Formal Specification of a Specification Library

by

Sriram K. Atreyia

B. Tech., Indian Institute of Technology, Kanpur.
(1979)

SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE

DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1982

© Massachusetts Institute of Technology

Signature of Author

______________________________

Department of Electrical Engineering and Computer Science

May 7, 1982

Certified by

______________________________

John V. Guttag

Thesis Supervisor

Accepted by

______________________________

Arthur C. Smith

Chairman, Departmental Graduate Committee

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

CCT 20 1982
Formal Specification of a Library of Specifications

by

Sriram K Atreyya

Submitted to the Department of Electrical Engineering and Computer Science on May 7, 1982
in partial fulfillment of the requirements for the Degree of Master of Science.

Abstract

While scientific subroutines, macros, and various other utility functions have been "reusable" since the early days of programming, what has often been lacking is a convenient way of locating programs in program libraries. A similar problem exists with specification libraries. A user can access specifications in specification libraries (which usually reside on a file system,) via the name of the file containing the specification. If a user doesn't know the file name, he is frequently reduced to examining file names in the hope that the file names reveal the contents of files. In this thesis we present the design of a specification library which seeks to ameliorate this problem.

Our specification library is a repository of specifications where a specifier can hope to find previously developed specifications that will be useful in developing new ones. Our library supports a mechanism for organizing the specifications in a library, operations for creating and maintaining a library, and operations for conveniently browsing through the specifications in a library. We use formal specifications as a tool for presenting our design.

Thesis Supervisor: John V. Guttag

Title: Associate Professor of Computer Science and Engineering

Keywords: formal specifications, algebraic specifications, interactive specification library, classification scheme
Acknowledgements

I am indebted to Professor John Guttag for his guidance and support. I am particularly thankful to John for painstakingly reading my drafts again and again -- his suggestions helped improve the contents of this thesis tremendously and made all this possible. I am also thankful to all the people in the SPD group for making my education at MIT a rewarding experience. I particularly wish to thank Jeannette Wing, Srivas Mandayam, Joc Zachary and Gene Stark for the many discussions that helped me clarify my ideas.

Finally, I would like to thank Mom, Dad, Shveta, Bhup, and Brij for their support.
## CONTENTS

**Table of Contents** ................................................................................................................. 4

1. **Introduction** .......................................................................................................................... 6
   1.1 Usefulness of Formal Specifications .................................................................................. 6
   1.2 Role of a Specification Library in Developing New Specification ................................. 8
   1.3 An Overview of the Structure of our Library ................................................................. 8
   1.4 Two Issues: Accessing Specifications and Browsing Operations ............................... 9
   1.5 Goals of the Thesis ........................................................................................................... 11
   1.6 Related Work .................................................................................................................. 11

2. **Specification Language** ...................................................................................................... 15
   2.1 Specification of a System ................................................................................................. 15
   2.2 Trait Specification .......................................................................................................... 17
   2.3 Including Traits ............................................................................................................. 19
   2.4 More Complex Trait Inclusion ....................................................................................... 22
   2.5 Syntactic Amenities ....................................................................................................... 24
   2.6 Routine Specification ..................................................................................................... 27

3. **Specification Library** ......................................................................................................... 32
   3.1 Classification Scheme ..................................................................................................... 32
   3.2 Bindings the Import Names of a Specification ............................................................... 35
   3.3 Installing Specifications in a Library ............................................................................. 36
   3.4 Major Abstractions in the Specification of our Library ................................................ 37
   3.5 Basic Traits .................................................................................................................... 37
   3.6 Constructing the Trait Library ....................................................................................... 43
   3.7 Specification of Maintenance Operations ....................................................................... 48

4. **Browsing Through A Library** ............................................................................................ 59
   4.1 What do we mean by Browsing ..................................................................................... 59
   4.2 Catalogs .......................................................................................................................... 62
   4.3 Modifying Trait InputSpec .............................................................................................. 65
   4.4 Compiling and Filtering Catalogs ................................................................................... 68
   4.5 Scanning through Catalogs ............................................................................................ 72
   4.6 Combining Catalogs ....................................................................................................... 75
   4.7 Displaying Catalogs ....................................................................................................... 77
   4.8 Using the CS as a RoadMap for Browsing ................................................................... 85

5. **Summary and Conclusions** ............................................................................................... 91
   5.1 Summary of the Design ................................................................................................. 91
   5.2 Building our Library on a Relational Data Base .......................................................... 95
   5.3 Dynamic Growth of the CS in our Library .................................................................... 96
   5.4 Computer Support for Writing Specifications ............................................................... 97
   5.5 Writing A Correct Specification is not Good Enough ................................................ 98
   5.6 Areas for Future Work ................................................................................................... 99

**Appendix I. Trait Integer** ...................................................................................................... 104
Appendix II. Trait String .................................................. 106
Appendix III. Mapping Frequently Used Operators Id's to Traits 107
Appendix IV. A List of Overloaded Operators ....................... 108
1. Introduction

This thesis concerns itself with the design of a library of formal specifications. The library is designed to be implemented on a computer, and to be used in an interactive user environment.

1.1 Usefulness of Formal Specifications

In recent years, specifications have become an area of important concern because of their impact on the software life cycle [3, 7, 8, 14, 11, 21]. Informally speaking, developing specifications for a software system involves identifying the requirements\(^1\) of the software system before attempting any implementation of it, and a specification is a description of these requirements in some specification language.

Existing specification languages vary from being highly formal at one end of the spectrum to natural languages such as English at the other end. Examples of some formal specification languages are Bicycle [17], OBJ [12], Z [1], SPECIAL [24]. A formal specification language has a formal underlying model, and the specifications in a formal language (i.e. formal specifications) have an unambiguous interpretation based on the semantics of this formal model. Existing formal specification languages are primarily of interest to researchers, and are not widely used in the industry.

Most specifications written in the industry tend to be either semi-formal or informal [2]. By semi-formal specifications we refer to specifications that are partly written in a formal language, and partly in some natural language such as English. Some examples of semi-formal

---

1. The literature in the area often makes a distinction between at least two kinds of specifications: requirements specification which characterizes the requirements of the problem from a client's point of view, and design specification which characterizes the software that intends to solve the problem [3].
specification languages are PSL [26], PDL [6], SSL [5], SPECLE [2].

Though informal specifications can be read, written, and understood by a larger number of people (as compared to formal specifications,) an absence of an underlying formal model prevents informal specifications from being checked by a machine (computer) for properties such as

1. incompleteness -- a function that the software is to perform may not be completely specified. This may force designers (or programmers) to make assumptions which may prove incorrect.

2. inconsistency -- parts of a specification imply contradictory properties.

Formal specifications, on the other hand, because of the formal model underlying the formal specification languages, can sometimes be checked by a machine for syntactic and semantic properties.

One obstacle in the utilization of formal specifications as a practical methodology for developing large software is the scarcity of tools for their construction, analysis, and use [16]. This thesis describes the design one such tool, which we call a specification library. Informally, a specification library is a repository of specifications where a specifier can hope to find previously developed specifications that will be useful in developing new ones.

In section 1.2 we discuss the usefulness of a library of specifications in developing new specifications. In section 1.3 we discuss certain requirement imposed on the structure of our library by the kind of formal specification with we shall be concerned with in this thesis. In section 1.4 we discuss two major issues that are relevant to the design our library. Finally, in sections 1.5 and 1.6 we outline the goals of the thesis and some related work.
1.2 Role of a Specification Library in Developing New Specification

Most veteran specifiers (like veteran programmers) develop new specifications by examining similar specifications they have developed in the past, and borrowing ideas and/or components from those specifications. The formal specifications with which we shall deal in this thesis are particularly suited to this idea of using previously written specifications in developing new ones. This idea will be explained in detail in chapter 2 when we discuss our specification language.

While scientific subroutines, macros, and various other utility functions have been "reusable" since the early days of programming, what has often been lacking is a convenient way doing so. One of the problems with existing program libraries is locating programs in them. A user can access programs in program libraries (which usually reside on a file system,) via the name of the file containing the program. If a user doesn't know the file name, he is frequently reduced to examining file names in the hope that the file names reveal the contents of files. A specification library seeks to ameliorate this problem by providing operations for creating and maintaining libraries of specifications, and operations for conveniently browsing through libraries.

We envisage a scenario where users will want to create a library of specifications for a variety of reasons. For example, a user might want to create a library of specifications that he frequently finds useful while developing new specifications. A group of users working on a large specification project might wish to maintain a library of all the specifications in the project.

1.3 An Overview of the Structure of our Library

In this section we will informally discuss the requirements imposed on the structure of our library by (i) the basic units in it, and (ii) the mechanism for organizing the basic units in the library.
The basic unit in our library is a specification. A specification has a fixed number of components, and these components have fixed interpretations based on the semantics of the specification language. A relevant thing that we need know about a specification at this point is that specifications in a library can have pointers to other specifications in the library. The specifications in a library form a directed (possibly cyclic) graph.

In addition to the specifications, our library has a mechanism for organizing the specifications in it. This mechanism, called a Classification Scheme is simply a tree of classes. (Intuitively, a CS is similar to the CS of a conventional book library such as the Library of Congress CS.) A specification in a library can be classified under more than one class of the CS. We will discuss a CS in more detail in chapter 3.

1.4 Two Issues: Accessing Specifications and Browsing Operations

In this section, we will discuss two issues that will be our main concerns in the design of our library. The first issue deals with providing indices for accessing the specifications in a library, and the second issue deals with operations for maintaining a library and browsing through them. In fact these two issues are relevant to all libraries in general, whether they are of programs or books or specifications. We shall explain the first issue by using examples from a library of books.

The first issue is providing multiple indices for accessing a book in a library. Each book in a library is uniquely identified by a number that is called its book number. A book number is actually composed of two parts: one generated from the Library of Congress Classification scheme based on the subject of the book, and another generated using the Cutter Classification Scheme based on other attributes of a book such as its author, its year of publication, etc. Before a user can locate a book in a library, he needs to know its book number. Libraries contain catalogs that provide a mapping from the name of an author or the title of a book to its book number. If a
user knows the exact title (or the author) of a book, locating that book in a library is straightforward. Using the title catalog (or the author catalog,) the user can determine the book number. Using this book number the user can go to the appropriate shelf in the library and locate the book. A more complex situation occurs when a user goes to a library looking for a book on a particular topic; he is interested in examining the books available on the topic, and then deciding whether any of the books are useful to him. From experience we know such situations are quite common. To help a user in such situations, libraries have a catalog of books by subject. These catalogs also have information about closely related subjects so that a particular book could be found under more than one subject.

While browsing through a library of books, a user has to frequently "oscillate" between the catalogs and the shelves. We will exploit the machine support for our specification library to minimize this "distance" between the catalogs and the shelves. Actually, we will completely eliminate this "distance" by including specifications themselves (and not some information about the specifications) in our catalogs. Also, by allowing a user to create his own personal catalogs, catalogs serve as repositories for temporarily storing a set of specifications through which a user can browse and select (or eliminate) certain specifications to create new temporary catalogs. We will discuss catalogs and their usefulness in browsing though a library in chapter 4.

The second issue deals with identifying a set of operations for maintaining libraries and browsing through them. By maintenance operations, we allude to operations that change the contents of a library. These include operations such as installing specifications in a library, expunging specifications from a library, replacing the text of a specification, classifying the specifications, adding new classes to the CS of the library, etc. Browsing operations are useful for examining the contents of a library and, in contrast to the operations for maintaining a library, these operations do not affect the contents of a library. These include operations such as
compiling a list of all the specifications classified under a particular class, compiling a list of all the specifications in a library that satisfy some proposition on the components of a specification, scanning through the components of a list of specification, and selecting specifications that need to be examined more thoroughly, etc.

1.5 Goals of the Thesis

The goal of this thesis is to present the design of a specification library. We will use formal specification as a tool for presenting our design.

Since we will present formal specifications for a library of specifications, we would like to warn the readers about two possible ways in which we will use the word specification: as a description and an object. When we say a specification of a library, we use specification as a description. When we say a specification in a library, we use specification as an object. In most cases the meaning will be clear from the context. When there is a possibility for confusion we shall explain the meaning we intend.

In chapter 2 we present our specification language. In chapter 3 we present the design of our library, and operations for maintaining its components. In chapter 4 we characterize a set of operations for browsing through a library. In chapter 6 we include our conclusions and future work.

1.6 Related Work

A lot of the ideas in the design of our library have been used in other systems, particularly in data bases. Our contribution lies in the adaptation and integration of these ideas in designing a library of a particular kind of specifications, and providing a formal specification of the design. The ideas fall into two general categories: one dealing with the structural organization of our
library, and the other dealing with operations for browsing through our library.

A number of program libraries, which are simply a collection of useful programs residing on a file system, exist [22]. Operations for accessing programs in such libraries are the same as the operations for accessing files in a file system. The MDL programming environment [18] contains a library system which maintains mappings from function names to functions. When a user compiles a MDL program with function names in them, the compiler automatically load functions in the compilation environment by using the mapping in the library system from a function name and to the function itself.

The CLU library provides a file system with a hierarchical name space for maintaining information about abstractions [20]. The leaf nodes of the library are called description units (DUs), one for each abstraction. A DU essentially contains information about its abstraction such as the modules (zero or more) that implement the abstraction, and the interface information needed to type check the uses of the abstraction. The information in the DU permits separate compilation of single modules, with complete type checking at compile-time of all external references in the module.

Programming libraries, the CLU library, and the MDL library, permit a user to refer to modules of a software system by "convenient" names if the user already knows the names; but do not support well the examination of modules whose name the programmer does not know. Our specification library imposes a structure on the specifications in the library that is analogous to the organization imposed on a library of books by the Library of Congress Classification Scheme. Such a scheme is useful for locating specifications in situations when a user does not know the name of a specification, but rather has an idea about the contents of the specifications.
The Smalltalk programming environment takes advantage of the simple, hierarchical system model provided by the Smalltalk programming language [10], and allows users to classify the basic units in the language (i.e. classes) into what they call categories and subcategories. Our approach is similar and a little more extensive than theirs in that our classification structure allows specifications to be classified under more than single class (unlike the hierarchical model of the SmallTalk library.)

We developed our intuition for浏览 through a specification library based on our experience with browsing through libraries of books. In order to relate our ideas to similar ideas in existing systems, we provide a summary description of our ideas wherever needed.

The idea behind our catalogs is very similar to idea of tables in relational data bases. As mentioned earlier, a catalog is essentially a repository for temporarily storing a set of specifications when a user is browsing through a library. In relational data bases [4] users can create temporary tables for storing results when they are browsing through a data base.

Two of the operations for creating new tables from existing tables is by taking the union and difference of two tables. We provide similar operations for creating new catalogs.

In our library the user can create a new catalog from a library or an existing catalog such that all the specifications in the new catalog satisfy some boolean propositions on the components of a specification. Data bases SYSTEM/2000 [25] and SYSTEM R [19] allow a user to access records in the data base by specifying boolean proposition on the record fields.

A user of our library can selectively display only some of the components of the specifications in a catalog. This facility is similar to the projection operation in a relational data base, which creates a new table from an existing table by projecting only some of the attributes of
the table.

One of the important ways a user of our library can create a new catalogs from an existing catalog is by scanning through the specifications in a catalog, and optionally marking some of the specifications as being selected. After scanning through the catalog, the user can create a new catalog by either including or excluding all the selected specifications. We not aware of any database user interface that provides a similar facility.
2. Specification Language

In this chapter we present our specification language. In section 2.1 we give an idea about the two kinds of components of a system for which we will write specifications: routines and objects. In sections 2.2, 2.3 and 2.4 we present our language for specifying the properties of an object. In section 2.5 we present certain notational conveniences that increase the readability of the specifications. In section 2.6 we present the primitives in our language for specifying a routine.

2.1 Specification of a System

In [Guttag80] the authors present a specification technique for developing the specification of a system. In their technique

One can view a system as consisting of a state and a set of mechanisms (which we shall call routines) for changing and extracting information from that state. Routines are regarded as actions of "the external world," so that the current state is always the result of some previously invoked routine. Any state can, however, be discussed or even fully understood without any reference to these routines. That is to say, one can deal with the information contained in a state without any reference to these routines.

In developing the specification for our library we adopt the above technique. We identify two kinds of components in our library: the set of routines available to a user, and the set of objects that are part of the state of the system. For example, the specification of a library will include routines such as compiling a list of all the specifications classified under a particular class, selecting specifications that need to be examined more thoroughly, installing specifications in a library, classifying the specifications, etc., and objects such as library, catalogs, etc.
Our specification language has an entity called a *routine* for describing the behavior of routines available to a user, and an entity called a *trait* for describing the properties of objects. (As we shall see later, traits are more general in that they can be used to can be used describe things more than just properties of objects.) We have borrowed the notion of routines and traits from the Bicycle Specification Language.¹ Justifying the usefulness of this language, or giving semantics for it, is outside the scope of this thesis. Our objective is merely to present our specification language in sufficient detail so that a reader can understand the various routines and traits in the specification of our library.

