system. In *Proceedings of the Specifications of Reliable Software*, pages 47–57, April 1979.

[25] Greg Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[26] Wolfgang Polak. Program verification based on denotational semantics. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, 1981.

[27] Wolfgang Polak. Predicate transformer semantics for Ada. Technical Report 89-39, Odyssey Research Associates, September 1989. original number was 17-12.

[28] Wolfgang Polak. A technique for defining predicate transformers. Technical Report 89-53, Odyssey Research Associates, 1989. original number was 17-4.

[29] T. Redmond. Simplifier description. Technical Report ATR-86A (8554)-2, Aerospace, November 1987.

[30] Thomas Reps and Bowen Alpern. Interactive proof checking. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 36–45, January 1984. Salt Lake City, UT.

[31] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System For Constructing Language-Based Editors*. Springer-Verlag, 1988.

[32] Oliver Schoett. *Data abstraction and the correctness of modular programming*. PhD thesis, Edinburgh, 1987. CST-42-87.

[33] Edmond Schonberg and Brian Siritzky. Adasem, Static semantics for Ada. Technical report, Dept. of Computer Science, Courant Institute of Mathematical Science, 1984.

[34] Schwartz, R.L. An Axiomatic Treatment of Algol68 Routines, In *Proc. Sixth Int'l Conf. on Automata, Languages and Programming* (July 1979), Springer Verlag, New York, 1979.

[35] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[36] Yemini, S., and Berry, D.M. An axiomatic treatment of exception handling in an expression-oriented language. *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987).