---

1. Bicycle is a specification language being developed by the SPD group at MIT [17]. Bicycle supports traits and interfaces. Our traits are similar to traits in Bicycle, and our routines are a special and a very simple kind of a Bicycle interface. The definitions of traits and interfaces in Bicycle are strictly more powerful than definitions of traits and routines in our language.
2.2 Trait Specification

One of the fundamental units in our language is a trait. Often a trait describes an abstract data type;\(^1\) however, traits may describe other things as well. Traits are simply viewed as a collection of signatures and properties. Figure 1 contains the concrete syntax of a simple trait, and Figure 2 gives a trait that captures the properties of booleans. An informal explanation of some of the components of trait Boolean is given below.

1. Boolean is simply the traitId of the trait. We will often use the same name for a traitId and the principal sortId of a trait. We can distinguish between them by the context in which they appear.

2. The \(<\text{opPart}\>\) is a set of operator definitions that provide the syntactic and sort checking information for the operators associated with trait Boolean.

3. For each \(<\text{op}\>\), we refer to its \(<\text{fcnId}\>\) part as the identifier part, and to the \(<\text{sig}\>\) part (containing the domain and the range of an operator) as the signature part.

4. The \(<\text{propPart}\>\) is a set of axioms that can be used for simplifying expressions involving operators. The set of axioms in a \(<\text{prop}\>\) is supposed to "constrain" only the operators listed in the \(<\text{opList}\>\). A sortId in the \(<\text{opList}\>\) of a constrains clause will stand for the list all fcnId's in whose signature the sortId appears. The axioms are universally quantified over the set of all the variables that appear in the axioms.

\(^1\) The specification of a trait is similar to what has been described in the literature as the "algebraic" approach to specification of a data type in [13, 28], among others. In this approach one first presents a set of operator names and associated domains and ranges. These names can be viewed as names that can be used to refer to values described by the trait, and a set of equations designed to answer questions that one can pose about these values [14]. A good introduction to reading and writing algebraic specifications is contained in [13].
Fig. 1. Concrete Syntax for a Trait

\[ \text{trait} \quad ::= \quad \text{trait} \ \langle \text{traitId} \rangle \ \langle \text{opPart} \rangle \ \langle \text{propPart} \rangle \]
\[ \text{opPart} \quad ::= \quad \text{there exists functions} \ \langle \text{op} \rangle^* \]
\[ \text{op} \quad ::= \quad \langle \text{fcnId} \rangle : \langle \text{sig} \rangle \]
\[ \text{sig} \quad ::= \quad \langle \text{sortId} \rangle^* \ \langle \text{sortId} \rangle \]
\[ \text{propPart} \quad ::= \quad \langle \text{prop} \rangle^* \]
\[ \text{prop} \quad ::= \quad \text{constrains} \ \langle \text{opList} \rangle \ 	ext{so that} \ \langle \text{axioms} \rangle^* \]
\[ \text{opList} \quad ::= \quad \langle \text{fcnId} \rangle^* \]
\[ \text{axiom} \quad ::= \quad \langle \text{expr} \rangle = = \langle \text{expr} \rangle \]

Fig. 2. Trait Boolean

trait Boolean

there exists functions

\[
\text{# true:} \quad \rightarrow \text{Boolean} \\
\text{# false:} \quad \rightarrow \text{Boolean} \\
\text{# and: Boolean X Boolean} \quad \rightarrow \text{Boolean} \\
\text{# or: Boolean X Boolean} \quad \rightarrow \text{Boolean} \\
\text{# imp: Boolean X Boolean} \quad \rightarrow \text{Boolean} \\
\text{# not: Boolean} \quad \rightarrow \text{Boolean}
\]

constrains \[
\text{[ # true, # false, # and, # or, # imp, # not] so that}
\]
\[
\text{# and ( # true, # true) = = # true} \\
\text{# and ( # false, # true) = = # false} \\
\text{# and ( # true, # false) = = # false} \\
\text{# and ( # false, # false) = = # false} \\
\text{# or ( # true, # true) = = # true} \\
\text{# or ( # false, # true) = = # true} \\
\text{# or ( # true, # false) = = # true} \\
\text{# or ( # false, # false) = = # false} \\
\text{# imp (b1, b2) = = (# not (b1)) # or b2} \\
\text{# not ( # true) = = # false} \\
\text{# not ( # false) = = # true}
\]

end Boolean
2.3 Including Traits

A new trait can be built using traits that have already been defined. We do this through "trait inclusion." Before we give an example of building a trait by including other traits, we add two new productions given below to the concrete syntax of a trait.

\[
\langle \text{trait} \rangle \quad ::= \quad \text{trait} \langle \text{traitId} \rangle \langle \text{includePart} \rangle \langle \text{opPart} \rangle \langle \text{propPart} \rangle \\
\langle \text{includePart} \rangle \quad ::= \quad \text{include} \langle \text{traitId} \rangle \ast
\]

The effect of including trait A in trait B is equivalent to appending the \langle opPart \rangle and \langle propPart \rangle of A to the \langle opPart \rangle and \langle propPart \rangle of B. Consider trait SimpleSet given in Figure 3. Figure 4 shows the result of including trait SimpleSet in another trait, SimpleSet1. Though trait SimpleSet1 includes three traits, we show the effect of expanding only one of the included traits.

---

Fig. 3. Trait SimpleSet

trait SimpleSet
include Boolean

there exist functions

\[
\begin{align*}
\# \text{empty}: & \quad \rightarrow \quad \text{Set} \\
\# \text{insert}: & \quad \text{Set} \times \text{Elem} \quad \rightarrow \quad \text{Set} \\
\# \text{delete}: & \quad \text{Set} \times \text{Elem} \quad \rightarrow \quad \text{Set} \\
\# \text{has}: & \quad \text{Set} \times \text{Elem} \quad \rightarrow \quad \text{Boolean}
\end{align*}
\]

constrains [Set] so that

\[
\begin{align*}
\# \text{delete} (\# \text{empty}, e) &= \# \text{empty} \\
\# \text{delete} (\# \text{insert}(s, e1), e2) &= \text{if } e1 = e2 \\
&\quad \text{then } \# \text{delete}(s, e2) \\
&\quad \text{else } \# \text{insert}(\# \text{delete}(s, e2), e1)
\end{align*}
\]

\[
\begin{align*}
\# \text{has} (\# \text{empty}, e) &= \# \text{false} \\
\# \text{has} (\# \text{insert}(s, e1), e2) &= \text{if } e1 = e2 \\
&\quad \text{then } \# \text{true} \\
&\quad \text{else } \# \text{has}(s, e2)
\end{align*}
\]

end SimpleSet
Fig. 4. An Example of Expanding an Included Trait

trait SimpleSet1

    include SimpleSet, Integer, Boolean

    there exists functions

    # size: Set --> Integer

    constrains [Set] so that

    # size ( # empty) = = 0
    # size ( # insert (s, e)) = = if # has (s, e)
      then # size (s)
      else # size (s) + 1

end SimpleSet1

partially expands to

trait SimpleSet1

    include Integer, Boolean

    there exists functions

    # empty: -- Set
    # insert: Set X Elem --> Set
    # delete: Set X Elem --> Set
    # has: Set X Elem --> Boolean
    # size: Set --> Integer

    constrains [Set] so that

    # delete ( # empty, e) = = # empty
    # delete ( # insert (s, e1), e2) = = if e1 = e2
      then # delete (s, e2)
      else # insert ( # delete (s, e2), e1)

# has ( # empty, e) = = # false
# has ( # insert (s, e1), e2) = = if e1 = e2
      then # true
      else # has (s, e2)

# size ( # empty) = = 0
# size ( # insert (s, e)) = = if # has (s, e)
      then # size (s)
      else # size (s) + 1

end SimpleSet1
2.3.1 Recursive Trait Inclusion

A trait may include itself directly or indirectly. Given the "text inclusion" semantics of trait inclusion, when expanding a trait T, any traitId that has been already been expanded is not expanded again. Such a rule prevents an "infinite" expansion when a trait including itself is expanded. (A semantic justification for this is contained in [17].)

2.3.2 Operator Name

Operator names include their signatures as well as the identifier parts, i.e.

\[ \langle \text{fcnId}\rangle; \langle \text{sortId}\rangle^* \rightarrow \langle \text{sortId}\rangle \]

However, when we refer to an operator we do not write the signature part, but use only the identifier part. This allows an apparent overloading of operator names which will prove useful for infix operators.

While developing the specification of our library, we will frequently build different "set" traits by renaming the fcnId's and sortId's of trait SimpleSet. For all such traits, we will use the overloaded infix operator \( \in \) to denote their respective set membership operator. In fact, we will use overloaded operators for a few other operators such as the string inequality operator, integer inequality operators, etc. Appendix IV contains a list of overloaded operators that we use in our specifications.

The specifications are intended to be read and developed using an editor, and the editor will help in unambiguously inferring the signature (if possible) from the context in which the the operator appears. For the convenience of our readers, in appendices III we include a mapping from operator names to the traits in which the operators "first" appear. A reader will find this mapping particularly useful while reading routines.
2.4 More Complex Trait Inclusion

One can also include a trait by (i) renaming fcnId's and sortld's in it, and (ii) discarding some of operators from it. Below we add new productions to the concrete syntax for incorporating this more complex trait inclusion.

\[
\text{<includePart> ::= include <includeList>^*}
\]
\[
\text{<includeList> ::= <traitId> <bindingList> <withoutList>}
\]
\[
\text{<bindingList> ::= [ <binding[^*] > ]}
\]
\[
\text{<withoutList> ::= without <fcnId[^*] >}
\]
\[
\text{<binding> ::= <fcnId --> <fcnId | <sortld --> <sortld>}
\]

The interpretation of the \(<\text{withoutList}>\) and \(<\text{bindingList}>\) in an \(<\text{includeList}>\) is as follows:

1. Before including the designated trait, remove any \(<\text{op}>\) or \(<\text{axiom}>\) part in which any fcnId in the \(<\text{withoutList}>\) occurs.

2. A \(<\text{bindingList}>\) is a list of "actual --> formal" identifier pairs. Before including the designated trait, for each occurrence of the formal identifier in the \(<\text{opPart}>\) and \(<\text{propPart}>\) of the designated trait substitute the corresponding actual identifier. All the fcnId's and the sortld's in the text of a trait are potential formal parameters of the trait.

In Figure 5 we give an example of using a complicated include statement to build trait Bag from trait SimpleSet. In this case, we include the text of trait Set, but we first rename the sortld Set to Bag. Furthermore, we specify that the operators #delete and #size (and the axioms containing them) not be included. In addition, we add two new operators, #delete and #occurrences, and the relevant axioms to trait Bag.
Fig. 5. A Complicated Trait Inclusion

trait Bag

include Integer, Boolean, SimpleSet [Bag → Set] without # delete, # size

there exists functions

# delete: Bag X Elem → Bag
# occurrences: Bag X Elem → Integer

constrains [Bag] so that

# delete ( # empty, e) = = # empty
# delete ( # insert (s, e1), e2) = = if e1 = e2
then s
else # insert ( # delete (s, e2), e1)

# occurrences ( # empty, e) = = 0
# occurrences ( # insert (s, e1), e2) = = if e1 = e2
then # occurrences (s, e2) + 1
else # occurrences (s, e2)

end Bag

partially expands to

trait Bag

include Integer, Boolean

there exists functions

# empty: Bag X Elem → Bag
# insert: Bag X Elem → Bag
# has: Bag X Elem → Boolean
# delete: Bag X Elem → Bag
# occurrences: Bag X Elem → Integer

constrains [Bag] so that

# has ( # empty, e) = = # false
# has ( # insert (s, e1), e2) = = if e1 = e2
then # true
else # has (s, e2)

# delete ( # empty, e) = = # empty
# delete ( # insert (s, e1), e2) = = if e1 = e2
then s
else # insert ( # delete (s, e2), e1)

# occurrences ( # empty, e) = = 0
# occurrences ( # insert (s, e1), e2) = = if e1 = e2
then # occurrences (s, e2) + 1
else # occurrences (s, e2)

end Bag
2.5 Syntactic Amenities

In the process of developing the specifications for our library, we found ourselves inventing and using a number of notational shorthands. We found these a great help. In this section we describe some of the notational shorthands that we will use for writing our specifications.

As we shall see later, operators of trait Boolean are used in the axioms of most non-trivial traits. Rather than explicitly include trait Boolean, we assume trait Boolean to be a pervasive trait. That is, operators of trait Boolean can be used in a trait without explicitly including trait Boolean in its <includePart>.

We also adopt a more familiar notation for denoting the operators of trait Boolean. Rather than use #true, #false, #and, #or, #imp, and #not, we use the more familiar and shorter notations of true, false, \(\wedge\), \(\vee\), \(\Rightarrow\), and \(\neg\), respectively. Also, we use an infix notation for the operators \(\wedge\), \(\vee\), and \(\Rightarrow\), and a prefix notation for the operator \(\neg\).

We also adopt convenient notation for denoting the set membership operators. Instead of writing #has (s, e), we use a more familiar notation \(e \in s\). Also, instead of writing \(\neg (e \in s)\), we simply write \(\not\in s\).

Appendix I contains a specification of trait Integer. Using the operators trait Integer, we can denote any integer as a composition of the #succ, and #pred operators. For example, \(2\) is denoted by the term #succ (#succ (0)), and \(1\) is denoted by the term #pred (0), and so on. Rather than denote an integer by a composition of #succ and #pred operators, we will simply use the symbol representing integers. Further, we will use the familiar infix notation for the

1. In Bicycle one can define an operator to be written in an infix notation.
various operators of trait Integer.

One syntactic convenience that we frequently found useful while writing our specification was a record notation. It adds no power to the specification language, but simply helps to shorten the text of a specification. Our notation is the following:

The expression

\[ S = \text{record} \{ f_1 : S_1, \ldots, f_i : S_i, \ldots, f_n : S_n \} \]

in the text of a trait is equivalent to adding the \langle op\rangle s

\# createS : S_1 \times \ldots \times S_i \times \ldots \times S_n \rightarrow S
\# getf_1 : S \rightarrow S_1
\ldots
\# getf_i : S \rightarrow S_i
\ldots
\# getf_n : S \rightarrow S_n

to the \langle opPart\rangle of the trait, and the \langle prop\rangle constrains \{S\} so that

\# getf_1 \ (\ # createS \ (s_1, \ldots, s_i, \ldots, s_n)) = = s_1
\ldots
\# getf_i \ (\ # createS \ (s_1, \ldots, s_i, \ldots, s_n)) = = s_i
\ldots
\# getf_n \ (\ # createS \ (s_1, \ldots, s_i, \ldots, s_n)) = = s_n

to the \langle propPart\rangle of the trait. For example, the specification

trait Tree
include Node, Forest

Tree = \text{record} \{ \text{Root: Node, Forest: Forest} \}

end Tree

is equivalent to the specification

trait Tree
include Node, Forest

there exist functions
# createTree: Node X Forest --> Tree
# getRoot: Tree --> Node
# getForest: Tree --> Forest

constrains [Tree] so that

# getRoot (# createTree (node, forest)) = node
# getForest (# createTree (node, forest)) = forest

end Tree
2.6 Routine Specification

As mentioned earlier, a routine in the specification of an interactive system corresponds to a routine available to a user. A routine specification is concerned with the manipulation of values of objects. When invoked with a sequence of objects and values as its argument, a routine can create new objects and/or modify the state of the argument objects.

Often, before a routine can be invoked, there are conditions that the arguments of the routine should satisfy. For example, a routine which deletes an element from a set might require that the element should be present in the set. Routines have a precondition clause that describe such conditions. The precondition that must be satisfied at the beginning of a routine call is specified as a predicate on values of objects.

The effect of a routine is described by a postcondition that must be satisfied at the end the call to the routine. The postcondition, which must be satisfied at the end of a routine call, is specified in two ways: as a predicate on values of objects, and as a signal statement which describes some error message.

One of the clauses in a routine lists all the argument objects whose value can be changed by a call to the routine. We call this the changes clause of a routine.

The pre(changes)/post clauses can be viewed describing the observable "effects" of calling a routine. Routines can have more than one set of pre(changes)/post clauses. At the beginning of a routine call, if more than one precondition is satisfied, then at the end of that routine call all the corresponding postconditions (of the preconditions that are satisfied at the beginning of the routine call) must be satisfied.
Like traits, routines have an include clause that list the traits included in the routine. The
predicates in the pre and postconditions of a routine can only use the operators of the included
traits.

2.6.1 Useful Terminology

The basic unit of information is a value. All values have an associated sort. A trait describes
the relationship between values. In the trait language (i.e. the language in which the trait is
described,) a value is denoted by an expression involving operator names of the traits.

The basic containers for information are objects. All objects have a value and a type. The
value is the information content of an object, and the type determines the "behavior" of an object.
There is a trivial mapping from the type of an object to the sort of the object's value. An object of
type T has a value of sort T.

2.6.2 Concrete Syntax of a Routine

\[
\begin{align*}
\text{<routine>} &::= \text{routineId} \text{<input>} \text{<return>} \text{<includePart>} \text{<effect>}^* \\
\text{<input>} &::= \text{Input} \text{<param>}^* \\
\text{<return>} &::= \text{Returns} \text{<param>} \\
\text{<param>} &::= \text{obj} \text{<objId>} \text{<typId>} | \text{<valId>} \text{<sortId>} \\
\text{<effect>} &::= \text{<pre>} \text{<changes>} \text{<post>} \\
\text{<pre>} &::= \text{Pre} \text{<assertion>} \\
\text{<changes>} &::= \text{<objId>}^* \\
\text{<post>} &::= \text{Post} \text{<assertion>} | \text{Post signal} \text{<string>} \\
\text{<assertion>} &::= \text{true} \\
\text{<assertion>} &::= \text{false} \\
\text{<assertion>} &::= \text{<expr>} = = \text{<expr>} \\
\text{<assertion>} &::= \forall \text{<varDecl>} \text{<assertion>} \\
\text{<assertion>} &::= \exists \text{<varDecl>} \text{<assertion>} \\
\text{<assertion>} &::= \text{<assertion>} \land \text{<assertion>} \\
\text{<assertion>} &::= \text{<assertion>} \lor \text{<assertion>} \\
\text{<assertion>} &::= \text{<assertion>} \Rightarrow \text{<assertion>} \\
\text{<assertion>} &::= \text{<assertion>} \Leftrightarrow \text{<assertion>} \\
\text{<assertion>} &::= \neg \text{<assertion>} \\
\text{<varDecl>} &::= \text{<varId>} \text{<sortId>} 
\end{align*}
\]
2.6.3 Conventions

We follow the following typographical conventions in a routine specification. RoutineId's are written in upper case alphabets. TypeId's (and SortId's as before) are written in italics font. The value of an object is denoted by following the object name with an up arrow \( (a^\uparrow, b1^\uparrow, b2^\uparrow) \).

For an object whose value can be changed by a call to a routine, in the postcondition of the routine, sometimes we need to distinguish between the initial and final values of the object. For such objects, the value that the object had at the beginning of a routine call is denoted by suffixing the object name with a "\( . \)". For example, suppose a call to routine \( R \) changes the value of object \( O \). In the postcondition, \( O^\uparrow \) refers to the value of \( O \) at the end of a call to routine \( R \), and \( O'^\uparrow \) refers to the value of \( O \) at the beginning of the call to the routine. (This problem does not exist in a precondition because a precondition is always defined on the values of objects at the beginning of the routine call.)

As with traits, trait Boolean is assumed to be a pervasive trait whose operators are available in all routines. Also, we shall use the infix notation for Boolean operators, Integer operators, and the set membership operator.

2.6.4 Examples

Before we present examples of routines, we build a trait that describes a set of integers.

\[
\text{trait IntegerSet} \\
\text{include SimpleSet [IntSet \rightarrow Set, Integer \rightarrow Elem], Integer} \\
\text{end IntegerSet}
\]
Below we give the specification of a routine for taking the union of two sets. The routine takes two objects of type \textit{IntSet} as its input arguments, and returns an object of type \textit{IntSet}. The Input clause gives the objld's and typeld's of the input arguments, and the Returns clause gives the objld and typeld of the return argument. The Pre clause simply says that there are no preconditions that the objects must satisfy.\textsuperscript{1} The Changes clause says that value of no input argument is changed by a call to the routine. The postcondition says that the value of "c3" at the end of the routine call is the same as given by the expression on the right hand side of the " = = ." Note that " = = " is the same equality operator that we used in the equations of the axioms.

\begin{verbatim}
Routine UNION
Input  obj s1: IntSet, obj s2: IntSet
Returns obj s3: IntSet
Includes IntegerSet
Pre  true
Changes nothing
Post  \forall i: Integer, 

    [(i \in s3) = (i \in s1 \lor v \in s2)]
\end{verbatim}

Below we give an example of a routine for removing an element from a set. Note that this routine has two arguments: an object of type \textit{IntSet}, and a value of sort \textit{Integer}. Though in the below example the predicates in the two pre clauses are mutually exclusive, in general the predicates in the pre clauses of a routine need not be mutually exclusive.

\begin{verbatim}
1. In our specification language, we assume that an input object is always initialized before it is passed as an argument to a routine call, i.e. it has a value.
\end{verbatim}
Routine REMOVE
Input obj : IntSet, i : Integer
Returns nothing
Includes IntegerSet

Pre \( i \in s \uparrow \)
Changes nothing
Post signal ("THE INTEGER TO BE REMOVED IS NOT PRESENT IN THE SET")

Pre \( i \in s \uparrow \)
Changes s
Post \( s \uparrow = = \# \text{delete}(s \uparrow, i) \)
3. Specification Library

In this chapter we present the organization and the components of our library, and the operations for maintaining these components. In section 3.1 we describe the component of our library that provides a mechanism for organizing the specifications in our library: the classification scheme. In section 3.2 we discuss the issue of binding of names in the Includes clause of a specification. (Recollect from chapter 2 that routines and traits have an Includes clause which contains names of traits.) In section 3.3 we discuss the issues related to the creation and installation of the basic component of our library, a specification. In section 3.4 we give an informal overview of the major abstractions in the specification of our library. In section 3.5 we give traits that describe four basic data structures. We will frequently use these traits in section 3.6 where we present traits corresponding to the various abstractions in our library. Section 3.7 contains routines that describe the effect of operations for changing the state of a library.

We developed the intuition behind the design of our library from conventional libraries of books. More specifically, we borrowed the well known idea of a classification scheme from them.

3.1 Classification Scheme

A classification scheme (CS), which is simply a tree of classes, is a mechanism for organizing the specifications in a library. The strategy followed in organizing the classes in a CS is similar to the strategy followed in a conventional CS such as the Library of Congress Classification Scheme. Figure 6 contains a possible CS for classifying the specifications of data structures. The root class, DATA STRUCTURES, refers to a general class of specifications. This general class is divided into two smaller classes: LINEAR (referring to the class of linear data structures) and NON-LINEAR (referring to the class of non-linear data structures.) These classes are further divided into smaller classes until a class refers to a small class of specifications.
Fig. 6. A Classification Scheme for Data Structures

Data Structures

Linear
Lists Stacks Queues Trees Graphs

Non Linear

Figure 7 shows a possible CS for classifying specifications of mathematical entities. A particular class in a CS is denoted by the sequence of classes from the root class to it.

Fig. 7. A Classification Scheme for Mathematical Entities

Mathematical Entities

Functions
Predicates Sets Sequences Relations Groups Rings

Structures
Let us briefly compare a CS with the structuring mechanisms of a file system. The directories in a file system, which are essentially a mapping from file names to tiles, help in organizing the files in the file system. In most file systems, the file name of a given file can exist in exactly one directory. (An exception is the UNIX file system [23] in which a file name can exist in more than one directory of the file system.) The CS of our library provides a structuring mechanism similar to that of UNIX. A specification in our library can be classified under more than one class of the CS. We justify the usefulness of this capability to classify a specification under more than class by pointing to conventional book libraries where it is possible to find a book classified under more than one subject. Also note that this capability allows a given set of specifications in a library to be organized in orthogonal ways.

How does a CS help a user in browsing through a library in a systematic and efficient fashion?

1. By examining the classes in the CS of a library, a user can quickly get an idea about the different specifications present in a library. In certain cases this information might be sufficient for a user to realize that a library does not contain the relevant specification. When searching for a specification in the library, the user can selectively browse through only certain classes under which a specification of interest is likely to be classified.

2. A CS can be invaluable as a road map for browsing through the specifications in a library. As a user browses through the specifications classified under various classes of the CS, the library keeps track of the classes that he "visits," and provides useful information to a user for systematically navigating through the classes in a CS. A user can begin browsing through classes under which he believes that the specification of interest is likely to be classified, and if necessary, proceed to browse through classes under which he believes that the specification interest is less likely to be classified. At any stage of browsing the user can ask the library for a list of classes that he has "visited," and the list of classes that he has not "visited" so far.

We will discuss the operations for accessing specifications and navigating through a CS in more detail in chapter 4 when we discuss the operations for browsing through a library.
A major drawback of a CS is that as new specifications are added to a library, the CS requires "maintenance." Certain classes might no longer contain a small number of specifications; in such cases it is desirable to partition these class into smaller classes. There might be occasions when a specification being added to a library cannot be classified under any of the existing classes. In such cases, the user can extend an existing class by adding a new sub class, and classify the specification under this new class. Addition of new classes might skew the tree structure of a CS; in such cases it is desirable to balance the tree structure of a CS. Operations for extending and deleting classes of a CS will be discussed in section 3.7.

3.2 Bindings the Import Names of a Specification

Remember routines and traits have an include clause containing names of traits. Let us call these names the import names of a specification. After a specification has been installed in a library, it’s import names can be bound to other trait specifications in the library.

The requirements for binding import names of a specification are quite flexible. Firstly, an import name of a specification A can be bound to a specification B only if both A and B are in the same library. The reason for disallowing import names to be bound to specifications in other libraries is to make a library an independent unit that does not depend on other libraries. Secondly, a specification can contain import names that are not bound to any specification. The reason we allow import names to be unbound is because sometimes a user has written a specification, A, and wants to install it in a library even though the specifications to which A's import names are to be bound have not been written so far.

The binding of an import name of a specification can be changed. This is useful when for some reason a user would like to bind an (already bound) import name to a new specification. For example, a user might discover that the specification to which an import name is bound is
"incorrect," and he might want to bind the import name to a "correct" specification.

3.3 Installing Specifications in a Library

What is installed in the library is either a routine specification or a trait specification. The specification being installed consists of two components: the text of the specification, and the set of import names of the specification. The library needs the information about the set of import names to perform their binding. (When we discuss the browsing operations in chapter 4, we will impose more structure on the text of a specification. For the purposes of this chapter, we assume the text of a specification to be simply a string of characters.) The library also needs to know whether a specification is a trait specification or a routine specification. The library needs this information to ensure that an import name is bound only to a trait specification (and not to a routine specification.)

For each specification installed in a library, the library maintains information about the author of the specification, and the time when the specification was installed in the library. A timestamp is generated automatically by the library, and is different for each specification. Thus a specification is uniquely identified in a library by its timestamp.

In addition to author and timestamp, there are two more attributes associated with each specification in the library: a set of classes under which the specification has been classified, and a table containing information about the specifications to which the specifications import names have been bound.
3.4 Major Abstractions in the Specification of our Library

In this section we give a brief overview of the major abstractions in the specification of our library. We make no attempt here to describe the abstractions in significant detail. We leave a careful description of the abstractions to their formal specifications in the next two sections, and content ourselves here with presenting a few preliminary comments on the abstractions.

1. A Library is a structured collection of specifications. In addition to specifications, a Library contains a CS.

2. A CS can be viewed as a tree of ClassNames. A Class in a CS is denoted by the sequence of ClassNames from the root class to it.

3. InputSpec corresponds to a specification that can be installed in a Library. An InputSpec can be viewed as consisting of three abstractions: SpecificationText corresponding to the text of the specification, SetOfImportNames corresponding to the set of import names of the specification, and SpecType containing the information about whether an InputSpec is a trait specification or a routine specification.

4. LibSpec corresponds to a specification in a Library. Besides an InputSpec, a LibSpec is composed of four other abstractions: Author corresponding to the author of the specification, TimeStamp corresponding to the time stamp of the specification, SetOfClasses corresponding to the classes under which the specification is classified, and TableOfBindings corresponding to the table containing the binding of import names of the specification.

3.5 Basic Traits

We begin by giving trait specifications that describe four familiar data structures: a set, a sequence, a table, and a tree.

<table>
<thead>
<tr>
<th>trait</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>there exist functions</td>
<td></td>
</tr>
<tr>
<td># empty:</td>
<td>Set X Elem --&gt; Set</td>
</tr>
<tr>
<td># insert:</td>
<td>Set X Elem --&gt; Set</td>
</tr>
</tbody>
</table>
# delete: \( \text{Set} \times \text{Elem} \rightarrow \text{Set} \)
# has: \( \text{Set} \times \text{Elem} \rightarrow \text{Boolean} \)
# union: \( \text{Set} \times \text{Set} \rightarrow \text{Set} \)
# difference: \( \text{Set} \times \text{Set} \rightarrow \text{Set} \)

constrains \([\text{Set}]\) so that

\[
\begin{align*}
# \text{delete} (\ # \text{empty}, \ e) &== \ # \text{empty} \\
# \text{delete} (\ # \text{insert} (s, e1), e2) &== \begin{cases} \\
&\text{if } e1 = e2 \\
&\text{then } # \text{delete} (s, e2) \\
&\text{else } # \text{insert} (\ # \text{delete} (s, e2), e1) \\
\end{cases} \\
# \text{delete} (\ # \text{union} (s1, s2), e) &== \ # \text{union} (\ # \text{delete} (s1, e), \ # \text{delete} (s2, e)) \\
# \text{delete} (\ # \text{difference} (s1, s2), e) &== \ # \text{difference} (\ # \text{delete} (s1, e), s2) \\
# \text{has} (\ # \text{empty}, e) &== \text{false} \\
# \text{has} (\ # \text{insert} (s, e1), e2) &== \begin{cases} \\
&\text{if } e1 = e2 \\
&\text{then } \text{true} \\
&\text{else } \# \text{has} (s, e2) \\
\end{cases} \\
# \text{has} (\ # \text{union} (s1, s2), e)) &== \ # \text{has} (s1, e) \lor \ # \text{has} (s2, e) \\
# \text{has} (\ # \text{difference} (s1, e2), e)) &== \ # \text{has} (s1, e) \land (\neg \ # \text{has} (s2, e))
\end{align*}
\]

end \( \text{Set} \)

---

trait \text{Sequence}

include \text{Integer}

there exist functions

\[
\begin{align*}
# \text{new}: &\quad \text{Sequence} \\
# \text{append}: &\quad \text{Sequence} \times \text{Elem} \\
# \text{contains}: &\quad \text{Sequence} \times \text{Elem} \\
# \text{first}: &\quad \text{Sequence} \\
# \text{rest}: &\quad \text{Sequence} \\
# \text{size}: &\quad \text{Sequence} \\
# \text{concat}: &\quad \text{Sequence} \times \text{Sequence} \\
# \text{same}: &\quad \text{Sequence} \times \text{Sequence}
\end{align*}
\]

constrains \([\text{Sequence}]\) so that

\[
\begin{align*}
# \text{contains} (\ # \text{new}, e) &== \text{false} \\
# \text{contains} (\ # \text{append} (s, e1), e2) &== \begin{cases} \\
&\text{if } e1 = e2 \\
&\text{then true} \\
&\text{else } \# \text{contains} (s, e2) \\
\end{cases} \\

\%
# \text{first} (\ # \text{new}) \text{ is intentionally not defined } \%
# \text{first} (\ # \text{append} (s, e)) &== \begin{cases} \\
&\text{if } \# \text{size} (s) = 0 \\
&\text{then } e \\
&\text{else } \# \text{first} (s) \\
\end{cases} \\
# \text{rest} (\ # \text{new}) &== \ # \text{new} \\
# \text{rest} (\ # \text{append} (s, e)) &== \begin{cases} \\
&\text{if } \# \text{size} (s) = 0 \\
&\text{then } \# \text{new} \\
\end{cases}
\end{align*}
\]
# containsIndex (# createTable) = false
# containsIndex (# addEntry (t, l, v), l1) =
    if i = i1
      then true
    else # containsIndex (t, l1)
end Table

Trait Path simply describes a sequence of nodes. Operator # isSubPath (p1, p2) checks if
p1 is a leading subsequence of p2.

trait Path

include Sequence
[Path -> Sequence, 
 Node -> Elem, 
 # nodeEq -> # elemEq]

there exist functions

# isSubPath: Path X Path -> Boolean

constrains [Path] so that

# isSubPath (p1, p2) =
    if # size (p1) = 0
      then true
    else if # size (p2) = 0
      then false
    else (# first (p1) = # first (p2)) \n    ( # isSubPath (# rest (p1), # rest (p2))
end Path

A pictorial description of the effect of some of the operator names is given below. Let T
denote the tree shown below. A node in the tree is denoted by a letter of the alphabet (example:
a, l, k). A path is denoted by a sequence of letters separated by the period symbol (example: a,
a.b, a.c.).

```
   a
  /   
 b     c
 /     / 
 d     e  f
       /   
       g
```
The term \( \#\text{addSubTree}(T, a.b, \#\text{makeTree}(h)) \) denotes the tree obtained by adding the tree \( \#\text{makeTree}(h) \) to node "a.b" in the tree T as shown below. Note that \( \#\text{addSubTree} \) can be used to create terms that do not denote valid a tree structure. For example, if \( \neg \#\text{containsPath}(T, P) \), then the term \( \#\text{addSubTree}(T, P, T') \) does not denote a valid tree structure. In the various routines, before we use the term \( \#\text{addSubTree}(T, P, T') \) to add a sub tree to an existing tree, we make sure that the condition \( \#\text{containsPath}(T, P) \) is true.

![Tree Diagram](image)

The term \( \#\text{deleteSubTree}(T, a.b) \) denotes the tree obtained by deleting the subtree whose root node is denoted by the path "a.b" as shown below. We do not allow the root node of a tree to be deleted because we have not introduced any operator for denoting a tree with no nodes in it. Also, axioms for \( \#\text{deleteSubTree}(T, P) \) delete the sub tree only if the path, P, exists in the tree, T. In the various routines, before we use the term \( \#\text{deleteSubTree}(T, P) \) to delete a sub tree from an existing tree, we make sure that the condition \( (\#\text{contains}(T, P) \land \#\text{size}(P) > 1) \) is true.

![Tree Diagram](image)

The term \( \#\text{getSubTree}(T, a.b) \) denotes the subtree of T whose root node is denoted by the path "a.b" as shown below. Axioms for \( \#\text{getSubTree}(T, P) \) return the right sub tree only if the path, P, exists in the tree, T, i.e. \( \#\text{contains}(T, P) \) is true.

![Tree Diagram](image)
trait Tree

include Path, Node

there exist functions

# makeTree: Node -> Tree
# addSubTree: Tree X Path X Tree -> Tree
# deleteSubTree: Tree X Path -> Tree
# getSubTree: Tree X Path -> Tree
# containsPath: Tree X Path -> Boolean
# getRoot: Tree -> Node

constrains [Tree] so that

# deleteSubTree (# makeTree (n), p) = = # makeTree (n)
# deleteSubTree (# addSubTree (t1, p1, t2), p2) = =
  if # size (p2) \leq 1
    then # addSubTree (t1, p1, t2)
  else if # isSubPath (p2, p1)
    then # deleteSubTree (t1, p2)
    else # addSubTree (#
      # deleteSubTree (t1, p2), p1, t2)

# getSubTree (# makeTree (n), p) = =
  if # size (p) = 1 \land # first (p) = n
    then # makeTree (n)
  % else is intentionally not defined %
# getSubTree (# addSubTree (t1, p1, t2), p2) = =
  if p1 = p2
    then t2
    else # getSubTree (t1, p2)

# containsPath (# makeTree (n), p) = = (# size (p) = 1) \land (# first (p) = n)
# containsPath (# addSubTree (t1, p1, t2), p2) = =
  if # size (p2) = 0
    then false
    else if p1 = p2
      then true
      else # containsPath (t1, p2)

# getRoot (# makeTree (n)) = = n
# getRoot (# addSubTree (t1, p1, t1), p2) = = # getRoot (n)

end Tree
3.6 Constructing the Trait Library

We begin with four simple traits. Traits Author, ImportName, ClassName, and SpecificationText are simple strings of characters. Trait TimeStamp is simply as Integer. Traits Integer and String are given in appendices I and II, respectively.

```
trait Author
    include String [Author -> String]
end Author

Trait ImportName
    include String [ImportName -> String]
end ImportName

Trait ClassName
    include String [ClassName -> String]
end ClassName

Trait SpecificationText
    include String [SpecificationText -> String]
end SpecificationText

 Trait TimeStamp
    include Integer [TimeStamp -> Integer]
end TimeStamp
```

Trait Class is simply a sequence of ClassName. The operator #parent returns the parent class of a given class.

```
trait Class
    include Sequence [Class -> Sequence,
                      ClassName -> Elem]
end Class

there exist functions
```
# parent: Class --> Class

constrains [Class] so that

- # parent (# new) is intentionally not specified
- # parent (# append (s, e)) = = if # size (s) = 0
  then # append (s, e)
  else s

end Class

Traits SetOfClasses and SetOfImportNames are constructed by renaming the operators and sorts of trait Set.

trait SetOfClasses

include Set [SetOfClasses --> Set,
  Class --> Elem]

end SetOfClasses

trait SetOfImportNames

include Set [SetOfImportNames --> Set,
  ImportName --> Elem]
  without # union, # difference

end SetOfImportNames

Trait InputSpec describes the abstraction that corresponds to a specification that can be installed in a library.

trait InputSpec

include SpecificationText, SetOfImportNames

InputSpec = record [InputSpec: SpecificationText}
ImportNames:SetOfImportNames,
SpecType:Boolean]

Trait LibSpec describes the abstraction corresponding to the specification that exists in the library.

trait LibSpec

include InputSpec [ #isATrait --> #getSpecType],
Author,
TimeStamp,
SetOfClasses,
TableOfBindings

LibSpec = record [InputSpec: InputSpec,
Author: Author,
TimeStamp: TimeStamp]

there exist functions

# classify: LibSpec X SetOfClasses --> LibSpec
# bind: LibSpec X TableOfBindings --> LibSpec
# getClassification: LibSpec --> SetOfClasses
# getBindings: LibSpec --> TableOfBindings

constrains [LibSpec] so that

# getClassification ( # createLibSpec (in, au, ts)) = # emptySetOfClasses
# getClassification ( # classify (l, c)) = = c
# getClassification ( # bind (l, b)) = = # getClassification (l)

# getBindings ( # createLibSpec (in, au, ts)) = # emptyTableOfBinding
# getBindings ( # classify (l, c)) = = # getBindings (l)
# getBindings ( # bind (l, b)) = = b

end LibSpec
Trait TableOfBindings corresponds to the abstraction that maintains the information about
the binding of the import names of a specification. It is constructed by renaming the operators
and sorts of trait Table.

```
trait TableOfBindings
  include Table
    [TableOfBindings --> Table,
     ImportName --> Index,
     LibSpec --> Value,
     # emptyTableOfBinding -->#c createTable,
     # addBinding --> # addEntry,
     # deleteBinding --> # deleteEntry,
     # getLibSpec --> # getValue,
     # containsImportName --> # containsIndex]

  there exist functions
    
    # containsLibSpec: Table X LibSpec --> Boolean

  constrains [TableOfBindings] so that
    
    # containsLibSpec (# emptyTableOfBinding, s) = = false
    # containsLibSpec (# addBinding (t, i, s), s1) = =
      if s = s1
        then true
      else # containsLibSpec (t, s1)
```

end TableOfBindings

---

Trait CS describes the abstraction that corresponds to the CS of a Library.

```
trait CS
  include Tree
    [CS --> Tree,
     ClassName --> Node,
     Class --> Path,
     # makeCS --> # makeTree,
     # addSubCS --> # addSubTree,
     # deleteSubCS --> # deleteSubTree,
```
# getSubCS --> # getSubTree,
# validClass --> # containsPath,
# getRootClass --> # getRoot

end CS

Trait Library describes the abstraction that corresponds to a Library. It is defined as a set of LibSpecs with an optional CS.

trait Library

include Set

[Library --> Set,
LibSpec --> Elem,
# emptyLibrary --> # empty,
# addLibSpec --> # insert,
# removeLibSpec --> # delete,
# containsLibSpec --> # has]
without # union, # difference

LibSpec,
CS

there exist functions

# addCS: Library X CS --> Library
# deleteCS: Library --> Library
# csExists: Library --> Boolean
# getCS: Library --> CS

constrains [Library] so that

# containsLibSpec (# addCS (l, cs)) = = # containsLibSpec (l)

# csExists (# emptyLibrary) = = false
# csExists (# addLibSpec (l, s)) = = # csExists (l)
# csExists (# addCS (l, cs)) = = true

# deleteCS (# emptyLibrary) = = # emptyLibrary
# deleteCS (# addLibSpec (l, s)) = = # addLibSpec (# deleteCS (l), s)
# deleteCS (# addCS (l, cs)) = = # deleteCS (l)

% # getCS (# emptyLibrary) is intentionally not defined %
# getCS (# addLibSpec (l, s)) = = # getCS (l)
3.7 Specification of Maintenance Operations

In this section we give routine specifications for ten operations on a library. We call these operations the maintenance operations of a library because each of these operations can potentially alter the state of a library, and therefore, in some sense help in maintaining a library. The operations that are useful for examining the state of a library are called the browsing operations. The maintenance operations are

1. INSTALL: For installing a specification in a library.

2. EXPUNGE: For expunging a specification from a library.

3. COPY: For copying a specification from one library to another.

4. REPLACE: For replacing the text of a specification.

5. CLASSIFY: For classifying a specification under a set of classes in the CS of a library.

6. UNCLASSIFY: For removing a set of classes from the classification (also a set of classes) of a specification.

7. BIND: For binding the import name of a specification.

8. UNBIND: For removing the binding of an import name of a specification.

9. EXTEND: For adding a new class to a CS.

10. REMOVE: For removing a class from a CS.

The remainder of the section contains routines that describe the effect of each of the above operations.
In order to install a specification a user needs to supply the specification to be installed, and the name of the author who wrote the specification. When a specification is installed, a library wide unique timestamp is associated with it. This timestamp is greater than the timestamps of all the specifications present in the library. The timestamp associated with a specification is permanent and can never be modified.

<table>
<thead>
<tr>
<th>Routine</th>
<th>INSTALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>obj l: Library, inspec: InputSpec, au: Author</td>
</tr>
<tr>
<td>Returns</td>
<td>nothing</td>
</tr>
<tr>
<td>Includes</td>
<td>Library, LibSpec, TimeStamp</td>
</tr>
</tbody>
</table>

| Pre       | true |
| Changes   | 1    |
| Post      | 3 ts: TimeStamp, \[\forall s: LibSpec, [s \in l' \Rightarrow \# \text{getTimestamp} (s) < ts] \land l' = = \# \text{addLibSpec} (l', \# \text{createLibSpec} (inspec, au, ts))\] |
EXPUNGE

This operation is useful for expunging a specification from a library. A specification can be expunged from a library only if there are no import names (of other specifications in the library) bound to it. Since import names can be bound to only trait specifications, expunging a routine specification from a library is always allowed.

Operation UNBIND (discussed later) can be used to remove the binding of the import names bound to a specification to be expunged. A user can get a list of all the specifications in a library that contain at least one import name which is bound a particular specification using the COMPILE operation. COMPILE operation has been described in chapter 4.

<table>
<thead>
<tr>
<th>Routine</th>
<th>EXPUNGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>obj: Library, s: LibSpec</td>
</tr>
<tr>
<td>Returns</td>
<td>nothing</td>
</tr>
<tr>
<td>Includes</td>
<td>Library, LibSpec, TableOfBinding</td>
</tr>
</tbody>
</table>

| Pre      | s ∈ I↑ |
| Changes  | nothing |
| Post     | signal ("SPECIFICATION TO BE EXPUNGED DOES NOT EXIST IN THE LIBRARY") |

| Pre | # isATrait (# getInputSpec (s)) ∧ ∃ s1: LibSpec, [s1 ∈ I↑ ∧ s ∈ # getBindings (s1)] |
| Changes | nothing |
| Post | signal ("SPECIFICATION TO BE EXPUNGED HAS AN IMPORT NAME BOUND TO IT") |

| Pre | s ∈ I↑ ∧ (∀ s1: LibSpec, s1 ∈ I↑ ⇒ s ∈ # getBindings (s1)) |
| Changes | I |
| Post | I↑ = = # removeLibSpec (I↑, s) |
Specifications can be copied from one library to another using the COPY operation. What is actually copied is the text of the specification and name of its author. A timestamp, which is unique with respect to the library into which the specification is copied, is associated with the new specification. Also the new specification is unclassified, and its import names are unbound. The COPY operation can also be used for creating copies of a particular specification in the same library with different timestamps.

<table>
<thead>
<tr>
<th>Routine</th>
<th>COPY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>objI1, I2: Library, s: LibSpec</td>
</tr>
<tr>
<td>Returns</td>
<td>nothing</td>
</tr>
<tr>
<td>Includes</td>
<td>Library, LibSpec, TimeStamp</td>
</tr>
</tbody>
</table>

**Pre**

\[ s \in I1 \uparrow \]

**Changes**

nothing

**Post**

signal ("SPECIFICATION TO BE COPIED DOES NOT EXIST IN THE LIBRARY")

**Pre**

\[ s \in I1 \uparrow \]

**Changes**

I2

**Post**

\[ \exists ts: TimeStamp, \]

\[ \forall s1: LibSpec, \]

\[ s1 \in I2' \uparrow \Rightarrow \# \text{getTimestamp} (s1) < ts \] \land

\[ I2' \uparrow = = \# \text{addLibSpec} (I2' \uparrow, \]

\[ \# \text{createLibSpec} ( \]

\[ \# \text{getInputSpec} (s), \]

\[ \# \text{getAuthor} (s, ts)) \]
REPLACE

This operation is useful in situations when the text of a specification needs to be replaced. The timestamp, the author name and the classification of the specification remain the same. The
import names of the new specification have to bound again. The binding of an import name
which is bound to the specification being replaced is not affected. Note that the effect of a
REPLACE operation on the state of a library is not equivalent to a sequence of EXPUNGE and
INSTALL operations because REPLACE does not modify the timestamp of a specification
whereas INSTALL always creates a new timestamp.

---

Routine  REPLACE
Input     obj: Library, s: LibSpec, i: InputSpec
Returns   nothing
Includes  Library, LibSpec

Pre       s ∈ l↑
Changes   nothing
Post      signal ("SPECIFICATION TO BE REPLACED DOES
           NOT EXIST IN THE LIBRARY")

Pre       s ∈ l↑
Changes   l
Post      l↑ = = # addLibSpec ( # removeLibSpec (l↑', s),
           # classify ( # createLibSpec (i,
                        # getAuthor (s),
                        # getTimeStamp (s)),
           # getClassification (s))

---
CLASSIFY

This operation is useful for classifying a specification under a set of classes. A specification has to be installed in a library before it can be classified. CLASSIFY simply adds to the list of classes under which a specification is classified (i.e. CLASSIFY does not affect the set of classes under which a specification is already classified.) Often before a user can supply a list of classes, he might wish to examine the set of classes in the classification scheme. The operations for traversing and examining the classes of a classification scheme are given in chapter 4.

<table>
<thead>
<tr>
<th>Routine</th>
<th>CLASSIFY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>obj l: Library, s: LibSpec, csfn: SetOfClasses</td>
</tr>
<tr>
<td>Returns</td>
<td>nothing</td>
</tr>
<tr>
<td>Includes</td>
<td>Library, LibSpec, CS, SetOfClass</td>
</tr>
</tbody>
</table>

Pre
\[ s \notin l \uparrow \]

Changes
nothing

Post
signal ("SPECIFICATION TO BE CLASSIFIED DOES NOT EXIST IN THE LIBRARY")

Pre
\[ \exists cl: Class, \]
\[ [cl \in csfn \land \neg \# \text{validClass} (\# \text{getCS} (l\uparrow), cl)] \]

Changes
nothing

Post
signal ("AT LEAST ONE OF CLASSES DOES NOT EXIST IN THE LIBRARY'S CS")

Pre
\[ s \in l\uparrow \land \]
\[ [\forall cl: Class, \]
\[ \quad cl \in csfn \Rightarrow \# \text{validClass} (\# \text{getCS} (l\uparrow), cl)] \]

Changes
1

Post
\[ l\uparrow = = \# \text{addLibSpec} (\# \text{removeLibSpec} (l\uparrow, s), \]
\[ \quad \# \text{classify} (s, (\# \text{getClassification} (s) \cup csfn))) \]
This operation is useful for deleting a set of classes from the classification of a specification.

---

Routine: UNCLASSIFY
Input: obj: Library, s: LibSpec, csfn: SetOfClass
Returns: nothing
Includes: Library, LibSpec, CS, SetOfClass

Pre: NOT s ∈ I↑
Changes: nothing
Post: signal ("SPECIFICATION TO BE UNCLASSIFIED

DOES NOT EXIST IN THE LIBRARY")

Pre: ∃ cl: Class,

[cl ∈ csfn ∧ (∄ # validClass (# getCS (I↑), cl))]

Changes: nothing
Post: signal ("AT LEAST ONE OF CLASSES DOES NOT

EXIST IN THE LIBRARY'S CS")

Pre: s ∈ I↑ ∧

∀ cl: Class,

[cl ∈ csfn ⇒ # validClass (# getCS (I↑), cl)]

Changes: 1
Post: I↑ = # addLibSpec (# removeLibSpec (I↑, s),

# classify (s, (# getClassification (s) − csfn)))
BIND

This operation is useful for binding an import name of a specification in a library to a trait specification in the same library. If the import name is already bound, the BIND operation replaces the old binding.

<table>
<thead>
<tr>
<th>Routine</th>
<th>BIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>obj I: Library, s: LibSpec, in: ImportName, bs: LibSpec</td>
</tr>
<tr>
<td>Returns</td>
<td>nothing</td>
</tr>
<tr>
<td>Includes</td>
<td>Library, LibSpec, TableOfBinding, InputSpec, SetOfImportName</td>
</tr>
</tbody>
</table>

**Pre**

\[s \notin I\]

**Changes**

nothing

**Post**

signal ("SPECIFICATION CONTAINING THE " + "IMPORT NAME DOES NOT EXIST IN THE LIBRARY")

**Pre**

\[in \notin \# getImportNames (\# getInputSpec (s))\]

**Changes**

nothing

**Post**

signal ("SPECIFICATION DOES NOT CONTAIN THE IMPORT NAME TO BE BOUND")

**Pre**

\[bs \notin I\]

**Changes**

nothing

**Post**

signal ("SPECIFICATION BEING BOUND TO DOES NOT EXIST IN THE LIBRARY")

**Pre**

\[\neg \# isATrait (\# getInputSpec (bs))\]

**Changes**

nothing

**Post**

signal ("SPECIFICATION BEING BOUND TO IS NOT A TRAIT SPECIFICATION")

**Pre**

\[s \in I \land bs \in I \land in \in \# getImportNames (\# getInputSpec (s)) \land
   \# isATrait (\# getInputSpec (bs))\]

**Changes**

I

**Post**

I' = = \# addLibSpec (\# removeLibSpec (I', s),
                  \# bind (s, \# addBinding (\# getBindings (s), in, bs)))
UNBIND

This operation is useful for removing the binding of an import name of a specification.

Routine UNBIND
Input obj: Library, s: LibSpec, in: ImportName
Returns nothing
Includes Library, LibSpec, TableOfBinding, InputSpec, SetOfImportName

Pre s ∈ I↑
Changes nothing
Post signal ("SPECIFICATION CONTAINING THE IMPORT NAME DOES NOT EXIST IN THE LIBRARY")

Pre in ∈ # getInputNames (# getInputSpec (s))
Changes nothing
Post signal ("IMPORT NAME TO BE UNBOUND IS NOT PRESENT IN THE SPECIFICATION")

Pre in ∈ # getInputNames (# getInputSpec (s)) ∧
in ∈ ∈ # getBindings (s)
Changes nothing
Post signal ("IMPORT NAME TO BE UNBOUND IS NOT BOUND")

Pre s ∈ I↑ ∧
in ∈ ∈ # getInputNames (# getInputSpec (s)) ∧
in ∈ ∈ # getBindings (s)
Changes I
Post I↑ = = # addLibSpec (# removeLibSpec (I↑, s),
                # bind (s, # deleteBinding (# getBindings (s), in)))
EXTEND

This operation is useful for defining a new class in the classification scheme of a library. This operation can also be used for defining the root class of the classification scheme. In order to define a new class, the user must first remove all the classes in the existing classification scheme, including the root class. Operation REMOVE described can be used for removing a class.

<table>
<thead>
<tr>
<th>Routine</th>
<th>extend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>obj l: Library, oldClass: Class, newClass: ClassName</td>
</tr>
<tr>
<td>Returns</td>
<td>nothing</td>
</tr>
<tr>
<td>Includes</td>
<td>Library, CS, Class</td>
</tr>
<tr>
<td>Pre</td>
<td># csExists (l↑) ∧ # size (oldClass) = 0</td>
</tr>
<tr>
<td>Changes</td>
<td>nothing</td>
</tr>
<tr>
<td>Post</td>
<td>signal (&quot;CANNOT DEFINE A NEW ROOT CLASS BECAUSE A ROOT CLASS ALREADY EXISTS&quot;)</td>
</tr>
</tbody>
</table>

Pre

# validClass (# getCS (l↑), # append (oldClass, newClass)) ∧
# CSexists (l↑)

Changes

nothing

Post

signal ("THE CLASS TO BE ADDED ALREADY EXISTS IN THE CS")

Pre

# size (oldClass) = 0 ∧ ¬ # csExists (l↑)

Changes

l

Post

l↑ = = # addCS (l'↑, # makeCS (newClass))

Pre

# size (oldClass) > 0 ∧
¬ # validClass (# getCS (l↑), # append (oldClass, newClass)) ∧
# CSexists (l↑)

Changes

l

Post

l↑ = = # addCS (# deleteCS (l'↑),
# addSubCS (cs, oldClass, newClass))
REMOVE

This operation is useful for removing a part of a CS. One of the arguments to the REMOVE operation is a class. The effect of this operation is to remove that part of the CS whose root class is same as the argument class. A sub CS can be deleted from a CS only if every class in the sub CS has no specification classified under it. Note that operation REMOVE can also be used for removing the complete CS from a library.

---

Routine: remove
Input: obj: Library, cl: Class
Returns: nothing
Includes: Library, CS, Class

Pre: \neg \text{validclass} (\# \text{getCS} (l\uparrow), cl)

Changes: nothing
Post: signal ("ARGUMENT CLASS IS NOT A VALID CLASS IN THE LIBRARY")

Pre: \exists cl1: Class, s: LibSpec,
    [[s \in l\uparrow \land
    \# \text{validClass} (\# \text{getCS} (l\uparrow), \# \text{concat} (cl, cl1)) \land
    \# \text{concat} (cl, cl1) \in \# \text{getClassification} (s)]

Changes: nothing
Post: signal ("CANNOT REMOVE THE DESIRED PART FROM THE CS: SPECIFICATIONS ARE CLASSIFIED UNDER IT'S CLASSES")

Pre: \# size (cl1) = 1 \land
    \# \text{getRootClass} (\# \text{getCS} (l\uparrow)) = \# \text{first} (cl) \land
    \forall s: LibSpec,
    [s \in l\uparrow \Rightarrow (\# \text{getClassification} (s) = = \# \text{emptySetOfClass})]

Changes: l
Post: l\uparrow = = \# \text{deleteCS} (l''\uparrow)

Pre: \# size (cl1) > 1 \land
    \# \text{validclass} (\# \text{getCS} (l\uparrow), cl) \land
    \forall cl1: Class, s: LibSpec,
    [[s \in l\uparrow \land
    \# \text{validClass} (\# \text{getCS} (l\uparrow), \# \text{concat} (cl, cl1)) \Rightarrow
    \# \text{concat} (cl, cl1) \notin \# \text{getClassification} (s)]

Changes: l
Post: l\uparrow = = \# \text{addCS} (\# \text{deleteCS} (l''\uparrow), \# \text{deleteSubCS} (cs, cl))
4. Browsing Through A Library

In this chapter we present a set of operations, and corresponding routines, for browsing through our specification library. Before we present the operations, we informally discuss what we mean by the term "browsing". In section 4.2 we present an abstraction for storing useful temporary results of the various browsing operations, called Catalogs. In chapter 3 we had introduced an abstraction called InputSpec corresponding to the specification that can be installed in a library. The browsing operations introduced in this chapter require an InputSpec to be partitioned further into smaller components. In section 4.3 we modify trait InputSpec (presented in chapter 3) to accommodate this partitioning of an InputSpec. In sections 4.4 to 4.7 we present operations for creating and displaying catalogs. In section 4.8 we present a set of operations for using a library's CS as a roadmap for systematically browsing through a library.

4.1 What do we mean by Browsing

We developed our intuition for designing the set of browsing operations based on our experience from browsing through a conventional library of books. Let us briefly discuss some of the aspects of browsing through a library of books. (Much of what we say in the remainder of this section is common knowledge that should be familiar to anybody who has used a library of books.)

The books in a library are arranged on the shelves in an order based on their book number. For our purposes, a book number is a library-wide unique number associated with every book in a library. Before a user can locate a book on the shelves he needs to know its book number.
Book libraries provide a number of catalogs for determining a book’s book number. A catalog is a collection of cards arranged in a particular order with each card containing information about a particular book in the library. This information includes things such as the book’s title, author(s), number, abstract, etc. Libraries usually contain three catalogs: a title catalog in which the cards are arranged alphabetically by a book’s title, an author catalog in which the cards are arranged alphabetically by a book’s author, and a subject catalog in which the cards are arranged alphabetically by subject headings.

It is fairly straightforward to use two of these catalogs: the title catalog and the author catalog. If the user knows the title (or the author) of a book, using the title (or the author catalog,) he can access the appropriate catalog card, and use the book number on this catalog card to access the book on the shelf. Conceptually, locating a book using its title or author is similar to locating a file in a file system by its file name, and not a very interesting kind of browsing.

Often a user is looking for a book on a particular subject, say S, and has an idea about the contents of a book. Let us call this “abstract book” for which a user is looking the book of interest (BOI.) Note that there can be more than one book in a library that satisfy the requirements of the BOI.

When the user attempts to look for the books classified under the subject S in the subject catalog, three possibilities can occur:

1. Subject S is not present in the catalog, in which case the user typically guesses another subject under which the BOI might be classified.

---

1. Sometimes a user selects two (or more) subjects, and is interested in examining only those books that are classified under both (or all) the subjects.
2. Subject S is present in the catalog, in which case the user examines the
information on the cards corresponding to the books classified under subject S,
and checks whether any of the books could possibly satisfy the requirements of
the BOI. Note that the information on the catalog card is quite minimal, and in
order to make sure that a book satisfies the requirements of the BOI the user has
to examine the book itself.

3. Under subject S, there are references like "ALSO SEE \{subject\}," where
\{subject\} denotes a list of subjects. This list of subjects in the "ALSO SEE
\{subject\}" are references to subjects that are related to S, and are suggestions to
a user that the BOI might be classified under these related subject. In such cases,
if the user feels that the BOI may be classified under any of the related subjects,
he can proceed to examine the books under that subjects.

Locating a book using the subject catalog might require a user to examine the books under a lot
of subjects before he finds a book satisfying the requirements of the BOI. It might also require a
user to frequently shift between browsing through the subject catalog, and going to the shelves to
examine books that could possibly be the BOI.

We would like to informally characterize the above kind of browsing as a trial and error
activity in which the user begins with a large search space, and browses by selecting (or
eliminating) books from the search space until he finds the BOI (or feels that the library does not
contain the BOI). The level of detail of information required at different stages of browsing is
usually different. In the initial stages, when the search space is large, the user usually needs
relatively little information information about the books. e.g. only their title and abstracts. During
the final stages of browsing, when the search space is relatively small, the user usually needs
detailed information about contents of the books.

Frequently, there is another kind of browsing activity in which a user is interested -- the kind
where he is simply interested in examining the books in a library not looking for any book in
particular. Actually, rather than the whole library, often a user is interested in examining the
books classified under certain subjects. In a book library, a user usually does this kind of
browsing by scanning through the titles of the books on the shelves, and examining contents of
books that interest him.

A user of our specification library can use the CS of the library as a road map for systematically browsing through specifications classified under the different classes of the CS. One of the concern of a user in this kind of browsing is that he would like to browse through the library in a systematic fashion, and avoid examining any specification more than once. In section 4.8 we will present routines for operations for navigating through the classification scheme of the library in a systematic fashion.

In the next section we will describe catalogs in our specifications library.

4.2 Catalogs

We exploit the machine support for the our specification library to support a general type of catalogs. Our catalogs are different from the catalogs in conventional book libraries in two major ways.

1. In book libraries a catalog contains information about the books in the library. The catalogs in our library are more general in that they contain specifications themselves, and not just some information about them. In a conventional library, the catalog card for a particular book contains a fixed amount of information about the book. In order to get more information about a book, the user has to examine the book. Since our catalogs contain specifications, potentially all the information about a specification is available in any catalog that contains the specification. In other words, unlike book libraries, where there is a division of information between the books and the catalogs, there is no such division of information in our specification library. The operation for displaying a catalog controls the detail of information displayed to a user about the specifications in a catalog.

2. The number of catalogs in a book library are fixed In our library, a user can create his own catalogs. Below we outline six operations for creating new catalogs. Our intent here is merely to give a reader an idea of what we mean by creating catalogs. We will describe these operations in more detail in the remainder of this chapter.
1. COMPILÉ: For creating a catalog of specifications from a library such that all the specifications in the catalog satisfy a proposition (section 4.4).

2. FILTER: For creating a new catalog from a catalog such that all the specifications in the new catalog satisfy a proposition (section 4.4).

3. RETAIN: For creating a new catalog containing the selected specifications from a catalog (section 4.5).

4. EXCLUDE: For creating a new catalog containing the unselected specifications in a catalog (section 4.5).

5. MERGE: For creating a catalog containing the union of the specifications in two catalogs (section 4.6).

6. DIFFERENCE: For creating a catalog containing the difference of the specifications between two catalogs (section 4.6). (By difference we refer to the familiar set theoretic difference operation.)

In addition to the above six operations for creating new catalogs, we describe two more operations on catalogs. They are

1. SELECT: For selecting a specification in a catalog (section 4.5).

2. DISPLAY: For displaying the specifications in a catalog (section 4.7).

Trait Catalog (given below) describes a catalog in our specification library. A catalog is a set of specifications in which some of the specifications may be marked as being selected.

```
trait Catalog

include Set

[Catalog --> Set,
 LibSpec --> Elem,
 # createCatalog --> # create,
 # insertLibSpec --> # insert,
 # deleteLibSpec --> # delete,
 # hasLibSpec --> # has]

without # union, # difference
```
there exist functions

\[
\text{# selectLibSpec:Catalog} \times \text{LibSpec} \rightarrow \text{Catalog}
\]
\[
\text{# isSelected:Catalog} \times \text{LibSpec} \rightarrow \text{Boolean}
\]

constrains [Catalog] so that

\[
\text{# deleteLibSpec ( # selectLibSpec (c, s1), s2) = = if s1 = s2}
\]
\[
\text{then # deleteSpec (c, s2)}
\]
\[
\text{else # selectLibSpec ( # deleteLibSpec (c, s2), s1)}
\]

\[
\text{# hasLibSpec ( # selectLibSpec (c, s1), s2) = = # hasLibSpec (c, s2)}
\]

\[
\text{# isSelected ( # createCatalog, s) = = false}
\]
\[
\text{# isSelected ( # insertLibSpec (c, s1), s2) = = # isSelected (c, s2)}
\]
\[
\text{# isSelected ( # selectLibSpec (c, s1), s2) = = if s1 = s2}
\]
\[
\text{then true}
\]
\[
\text{else # isSelected (c, s2)}
\]

end Catalog
4.3 Modifying Trait InputSpec

From chapter 3 recall that trait InputSpec described the abstraction that corresponded to a specification that could be installed in a library. We referred to this abstraction as InputSpec. Also recall that abstraction SpecificationText, one of the components of an InputSpec, was simply described as a string of characters (see trait SpecificationText.)

We could assume a SpecificationText to be a single indivisible abstraction because the operations for maintaining a library did not require any information about the different components of a SpecificationText. On the other hand, the browsing operations require a "finer" view of a SpecificationText. More specifically,

1. The operation for displaying the specifications permits a user to request only some of the components of a SpecificationText to be displayed.

2. In section 4.4, we present operations for creating a catalog using a proposition. We allow propositions to be defined on the components of a specification, and one of the specification components on which a proposition can be defined is the name of the specification. In order to access the name of specification, we could no longer afford to view a SpecificationText as an indivisible abstraction.

Below we give a new specification of trait InputSpec that incorporates a "finer" view of a SpecificationText.

```
trait InputSpec

  Include Comments, SpecName, AuxDef, SetOfImportNames

  InputSpec = record
    [Comments: Comments,
     SpecName: SpecName,
     AuxDef: AuxDef,
     ImportNames: SetOfImportNames,
     SpecType: Boolean]

end InputSpec
```
The new specification a `SpecificationText` as being composed of three abstractions: `Comments` corresponding to the comments in English that usually precede a specification, `SpecName` corresponding to the name of a specification, and `AuxDef` containing the body of the specification. For a trait specification, `AuxDef` includes the operators and the axioms of the trait; for a routine specification, `AuxDef` includes the input, returns, pre, changes, and post clauses of the routine. Traits `Comments`, `SpecName`, and `AuxDef`, which are simply strings, are given below.

<table>
<thead>
<tr>
<th>Trait</th>
<th>Comments</th>
<th>[Comments \rightarrow String]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>include String</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>Comments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SpecName</td>
<td>[SpecName \rightarrow String]</td>
</tr>
<tr>
<td></td>
<td>include String</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>SpecName</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AuxDef</td>
<td>[AuxDef \rightarrow String]</td>
</tr>
<tr>
<td></td>
<td>include String</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>AuxDef</td>
<td></td>
</tr>
</tbody>
</table>

At this point, a reader may ask us the reason for deciding to modify trait `InputSpec` rather than trait `SpecificationText`. Since we are partitioning a `SpecificationText`, it would seem more appropriate to modify trait `SpecificationText` and leave trait `InputSpec` as it is. Let us suppose that we modified trait `SpecificationText` instead of trait `InputSpec`. Then in order to obtain the values for the components `Comments`, `SpecName`, and `AuxDef` of a `LibSpec`, we would have to go through three levels of abstractions. For example, to obtain the `SpecName` of a `LibSpec`, we would first have to obtain the `InputSpec` component of a `LibSpec`, then obtain the `SpecificationText` component from the `InputSpec`, and finally obtain the `SpecName` component from the `SpecificationText`. That is, to denote the `SpecName` of a `LibSpec`, say I, we would have to use the expression `#getSpecName (#getSpecificationText (#getInputSpec (I)))`. By
modifying trait InputSpec, we can now denote the SpecName of a LibSpec, l, with the expression
# getSpecName (# getInputSpec (l)). Modifying trait SpecificationText would have introduced
an unnecessary level of abstraction in describing an InputSpec, and by modifying trait InputSpec
we simply avoided that extra level of abstraction.

Modifying trait InputSpec does not affect the other specifications introduced in chapter 3.
More specifically, trait LibSpec, which is the only specification in chapter 3 that includes trait
InputSpec, remains unaffected.
4.4 Compiling and Filtering Catalogs

A user can create personal catalogs from a library using the COMPILE operation. The user characterizes the catalog to be compiled by specifying a Proposition on the components of a specification, and the resulting catalog contains all the specifications in the library that satisfy this proposition. For instance, a user can use the COMPILE operation to create a catalog of all the specifications that were installed in a library during a particular period of time.

In its simplest form, a proposition can be defined on six components of a specification. The six components are:

- the name of a specification
- the author of a specification
- the timestamp of a specification
- the classes under which a specification is classified (classification)
- the binding of a specification's import names (bindings)
- the type of the specification (i.e. whether a specification is a trait or a routine)

We feel it would be useful to define propositions on other components of specifications. For example, a proposition that checks whether a particular operator in a trait is a commutative operator is an interesting proposition. In their most general form, such propositions require knowledge about the theorem proving aspects of a specification, and are outside the scope of this thesis.

In addition to the traits introduced in chapter 3, trait Proposition requires definition of two other traits.
1. Trait Period describing a period of timestamps.

2. Trait Proposition describing a proposition on a single component of a specification.

---

```plaintext
trait Period

include TimeStamp

Period = record [#initial: TimeStamp,
    # final: TimeStamp]

there exist functions

    # within: Period X TimeStamp --> Boolean

constrains [SimplePredicate] so that

    # within (#createPeriod (t1, t2), t) = (t1 < t \land t < t2)

end Period
```

---

Since the classification and the bindings of a specifications are sets, propositions on these component test for set membership. Propositions on the name and author test for equality. A proposition on timestamp is constructed by giving a pair of timestamps representing a period of time, and the proposition tests whether the timestamp of the specification lies within that period of time.

---

```plaintext
trait SimpleProposition

include LibSpec, SpecName, Author, Class,
    ImportName, Period, SetOfClasses, TableOfBindings

there exist functions

    # nameConstant: SpecName --> Proposition
    # authorConstant: Author --> Proposition
```
# classConstant: Class \rightarrow Proposition
# importConstant: LibSpec \rightarrow Proposition
# periodConstant: Period \rightarrow Proposition
# isTrait: Boolean \rightarrow Proposition
# satisfies: Proposition \times LibSpec \rightarrow Boolean

constrains [Proposition] so that

# satisfies ( # nameConstant (s), I) = =
   s = # getSpecName (# getInputSpec (I))
# satisfies ( # authorConstant (a), I) = =
   a = # getAuthor (I)
# satisfies ( # classConstant (c), I) = =
   c \in # getClassification (I)
# satisfies ( # importConstant (I1), I2) = =
   \# containsLibSpec ( # getBindings (I2), I1)
# satisfies ( # periodConstant (p), I) = =
   \# within (p, \# getTimestamp (I))
# satisfies ( # isTrait (b), I) = =
   \# isATrait ( # getInputSpec (I))

end SimpleProposition

Propositions can be combined using boolean operators \( \land \), \( \lor \), and \( \neg \) to form new propositions. Trait Proposition given below describes the most general form of propositions involving boolean operators.

trait Proposition

include SimpleProposition

there exist functions

# and: Proposition \times Proposition \rightarrow Proposition
# or: Proposition \times Proposition \rightarrow Proposition
# not: Proposition \rightarrow Proposition

constrains [Proposition] so that

# satisfies ( # and (p1, p2), I) = = # satisfies (p1, I) \land # satisfies (p2, I)
# satisfies ( # or (p1, p2), I) = = # satisfies (p1, I) \lor # satisfies (p2, I)
# satisfies ( # not (p), I) = = \neg # satisfies (p, I)
Once we have developed trait Proposition, specification of the COMPILE operation is fairly straightforward.

**Routine** COMPILE  
**Input** obj l: Library, p: Proposition  
**Returns** obj c: Catalog  
**Includes** Library, Catalog, Predicate  
**Pre** true  
**Changes** nothing  
**Post** $\forall s: LibSpec, [s \in c \models \models = (s \in l) \land \#satisfies(p, s)]$

A FILTER operation is similar to a COMPILE operation. The difference between the two operations is that while the COMPILE operation is useful for creating a new catalog from a Library, the FILTER operation is useful for creating a new catalog from a Catalog.

**Routine** FILTER  
**Input** obj c: Catalog, p: Proposition  
**Returns** obj c1: Catalog  
**Includes** Predicate, Catalog  
**Pre** true  
**Changes** nothing  
**Post** $\forall s: LibSpec, [s \in c1 \models \models = (s \in c) \land \#satisfies(p, s)]$
4.5 Scanning through Catalogs

In the previous section we introduced the operation filter which is useful for creating a catalog from another catalog by specifying a proposition. In this section we present operations that provide a different way of creating a catalog from another catalog: by explicitly selecting specifications in a catalog, and then creating a catalog all the selected specifications. More specifically, we present introduce three operations.

1. SELECT: For selecting a specification from a catalog.

2. RETAIN: For creating a new catalog containing the specifications that have been selected in a catalog.

3. EXCLUDE: For creating a new catalog containing the specifications that have not been selected in a catalog.

The following browsing scenario will illustrate the usefulness of operations SELECT, and RETAIN. Suppose a user is interested in locating a trait specification satisfying certain properties. He begins by examining the CS of a library, and selects a number of classes that could possibly contain the SOI. He compiles a catalog, C, of all trait specifications that are classified under at least one of the selected classes. He then invokes the display operation with appropriate arguments to display the SpecName and Comments of the specifications in the catalog displayed at the user interface. (The operation for displaying a catalog will be discussed later. For the present assume a user can specify the particular components of a specification that he wants to display.) On examining the information present in the SpecName and Comments of the specifications in C, he feels that some of these specifications could possibly be the SOI, and wants to examine their AuxDef (i.e. operators and axioms.) The user can use the SELECT and RETAIN operation to in such a situation.
As the user scans through the SpecName and Comments of the specifications in C, he can select the specifications that we want to examine. After the user has finished scanning through the catalog, he can use the retain operation to create a new catalog, C1, containing all the selected specifications from C, and use the display operation to display the AuxDef of the traits in C1 at the user interface.

Routines describing the SELECT and the RETAIN operations are given below.

---

**Routine** SELECT

**Input** obj c: Catalog, s: LibSpec

**Returns** nothing

**Includes** Catalog

**Pre** s ∈ c

**Changes** nothing

**Post** signal ("LIBSPEC TO BE SELECTED DOES NOT EXIST IN THE CATALOG")

**Pre** s ∈ c

**Changes** c

**Post** c = #selectLibSpec(c, s)

---

**Routine** RETAIN

**Input** obj c: Catalog

**Returns** obj c1: Catalog

**Includes** Catalog

**Pre** true

**Changes** nothing

**Post** ∀ s: LibSpec,
[(#isSelected (c↑, s)) = = s ∈ c1↑) \land \neg #isSelected (c1↑, s)]

There might be situations when a user would like to select all except some of the specifications in a catalog. In such situations, it would be more convenient to select the specifications that should not be included in the new catalog, and create a new catalog of unselected specifications. Operation EXCLUDE is useful in such situations. The routine corresponding to the EXCLUDE operation is given below.

---

**Routine**  EXCLUDE

**Input**  obj c: Catalog

**Returns**  obj c1: Catalog

**Includes**  Catalog

**Pre**  true

**Changes**  nothing

**Post**  \( \forall s: \text{Lit Spec}, \)

\[
[(s ∈ c↑ \land \neg #isSelected (c↑, s)) = = s ∈ c1↑) \land
(\neg #isSelected (c1↑, s))]
\]
4.6 Combining Catalogs

In this section we present two new ways of creating catalogs. They are:

1. MERGE: For merging the specifications in two catalogs into a new catalog.

2. DIFFERENCE: For creating a new catalog containing specifications that are present in one catalog, and not present in another.

The MERGE operation is useful in situations when a user finds it convenient to browse through catalogs of small sizes and merge the results of browsing through these catalogs into a single catalog. For example, in the initial stages of browsing, when the search space can contain a large number of specifications, a user might find it more convenient to browse separately through smaller catalogs, and combine the results of these separate browsings later into a single catalog. The MERGE operation returns a new catalog that is the union of the specifications in two catalogs.

---

Routine MERGE
Input obj c1: Catalog, obj c2: Catalog
Returns obj c3: Catalog
Includes Catalog
Pre true
Changes nothing
Post ∀ s: LibSpec,

\[ [(s ∈ c3∗ = (s ∈ c1∗ ∧ s ∈ c2∗)) ∧ ¬ #isSelected (c3∗, s)] \]

---

The following browsing scenario illustrates the usefulness of a DIFFERENCE operation. After examining the classes in the CS of a library, a user has compiled a catalog, C, of specifications classified under a set of classes that have a "good" possibility of containing the
SOI. On examining the specifications in C he does not find any specification that satisfy the requirements of the SOI. He decides to expand his search space, and chooses a new set of classes that have a "faint" possibility of containing the SOI. He compiles a new catalog, C', containing specifications classified under these new classes. It is possible that C' contains specifications that he has already examined as part of C. To avoid re-examining the specifications common to C and C', the user would like to remove the specifications that are common to C and C' from C'.

The DIFFERENCE operation takes two catalogs, C₁ and C₂ as its arguments, and returns a catalog containing specifications present in C₁ and not in C₂. The routine describing the DIFFERENCE operation is given below.

---

**Routine** DIFFERENCE

**Input** obj c₁: Catalog, obj c₂: Catalog

**Returns** obj c₃: Catalog

**Includes** Catalog

**Pre** true

**Changes** nothing

**Post** ∀ s: LibSpec,

\[\{(s ∈ c₃) \iff (s ∈ c₁ ∧ s ∈ c₂)\} ∧ \neg #\text{isSelect}(c₃, s)\]
4.7 Displaying Catalogs

The operation for displaying the specifications in a catalog allows a user to control two factors:

1. The components of the specifications that he wants to display, and
2. The order in which he wants the specifications to be displayed.

In most situations when a catalog contains a large number of specifications, the user would probably want to examine only the short components of the specifications, and avoid cluttering up the display with unrequired components. By short components of a specification we refer to those components that occupy relatively little space at the user interface such as the SpecName, Author, TimeStamp, etc.

A user of our library can specify the specifications in a catalog to be displayed in three different orders. The three orderings are based on using the

1. SpecName of a specification as the sorting index,
2. Author of a specification as the sorting index, and
3. TimeStamp of a specification as the sorting index.

Parts of the routine describing the DISPLAY operation is given below. The complete specification of the DISPLAY operation is given later at the end of the section. Our intent in presenting the partial specification is to give the readers an idea about the signature of the routine operation. The second input argument to the display routine describes the components of the specification that should be displayed. The third input argument to the display routine describes the order in which the specifications are to be displayed. The object returned by the display routine is a special catalog that can be displayed at the user interface.
Routine DISPLAY

Input C: Catalog, sel: Selections, sf: SortFunction

Returns D: DisplayCatalog

Before we present the traits useful for describing the pre and post conditions of the DISPLAY routine, we give a brief overview of some of the abstractions that will be used to describe the effect of the DISPLAY operation.

1. Selections contains information about the components of a specification that should be displayed at the user interface.

2. SortFunction describes the order in which the specifications should be displayed.

3. DisplaySpec is a specification that can be displayed at the user interface.

4. As mentioned earlier, a DisplayCatalog is a special catalog that can be displayed at the user interface. A DisplayCatalog is a sequence of DisplaySpec with no DisplaySpec occurring more than once in it.

5. OrderedCatalog is a sequence of LibSpecs with no LibSpec occurring more than once in it.

First we present trait Selections.

trait Selections

there exist functions

  # noSelection: --> Selections
  # selectSpecName: Selections --> Selections
  # selectComments: Selections --> Selections
  # selectAuxiliaryDef: Selections --> Selections
  # selectAuthor: Selections --> Selections
  # selectTimeStamp: Selections --> Selections
  # selectClassification: Selections --> Selections
  # selectBindings: Selections --> Selections
Trait DisplaySpec describes the specification that can be displayed at the user interface.

trait DisplaySpec

there exist functions

# emptyDisplaySpec: --> DisplaySpec
# insertSpecName: DisplaySpec X SpecName --> DisplaySpec
# insertComments: DisplaySpec X Comments --> DisplaySpec
# insertAuxiliaryDef: DisplaySpec X AuxDef --> DisplaySpec
# insertAuthor: DisplaySpec X Author --> DisplaySpec
# insertTimeStamp: DisplaySpec X TimeStamp --> DisplaySpec
# insertClassification: DisplaySpec X SetOfClass --> DisplaySpec
# insertBindings: DisplaySpec X TableOfBindings --> DisplaySpec

end DisplaySpec

Trait Projection describes the construction of a DisplaySpec using a LibSpec and a Selections.

trait Projection

include Selections, DisplaySpec, LibSpec, InputSpec

there exist functions

# project: Selections X LibSpec --> DisplaySpec

constrains [ # project] so that

# project (# noSelections, l) = = # emptyDisplaySpec

# project (# selectSpecName(s), l) = =
  # insertSpecName (# project (s, l),
    # getSpecName (# getInputSpec (l))))
# project ( # selectComments(s), l) = =
  # insertComments ( # project (s, l),
  # getComments( # getInputSpec (l)))

# project ( # selectAuxiliaryDef(s), l) = =
  # insertAuxiliaryDef ( # project (s, l),
  # getAuxiliaryDef ( # getInputSpec (l)))

# project ( # selectAuthor(s), l) = =
  # insertAuthor ( # project (s, l), # getAuthor (l))

# project ( # selectTimeStep(s), l) = =
  # insertTimeStep ( # project (s, l), # getTimeStep (l))

# project ( # selectClassification(s), l) = =
  # insertClassification ( # project (s, l), # getClassification (l))

# project ( # selectBindings(s), l) = =
  # insertBindings ( # project (s, l), # getBindings (l))

end    Projection

Trait OrderedSet described below essentially describes a set in which the elements are ordered.

trait    OrderedSet

include    Set    [OrdSet -> Set]
            without    #union, #difference,
                        Integer

there exist functions

  # first:    OrdSet    -->    Elem
  # rest:     OrdSet    -->    OrdSet
  # size:     OrdSet    -->    Integer

constrains [OrdSet] so that

  % # first ( # create) is intentionally not defined %
  # first ( # insert (s, e)) = =
             if # size (s) = 0
              then e
else # first (s)

# rest (create) = = # create
# rest ( # insert (s, e)) = =
    if # size (s) = 0
        then # create
    else if # first (s) = e
        then # delete (s, e)
        else # insert ( # rest (s), e)

# size ( # create) = = 0
# size ( # insert (s, e)) = = if e ∈ s
    then # size (s)
    else # size (s) + 1

end OrderedSet

Trait OrderedCatalog describes a OrderedCatalog by renaming the operators of trait OrderedSet.

trait OrderedCatalog
include OrderedSet
[OrderedCatalog --> OrdSet,
LibSpec --> Elem,
# emptyOrderedCatalog --> # empty,
# insertLibSpec --> # insert]
end OrderedCatalog

Trait DisplayCatalog describes DisplayCatalog, the abstraction that can be displayed at the user interface. The axioms for operator # build describe how a Displaycatalog is constructed from a OrderedCatalog and Selections.

trait DisplayCatalog
include OrderedSet
[DisplayCatalog --> OrdSet,
DisplaySpec -> Elem,
    # emptyDisplayCatalog -> # empty,
    # insertDisplaySpec -> # insert,
    # libSpecEq -> # elemEq]

OrderedCatalog,
Projection

there exist functions

    # buildDisplayCatalog: OrderedCatalog X Selections -> DisplayCatalog

constrains [# buildDisplayCatalog] so that

    # buildDisplayCatalog (# emptyOrderedCatalog, sel) = =
        # emptyDisplayCatalog
    # buildDisplayCatalog (# insertLibSpec (oc, l), sel) = =
        # insertDisplaySpec (# buildDisplayCatalog (oc, sel), # project (sel, l))

end   DisplayCatalog


Trait SortFunction describes the abstraction that specifies the order in which the specifications of a catalog should be displayed.

trait   SortFunction

there exist functions

    # byAuthor: -> SortFunction
    # byTimeStamp: -> SortFunction
    # bySpecName: -> SortFunction

end   SortFunction


Trait Sort contains just one operator (and four axioms) for determining whether the specifications in a OrderedCatalog are sorted with respect to a Sortfunction or not.

trait   Sort
include OrderedCatalog, SortFunction

there exist functions

# isSorted: OrderedCatalog X SortFunction --> Boolean

constrains [ # isSorted] so that

# isSorted ( # create, sf) = = true

# isSorted ( # append (oc, s), # byAuthor) = =
if # size (oc) = 1
then true
else if # getAuthor ( # first (oc)) ≤
    # getAuthor ( # first ( # rest (oc))
then # isSorted (oc, # byAuthor)
else false

# isSorted ( # append (oc, s), # byTime) = =
if # size (oc) = 1
then true
else if # getTimestamp ( # first (oc)) <
    # getTimestamp ( # first ( # rest (oc))
then # isSorted (oc, # byTime)
else false

# isSorted ( # append (oc, s), # byName) = =
if # size (oc) = 1
then true
else if # getName ( # first (oc)) ≤
    # getName ( # first ( # rest (oc))
then # isSorted (oc, # byName)
else false

end Sort

The specification of the routine corresponding to the DISPLAY operation is given below.

Routine DISPLAY
Input obj c: Catalog, sel: Selections, sf: SortFunction
Returns d: DisplayCatalog
Includes Catalog, OrderedCatalog, Sort, DisplayCatalog
Pre: true
Changes: nothing
Post: \exists oc: OrderedCatalog,
\[
[\forall s: \text{LibSpec},
\quad [s \in c\uparrow = = s \in oc] \land
\quad \#\ isSorted (oc, sl) \land
\quad d = = \#\ buildDisplayCatalog (oc, sel)]
\]
4.8 Using the CS as a RoadMap for Browsing

Our goal in this section is to present a set of operations that are useful for systematically examining the specifications in a library using the library's CS as the road map. An example of a situation when a user might want to examine the specifications classified under various classes in a systematic fashion is when he wants to classify a recently installed specification. One of the requirements for systematic browsing is that the user should know about the classes whose specifications he has already examined, so that he can avoid examining the specifications in a class more than once.

We first introduce an abstraction called NavigationStatus that keeps track of two pieces of information:

1. the class in the CS that a user is currently "visiting," and
2. the classes in the CS that a user has already "visited."

Information about the classes visited by a user is maintained by marking the classes in the CS. The abstraction that maintains this information is called MarkedCS.

trait MarkedCS

include CS, Class

there exist functions

# CStoMarkedCS: CS --> MarkedCS
# markClass: MarkedCS X Class --> MarkedCS
# isMarked: MarkedCS X Class --> Boolean
# markedCStoCS: MarkedCS --> CS

constrains [MarkedCS] so that

# isMarked (# CStoMarkedCS (cs), cl) = = false
# isMarked (# markClass (cs, cl1), cl2) = 1 if cl1 = cl2
then true
else # isMarked (cs, cl2)

# markedCStoCS (# CStoMarkedCS (cs)) = = cs
# markedCStoCS (# markClass (cs, cl1)) = = # markedCStoCS (cs)

end MarkedCS

trait NavigationStatus

include MarkedCS, Class

NavigationStatus = record
  [CurrentClass: Class,
   VisitedClasses: MarkedCS]

end NavigationStatus

There are six operations for navigating through a CS. Since a CS is essentially a tree
structure, we have freely used the conventional tree terminology (i.e. the terminolgy for talking
about trees) in describing the effect of these six operations. The six operations are:

1. SHOW: For displaying the sub CS of a particular class in the CS.

2. BEGIN: For beginning the navigation of a CS. This operations is also useful for
   restarting the navigation of a CS.

3. UP: For defining the parent class of the current class as the new current class.
   If the parent has already been visited, the user is so informed.

4. DOWN: For defining an unvisited sub class of the current class as the new current class.
   If all the classes at the next lower level have already been visited, the user is so informed.

5. NEXT: For defining an unvisited class at the same level as the current class as the new current class.
   If all the classes at the same level (as the current class) have already been visited, the user is so informed.
6. JUMP: For defining a particular class in the CS as the current class. Note that UP, DOWN and NEXT are useful for visiting only unvisited classes. In some cases, a user might want to visit a class even though he has already visited it. In such situations a user can use the JUMP operation to visit classes that he has already visited.

In the remainder to this section, we present routine specifications for the above six operations.

---

**Routine** SHOW

**Input** obj l: Library, cl: Class

**Returns** cs: CS

**Includes** Library

**Pre** \( \neg \# \text{csExists}(l) \)

**Changes** nothing

**Post** signal ("THERE IS NO CLASSIFICATION SCHEME IN THE LIBRARY")

**Pre** \( \# \text{csExists}(l) \land \neg \# \text{validClass}(\# \text{markedCStoCS}(l), cl) \)

**Changes** nothing

**Post** signal ("THERE IS NO SUCH CLASS IN THE CLASSIFICATION SCHEME")

**Pre** \( \# \text{validClass}(\# \text{markedCStoCS}(l), cl) \)

**Changes** nothing

**Post** cs = = \# \text{getSubCS}(\# \text{getCS}(l), cl)

---

**Routine** BEGIN

**Input** obj l: Library

**Returns** obj ns: NavigationStatus

**Includes** Library, NavigationStatus

**Pre** \( \# \text{csExists}(l) \)
Changes nothing
Post \( \text{ns}^{\uparrow} = = \text{# createNavigationStatus (}
\quad \text{# getRoot (\text{# getCS (\text{\uparrow})})},
\quad \text{# CStoMarkedCS (\text{# getCS (\text{\uparrow}})))} \)

---

Routine UP
Input \( \text{ns: NavigationStatus} \)
Returns nothing
Includes NavigationStatus, Integer, Class

Pre \( \text{# size (\text{# getCurrentClass (\text{ns}^{\uparrow})})} = 1 \)
Changes nothing
Post \( \text{signal ("CURRENT CLASS IS THE ROOT CLASS")} \)

Pre \( \text{# isMarked (\text{# getVisitedClasses (\text{ns}^{\uparrow})}, \text{# parent (\text{# getCurrentClass (\text{ns}^{\uparrow})})})} \)
Changes nothing
Post \( \text{signal ("THE ABOVE CLASS HAS ALREADY BEEN VISITED")} \)

Pre \( \text{# size (\text{# getCurrentClass (\text{ns}^{\uparrow})})} > 1 \land
\quad \lnot \text{# isMarked (\text{# getVisitedClasses (\text{ns}^{\uparrow})}, \text{# parent (\text{# getCurrentClass (\text{ns}^{\uparrow})})})} \)
Changes \( \text{ns} \)
Post \( \text{ns}^{\uparrow} = = \text{# createNavigationStatus (}
\quad \text{# parent (\text{# getCurrentClass (\text{ns}^{\prime}\uparrow})},
\quad \text{# markClass (\text{ns}^{\prime}\uparrow}, \text{# parent (\text{# getCurrentClass (\text{ns}^{\prime}\uparrow}})))} \)

---

Routine DOWN
Input \( \text{obj ns: NavigationStatus} \)
Returns nothing
Includes NavigationStatus, MarkedCS, CS, Integer
Pre \( \forall cl: Class, \)
\[
[\neg \# \text{validClass (}
\qquad \# \text{markedCStoCS (} \# \text{getVisitedClasses (ns\textsuperscript{↑})},
\qquad \# \text{concat (} \# \text{getCurrentClass (ns\textsuperscript{↑}), cl})))\]
\]

Changes nothing

Post \( \text{signal ("YOU ARE AT A LEAF CLASS OF THE CLASSIFICATION SCHEME"')} \)

Pre \( \forall cl: Class, \)
\[
[\# \text{size (cl)} = \# \text{getCurrentClass (ns\textsuperscript{↑})} + 1 \land
\qquad \# \text{validClass (} \# \text{markedCStoCS (} \# \text{getVisitedClasses (ns\textsuperscript{↑})}, cl) \land
\qquad \# \text{isMarked (} \# \text{getVisitedClasses (ns\textsuperscript{↑}), cl})\]
\]

Changes nothing

Post \( \text{signal ("ALL CLASSES AT THE NEXT LOWER LEVEL HAVE BEEN VISITED")} \)

Pre \( \exists cl: Class, \)
\[
[\# \text{size (cl)} = \# \text{getCurrentClass (ns\textsuperscript{↑})} + 1 \land
\qquad \# \text{validClass (} \# \text{markedCStoCS (} \# \text{getVisitedClasses (ns\textsuperscript{↑})}, cl) \land
\qquad \neg \# \text{isMarked (} \# \text{getVisitedClasses (ns\textsuperscript{↑}), cl})\]
\]

Changes ns

Post \( \text{ns\textsuperscript{↑} = \# createNavigationStatus (cl, \# markClass (ns\textsuperscript{↑}, cl)} \)

---

Routine NEXT
Input \( \text{obj ns: NavigationStatus} \)
Returns nothing
Includes NavigationStatus

Pre \( \forall cl: Class, \)
\[
[\# \text{size (cl)} = \# \text{getCurrentClass (ns\textsuperscript{↑})} \land
\qquad \# \text{validClass (} \# \text{markedCStoCS (} \# \text{getVisitedClasses (ns\textsuperscript{↑})}, cl) \land
\qquad \# \text{isMarked (} \# \text{getVisitedClasses (ns\textsuperscript{↑}), cl})\]
\]

Changes nothing
Post signal("ALL CLASSES AT THE CURRENT LEVEL HAVE BEEN VISITED")

Pre \exists \text{cl: Class},

\[ \#\text{size(cl)} = \#\text{getCurrentClass(ns)} \land \\
\#\text{validClass(\#markedCStoCS(\#getVisitedClasses(ns)), cl)} \land \\
\neg \#\text{isMarked(\#getVisitedClasses(ns), cl)} \]\n
Changes ns
Post ns = = \#\text{createNavigationStatus(cl, \#markClass(ns', cl))}

---

Routine JUMP
Input obj ns: NavigationStatus, cl: Class
Returns nothing
Includes NavigationStatus, MarkedCS, CS

Pre \neg \#\text{validClass(\#markedCStoCS(\#getVisitedClasses(ns)), cl)}
Changes nothing
Post signal("CLASS DOES NOT EXIST IN THE CLASSIFICATION SCHEME")

Pre \#\text{validClass(\#markedCStoCS(\#getVisitedClasses(ns)), cl)}
Changes ns
Post ns = = \#\text{createNavigationStatus(cl, \#markClass(ns', cl))}
5. Summary and Conclusions

In this chapter we summarize the design of our library, discuss how we can implement our library on a relational data base, present some conclusions based on our experience from writing a reasonably large specification, and finally, suggest some areas for future work.

5.1 Summary of the Design

We summarize the design in three parts. In the first part we deal with the structure of the library. In the second and the third parts we list the operations for maintaining the components of a library, and the operations for browsing through the components of a library.

5.1.1 Structure

A library has two kinds components: specifications and classes. The classes form a hierarchical structure for classifying the specifications in the library. We refer to this hierarchical structure as the Classification Scheme.

The structure of a library can be represented as three directed graphs, only one of which can have cycles. Before we talk about the three graph structures, let us define some graph terminology.

Let graph $G = (T, R)$, where $T$ is set of nodes in the graph, and $R$ is a set of ordered pairs of nodes. We shall refer to an ordered pair of nodes as an arc, and denote an arc as $(\text{node}_1, \text{node}_2)$.

Let $G_1 = (T_1, R_1), G_2 = (T_2, R_2)$, and $G_3 = (T_3, R_3)$ be the three graphs that represent the structure of a library, $L$. 
1. G₁ is a tree structure that represents the classification scheme of L. For each class in L, T₁ contains a class node, and if class C₁ is the parent class of C₂ in the classification scheme, then R₁ contains (C₁, C₂).

2. For each specification, S, (or class, C,) in L, T₂ contains a node. If a specification, S, is classified under class, C, then R₂ contains (S, C). A specification can be classified under more than one class, and a class can have more than one specification classified under it.

3. For each specification, S, in L, T₃ contains a node. If an import name of specification S₁ is bound to specification S₂ then R₃ contains (S₁, S₂). Since import name of a specification, S, can be potentially bound to S itself, G₃ can be cyclic.

5.1.2 Maintenance Operations

These operations are called maintenance operations because they help a user in maintaining a library. Below we list these operations, and explain their effect on the structure of the library. Let L be a library, and G₁ = (T₁, R₁), G₂ = (T₂, R₂), and G₃ = (T₃, R₃) be the three graphs that represents its structure.

1. INSTALL: Installing a specification into L corresponds to adding a specification node to T₂ and T₃.

2. EXPUNGE: Expunging a specification from L corresponds removing a specification node from T₂ and T₃.

3. EXTEND: Extending a class in L by adding a sub class corresponds to adding a class node to T₁ and T₂, and an arc to R₁.

4. REMOVE: Removing a part of the classification scheme from L corresponds to removing class nodes from T₁ and T₂, and arcs from R₁.

5. CLASSIFY: Classifying a specification under a set of classes corresponds to adding a set of arcs to R₂.

6. UNCLASSIFY: Unclassifying a specification from a set of classes corresponds to removing a set of arcs from R₂.
7. **BIND**: Binding the import name of a specification corresponds to adding an arc to \( R_3 \).

8. **UNBIND**: Unbinding the import name of a specification corresponds to removing an arc from \( R_3 \).

In addition to the above operations that change the structure of the graph, there are two more operations for maintaining the components of a library.

1. **COPY**: Copying a specification is useful for copying a specification from one library into another. Copying a specification from a library \( L_1 \) to a library \( L_2 \) corresponds to adding a specification node to the structure of \( L_2 \). The structure of \( L_1 \) is unaffected.

2. **REPLACE**: Replacing a specification allows the contents of a specification to be replaced. This operation has no effect on the structure of the library.

Note that removing a class from a library can potentially cause specifications to be classified under a nonexistent class, and and expunging a specification may cause an import name to be bound to a nonexistent specification. The REMOVE operation (which removes a sub classification scheme) requires that there should be no specifications classified under any of classes being removed. The EXPUNGE operation requires that the specification being expunged has no import names (of specifications in the library) bound to it. The classification of all specifications in a library is always valid. Stated formally,

\[
\forall l: \text{Library}, s: \text{LibSpec}, c: \text{Class},
\quad [(s \in l \land c \notin \#\text{getClassification}(s)) \Rightarrow \\
\quad \#\text{validClass}(\#\text{getCS}(l), c)]
\]

Also, if an import name of a specification is bound, it is always bound a valid specification in the library. Stated formally,

\[
\forall l: \text{Library}, s_1, s_2: \text{LibSpec}
\quad [(s_1 \in l \land s_2 \in \#\text{getBindings}(s_1)) \Rightarrow s_2 \in l]
\]
5.1.3 Browsing Operations

The main purpose of a library is to serve as a repository where a user can hope to find previously developed specifications that will be useful in developing new ones. The term browsing refers to the process of finding "useful" specifications in a library. We introduce two kinds of objects to help a user in browsing through a library: a Catalog and a NavigationStatus.

A catalog serves as a repository for temporarily storing a set of specifications. We view browsing as an activity in which a user is concerned with the creation of catalogs, and examination of the specifications in the catalogs. The operations on a catalog are:

1. COMPILe: For creating a catalog of specifications from a library such that all the specifications in the catalog satisfy a boolean proposition.

2. FILTER: For creating a new catalog from a catalog such that all the specifications in the new catalog satisfy a boolean proposition.

3. SELECT: For selecting a specification in a catalog.

4. RETAIN: For creating a new catalog containing the selected specifications from a catalog.

5. EXCLUDE: For creating a new catalog containing the unselected specifications in a catalog.

6. MERGE: For creating a catalog containing the union of the specifications in two catalogs.

7. DIFFERENCE: For creating a catalog containing the difference of the specifications between two catalogs.

8. DISPLAY: For displaying the specifications in a catalog. The particular components (of the specifications) that should be displayed, and the order in which the specifications should be displayed, can be specified as arguments to the display operation.
A user can examine the specifications in a library in a systematic fashion using the CS of the library as the roadmap. A NavigationStatus serves the purpose of a roadmap by keeping track of (i) the class whose specifications the user is currently examining, and (ii) all those classes whose specifications the user has already examined. We refer to the class whose specifications the user is currently examining as the current class. Operations for navigating through a CS include:

1. **BEGIN**: For beginning the navigation of a CS.

2. **UP**: For defining the parent class of the current class as the new current class. This operation requires that the user has not "visited" the parent class.

3. **DOWN**: For defining an "unvisited" sub class of the current class as the new current class.

4. **NEXT**: For defining an "unvisited" class at the same level as the current class as the new current class.

5. **JUMP**: For defining a particular class in the CS as the current class. Unlike operations UP, DOWN and NEXT, which always define a class that the user has not "visited" as the current class, JUMP can be used to define a class that a user has already "visited" as the current class.

### 5.2 Building our Library on a Relational Data Base

In a relational model, a data base is viewed as a set of relations, where each relation may be viewed as a two dimensional table having an arbitrary number of rows (tuples) with a fixed number of columns (attributes), each of which has a value from a specified domain. Each tuple is unique; the uniqueness is determined by a subset of the attributes known as the key attribute. There is no other structure in a relational data base. Operations available on relations are the set theoretic operations.
One way of storing data objects having pointers to other data objects in a relational data base is by storing the key attribute of the second object as one of the attributes of the first object. Using the above idea, below we give a possible way in which we could implement our library as a relational data base.

The relations in the data base are divided into four categories:

1. The first category contains just one relation. The number of tuples in this relation is equal to the number of specifications in the library with each tuple containing the information about the components of a specification.

2. The second category also contains just one relation containing information about the structure of the CS. This relation contains a tuple for every class in the CS of the library. Each tuple has a key attribute that contains the name of the class, and another attribute that contains the name of the class's parent.

3. The third category contains a relation for each class in the library’s CS and contains the information about the specifications classified under each class. The tuple for a class C simply contains the key attribute of all the specifications that are classified under C.

4. The relations in the fourth category contain the information about the pointers between specifications. For each specification in the library there is a corresponding relation in this category. The tuples in the relation for a specification, S, contain the key attributes of all the specifications to which S has a pointer. One might wonder why we cannot include this information about the pointers as one of the attributes in the same relation that contains the components of a specification. We cannot do so because the number of specifications to which a given specification can have pointers is not fixed, and the number of attributes in a relation are fixed.

5.3 Dynamic Growth of the CS in our Library

In this section we comment briefly on a fundamental difference between the CS in our specification library, and the CS in a conventional library of books. While our CS evolves continously, the CS of a library of books is fixed, and evolves much less over a period of time.
Let us first describe the structure of a CS of a library of books such as the Library of Congress CS. In developing the Library of Congress CS, a lot of time and effort was spent making its structure balanced. Not only the structure is balanced, it is designed in such a way that as new subjects come into existence and are added to the Library of Congress CS, its structure remains "reasonably" balanced.

In contrast, the CS of our specification library evolves continuously in a manner that is specified by a user, not predictable in advance, and liable to become unbalanced as new specifications are added to a library. As specifications are added to a library, new classes might have to be defined to classify them. Certain classes may no longer contain a small number of classes; in such cases it is desirable to partition the classes into smaller classes. Finally, addition and/or deletion of classes may skew the tree structure of our CS, and thus our CS required to be re-balanced from time to time.

5.4 Computer Support for Writing Specifications

The specification of our library includes 33 traits and 24 routines. To create a well formed specification that correctly describe the requirements of a system without a computer support is roughly equivalent to writing a bug free program containing 33 type abstractions and 24 procedure abstractions without any syntax errors at the first try. We spent a large number of man hours (it is difficult to give a realistic estimate) in trying to remove syntax errors, and maintaining consistency between the operator definitions and operator usage.

A specification language based editor (or at least a syntax checker) would be extremely useful in writing specifications. In addition, a theorem prover would be useful for checking specifications for consistency and completeness properties. A theorem prover can also be useful for asking questions about the formal specifications [14].
The creation of such a system is the primary goal of the Systematic Program Development Group at Laboratory for Computer Science, MIT [15, 27]. A syntax directed editor for the specification language has been designed and is in the process of being implemented [27]. The design of a theorem prover is under progress.

5.5 Writing A Correct Specification is not Good Enough

We claim it is not more difficult to write a "correct" specification than it is to write a "correct" program. In fact, we feel the effort required for an interested programmer to learn our specification language is less than the effort required to learn a new programming language. The syntax of the language is simple, and the number of concepts are fewer than most programming languages.

However, based on our experience, we feel that it is not enough to write a specification that "correctly" describes the properties of a system; it is equally important (if not more important) to write specifications that are also easy to understand. A specification that is difficult to understand is almost as bad as not having a specification. (Programs that are very hard to read (and understand) are still useful as long as they work "correctly.") We feel, developing a large size specification, which is both correct and readable, requires multiple iterations over the specification, possibly trying out alternative ways to decompose the specification into abstractions.

In the process of developing our specifications, we had to constantly make modifications to the specifications to make them more readable. Often, we had to do "fine tuning" by adding and deleting operators to traits to make the specifications more readable. Often decisions about when to introduce a new abstraction, and when to merge abstractions into a single one were taken simply to make the specifications more readable. In section 4.3 we discuss an instance
when we avoided introducing a new level of abstraction simply to improve readability.

5.6 Areas for Future Work

In this section we suggest some useful ways in which the design of our library can be extended.

1. Interesting Propositions for Accessing Specifications

We feel it would be useful to define propositions on the axioms and operators of the traits. For example, a user might want to compile a catalog of all traits in a library that are essentially a collection of index-value pairs. That is, the user might give two operators,

```plaintext
# addEntry: Table X Index X Value --> Table
# getValue: Table X Index --> Value
```

and, an axiom

```plaintext
# getValue (# addEntry (t, i, v), i1) = =
    if i = i1
    then v
    else # getValue (t, i1)
```

and, ask the library to compile a catalog of all traits that have two operators and an axiom isomorphic to the given operators and axiom.

In general, propositions on the operator and axioms of a specification might require the assistance of a theorem prover. For example, the user might want to compile a catalog of all traits that have a operator which defines a complete ordering of the values of a sort. That is, an operator definition syntactically isomorphic to

```plaintext
# op: T X T --> Boolean
```

is contained in the trait, and the proposition

```plaintext
# op (t1, t2) => ¬ # op (t2, t1)
```

is included in the theory of the trait. Defining the above kinds of propositions on the operators and axioms of a trait require a knowledge of the theorem proving aspects of our specification language, and is an interesting area for future work.
2. More Complicated Ordering Relations for Sorting a Catalog

In our design a user can order the specifications in a catalog in three predefined ways. A useful extension to the design might be to support user defined ordering relations. For example, allow a user to sort a catalog alphabetically by the author of the specifications, and sort the specifications with the same author by the timestamp of the specifications.

3. Macro Facility

As a user continues to use the library, it is inevitable that he will discover a sequence of operations that he uses frequently. He might want to define an "arsenal" of commonly used sequence of operations (macros.)

4. Save Facility

Often a user might want to freeze the status of his browsing session and later pick up from where he left off. In such situations an operation that saved the state of the library would be useful.

5. Default Facility

A facility for defining defaults for the arguments to various the operations would be useful.
References


15. J. V. Guttag, A Specification Method and Environment (A Research Proposal)," Available as Internal Memo, SPDG Group, Laboratory for Computer Science, M.I.T.


17. J. V. Guttag, "An Introduction to Bicycle Trait Language," Available as Internal Memo, SPDG Group, Laboratory for Computer Science, M.I.T.


Appendix I - Trait Integer

trait SimpleInteger

there exists functions

0: \rightarrow \text{Integer}

# next: \text{Integer} \rightarrow \text{Integer}
# succ: \text{Integer} \rightarrow \text{Integer}
# pred: \text{Integer} \rightarrow \text{Integer}
# nonNeg: \text{Integer} \rightarrow \text{Boolean}
# eq: \text{Integer} \times \text{Integer} \rightarrow \text{Integer}

constrains [\text{Integer}] so that

# succ (0) = # next (# next (0))
# succ (# next (0)) = 0
# succ (# next (# next (x))) =
  if nonNeg(x)
    then # next (# next (# next (# next (x)))))
    else x

# pred (0) = # next (0)
# pred (# next (0)) = # next (# next (# next (0)))
# pred (# next (# next (x))) =
  if nonNeg(x)
    then x
    else # next (# next (# next (# next (x))))

# eq (0,0) = true
# eq (# next (x), 0) = false
# eq (0, # next (x)) = false
# eq (# next (x), # next (y)) = # eq (x, y)

# nonNeg (0) = true
# nonNeg (# next (0)) = false
# nonNeg (# next (# next (x))) = # nonNeg (x)

end SimpleInteger
trait Integer

include SimpleInteger

# gt:  Integer X Integer --> Boolean
# lt:  Integer X Integer --> Boolean
# ge:  Integer X Integer --> Boolean
# le:  Integer X Integer --> Boolean
# ne:  Integer X Integer --> Boolean

constrains [Integer] so that

# gt (0, 0) = = false
# gt (# next (x), 0) = = ¬ # nonNeg (x)
# gt (0, # next (x)) = = # nonNeg (x)
# gt (# next (x), # next (y)) = =
    if nonNeg (x) ∧ ¬ # nonNeg (y)
    then false
    else if ¬ # nonNeg(x) ∧ # nonNeg (x)
    then true
    else ¬ # gt (x, y)

# lt (x, y) = = ¬ (# gt (x, y) ∨ # eq (x, y))

# ge (x, y) = = # gt (x, y) ∨ # eq (x, y)

# le (x, y) = = # lt (x, y) ∨ # eq (x, y)

# ne (x, y) = = ¬ # eq (x, y)

end Integer
Appendix II - Trait String

trait String

include Char, Integer

there exist functions

# new: → String
# insert: String X Char → String
# first: String → Char
# size: String → Integer
# concat: String X String → String
# same: String X String → Boolean
# lessEq: String X String → Boolean

constrains [String] so that

% # first (# new) is intentionally not defined %
# first (# insert (s, c)) = = c

# rest (# new) = = # new
# rest (# insert (s, c)) = = s

# size (# new) = = 0
# size (# insert (s, c)) = = # size (s) + 1

# concat (# new, s) = = s
# concat (# insert (s, c), s1) = = # insert (# concat (s, s1), c1)

# same (# new, # new) = = true
# same (# new, # insert (s, c)) = = false
# same (# insert (s, c), # new) = = false
# same (# insert (s, c), # insert (s1, c1)) = =
  if c = = c1
    then # same (s, s1)
  else false

# lessEq (# new, # new) = = true
# lessEq (# new, # insert (s, c)) = = true
# lessEq (# insert (s, c), # new) = = false
# lessEq (# insert (s, c), # insert (s1, c1)) = =
  if c < c1
    then true
    else c = = c1
  else false

end String
## Appendix III - Mapping Frequently Used Operators Id’s to Traits

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait(page-no.)</th>
</tr>
</thead>
<tbody>
<tr>
<td># addBinding</td>
<td>TableOfBindings X ImportName X LibSpec</td>
<td>TableOfBindings</td>
<td>TableOfBindings(46)</td>
</tr>
<tr>
<td># addCS</td>
<td>Library X CS</td>
<td>Library</td>
<td>Library(47)</td>
</tr>
<tr>
<td># addLibSpec</td>
<td>Library X LibSpec</td>
<td>Library</td>
<td>Library(47)</td>
</tr>
<tr>
<td># addSubCS</td>
<td>CS X Class X CS</td>
<td>CS</td>
<td>CS(47)</td>
</tr>
<tr>
<td># append</td>
<td>Class X ClassName</td>
<td>Class</td>
<td>Class(44)</td>
</tr>
<tr>
<td># bind</td>
<td>LibSpec X TableOfBindings</td>
<td>LibSpec</td>
<td>LibSpec(45)</td>
</tr>
<tr>
<td># buildDisplayCatalog</td>
<td>OrderedCatalog X Selections</td>
<td>DisplayCatalog</td>
<td>DisplayCatalog(82)</td>
</tr>
<tr>
<td># classify</td>
<td>LibSpec X SetOfClasses</td>
<td>LibSpec</td>
<td>LibSpec(45)</td>
</tr>
<tr>
<td># concat</td>
<td>Class X Class</td>
<td>Class</td>
<td>Class(44)</td>
</tr>
<tr>
<td># containsLibSpec</td>
<td>TableOfBindings X LibSpec</td>
<td>Boolean</td>
<td>TableOfBindings(46)</td>
</tr>
<tr>
<td># csExists</td>
<td>Library</td>
<td>Boolean</td>
<td>Library(47)</td>
</tr>
<tr>
<td># CstoMarkedCS</td>
<td>CS</td>
<td>MarkedCS</td>
<td>MarkedCS(86)</td>
</tr>
<tr>
<td># deleteBinding</td>
<td>TableOfBindings X ImportName</td>
<td>TableOfBindings</td>
<td>TableOfBindings(46)</td>
</tr>
<tr>
<td># deleteCS</td>
<td>Library</td>
<td>Library</td>
<td>Library(47)</td>
</tr>
<tr>
<td># deleteSubCS</td>
<td>CS X Class</td>
<td>CS</td>
<td>CS(47)</td>
</tr>
<tr>
<td># first</td>
<td>Class</td>
<td>Class</td>
<td>Class(44)</td>
</tr>
<tr>
<td># getBindings</td>
<td>LibSpec</td>
<td>TableOfBindings</td>
<td>LibSpec(45)</td>
</tr>
<tr>
<td># getClassification</td>
<td>LibSpec</td>
<td>SetOfClasses</td>
<td>LibSpec(45)</td>
</tr>
<tr>
<td># getCS</td>
<td>Library</td>
<td>Class</td>
<td>Library(47)</td>
</tr>
<tr>
<td># getCurrentClass</td>
<td>NavigationStatus</td>
<td>Class</td>
<td>NavigationStatus(87)</td>
</tr>
<tr>
<td># getLibSpec</td>
<td>TableOfBindings X ImportName</td>
<td>LibSpec</td>
<td>TableOfBindings(46)</td>
</tr>
<tr>
<td># getRootClass</td>
<td>CS</td>
<td>ClassName</td>
<td>CS(47)</td>
</tr>
<tr>
<td># getSubCS</td>
<td>CS X Class</td>
<td>CS</td>
<td>CS(47)</td>
</tr>
<tr>
<td># getVisitedClasses</td>
<td>NavigationStatus</td>
<td>MarkedCS</td>
<td>NavigationStatus(87)</td>
</tr>
<tr>
<td># isATrait</td>
<td>InputSpec</td>
<td>Boolean</td>
<td>LibSpec(45)</td>
</tr>
<tr>
<td># isMarked</td>
<td>MarkedCS X Class</td>
<td>Boolean</td>
<td>MarkedCS(86)</td>
</tr>
<tr>
<td># isSorted</td>
<td>OrderedCatalog X SortFunction</td>
<td>Boolean</td>
<td>Sort(83)</td>
</tr>
<tr>
<td># isSelected</td>
<td>Catalog X LibSpec</td>
<td>Boolean</td>
<td>Catalog(64)</td>
</tr>
<tr>
<td># makeCS</td>
<td>ClassName</td>
<td>CS</td>
<td>CS(47)</td>
</tr>
<tr>
<td># markClass</td>
<td>MarkedCS X Class</td>
<td>MarkedCS</td>
<td>MarkedCS(86)</td>
</tr>
<tr>
<td># markedCstoCS</td>
<td>MarkedCS</td>
<td>CS</td>
<td>MarkedCS(86)</td>
</tr>
<tr>
<td># parent</td>
<td>Class</td>
<td>Class</td>
<td>Class(44)</td>
</tr>
<tr>
<td># removeLibSpec</td>
<td>Library X LibSpec</td>
<td>Library</td>
<td>Library(47)</td>
</tr>
<tr>
<td># rest</td>
<td>Class</td>
<td>Class</td>
<td>Class(44)</td>
</tr>
<tr>
<td># satisfied</td>
<td>Proposition X LibSpec</td>
<td>Boolean</td>
<td>Proposition(70)</td>
</tr>
<tr>
<td># same</td>
<td>Class X Class</td>
<td>Boolean</td>
<td>Class(44)</td>
</tr>
<tr>
<td># size</td>
<td>Class</td>
<td>Integer</td>
<td>Class(44)</td>
</tr>
<tr>
<td># validClass</td>
<td>CS X Class</td>
<td>Boolean</td>
<td>CS(47)</td>
</tr>
</tbody>
</table>
### Appendix IV - A List of Overloaded Operators

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td># containsImportName</td>
<td>TableOfBindings X ImportName</td>
<td>Boolean</td>
<td>TableOfBindings</td>
</tr>
<tr>
<td># containsLibSpec</td>
<td>Library X LibSpec</td>
<td>Boolean</td>
<td>Library</td>
</tr>
<tr>
<td># has</td>
<td>SetOfClasses X Class</td>
<td>Boolean</td>
<td>SetOfClasses</td>
</tr>
<tr>
<td># has</td>
<td>SetOfImportNames X ImportName</td>
<td>Boolean</td>
<td>SetOfImportNames</td>
</tr>
<tr>
<td># hasLibSpec</td>
<td>Catalog X LibSpec</td>
<td>Boolean</td>
<td>Catalog</td>
</tr>
</tbody>
</table>

#### Infix Operator ≤

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td># le</td>
<td>Integer X Integer</td>
<td>Boolean</td>
<td>Integer</td>
</tr>
<tr>
<td># le</td>
<td>TimeStamp X TimeStamp</td>
<td>Boolean</td>
<td>TimeStamp</td>
</tr>
<tr>
<td># lessEq</td>
<td>String X String</td>
<td>Boolean</td>
<td>String</td>
</tr>
<tr>
<td># lessEq</td>
<td>Author X Author</td>
<td>Boolean</td>
<td>Author</td>
</tr>
<tr>
<td># lessEq</td>
<td>SpecName X SpecName</td>
<td>Boolean</td>
<td>SpecName</td>
</tr>
</tbody>
</table>

#### Infix Operator ≥

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td># gt</td>
<td>Integer X Integer</td>
<td>Boolean</td>
<td>SimpleInteger</td>
</tr>
<tr>
<td># gt</td>
<td>TimeStamp X TimeStamp</td>
<td>Boolean</td>
<td>TimeStamp</td>
</tr>
</tbody>
</table>

#### Infix Operator <

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td># lt</td>
<td>integer X Integer</td>
<td>Boolean</td>
<td>Integer</td>
</tr>
<tr>
<td># lt</td>
<td>TimeStamp X TimeStamp</td>
<td>Boolean</td>
<td>TimeStamp</td>
</tr>
</tbody>
</table>

#### Infix Operator >

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td># ge</td>
<td>Integer X Integer</td>
<td>Boolean</td>
<td>Integer</td>
</tr>
<tr>
<td># ge</td>
<td>TimeStamp X TimeStamp</td>
<td>Boolean</td>
<td>TimeStamp</td>
</tr>
</tbody>
</table>

#### Infix Operators ∪ and –

<table>
<thead>
<tr>
<th>Opld</th>
<th>Domain</th>
<th>Range</th>
<th>Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td># union</td>
<td>SetOfClasses X SetOfClasses</td>
<td>SetOfClasses</td>
<td>SetOfClasses</td>
</tr>
<tr>
<td># difference</td>
<td>SetOfClasses X SetOfClasses</td>
<td>SetOfClasses</td>
<td>SetOfClasses</td>
</tr>
</tbody>
</table>