A Formal Specification Primer --
A Companion Document to the SpecL Reference Manual
Version 4, September 2006

Contents

1. Introduction

This primer presents a detailed example of formal software specification. The example is a simple electronic Rolodex system that stores and retrieves information records. The system performs functions that are common to a variety of other information processing applications. Hence, the specification techniques used in the example are applicable in general to other software applications.

The example begins with the definition of a concrete user interface. The elements of the interface are used in the first step of the formal specification process -- the identification of the system's objects and operations. Once the objects and operations are defined, the specification is formalized by adding mathematical logic that precisely defines system requirements and constraints.

1.1. Motivation

The major purpose of this primer is to portray formal specification as a *practical* tool. Far too many software engineers view formal mathematics as tedious and largely irrelevant to their activities. This is a rather unusual view when one compares software engineering to other science and engineering disciplines. In almost all such disciplines, mathematical reasoning is an everyday practice.

One may argue that software engineering is more like a construction project than science or "real" engineering. Even if this were the case, the use of mathematics would be no less important. No construction firm would dream of undertaking a major building task without complete specifications, including the necessary engineering calculations. Such calculations rely heavily on mathematical analysis.

The branch of mathematics that is most relevant to software engineering is *logic*. Both the specification and implementation of software can be defined in logical terms. Hence, mastery of mathematical logic should be as important to the software engineer as mastery of mathematical analysis is to the civil engineer.

One cause for lack of mathematical rigor in software engineering is that mathematics can be *hard*. When given the chance, human nature will steer us away from hard tasks. For example, the civil engineer might well prefer to draw some simple pictures and perform some informal analysis when designing a bridge. Such would be the kind of analysis that is frequently considered adequate for software engineering projects. Fortunately, the competent civil engineer knows that informal analysis is not sufficient, and that the bridge may well collapse if a careful mathematical analysis is not performed. The civil engineer learns this as part of basic training, and the practice of civil engineering requires that mathematical analysis is an integral part of the job.

Like the civil engineer, the software engineer must learn that careful mathematical reasoning is necessary to keep programs from collapsing. Unfortunately, many software engineers are not trained this way, nor does the practice of software engineering require the same degree of mathematical rigor as is required for other branches of engineering. As software engineering matures into a genuine engineering discipline, the acceptance and use of formal mathematics will be an important part of its maturation.

1.2. Notation

The example in this primer is given in a formal specification language called SpecL. This primer does not present a full definition of SpecL. Rather, necessary notation is introduced as the Rolodex example evolves. Complete details of SpecL are presented in the SpecL Reference Manual, Version 4.22, August 2006.

This primer also does not provide an introduction to the fundamentals of mathematical logic. As with the notation of SpecL, the notations of predicate logic are introduced as needed in the example. The logic used in the primer is typically covered in a standard first course on discrete mathematics, such as CSC 245.

1.3. Scope of the Primer

The primary focus of this primer is *specification*, and the secondary focus is *requirements analysis*. Requirements analysis entails:

- determining what end users want and need from a computer system;
- developing a prototype user interface and scenarios of system usage, to help elucidate end-user requirements;
- developing user-level documentation that describes system functionality and requirements in terms understandable to the end user (e.g., a users manual).

Specification entails

- defining system functionality in a formal language;
- encoding requirements in formal logic;
- iterating with the requirements analysis process as necessary to ensure that the user-level view and formal specification are consistent.

In the overall context of software engineering, the interaction between requirements analysis and formal specification is quite important. As the specification is formalized, the specifier will typically discover changes that must be made to the user interface in order maintain consistency. Complementarily, user-initiated changes to requirements will drive changes in the specification. In this way, user-level analysis and formal specification proceed in tandem, resulting eventually in a complete and consistent definition of a system.

In this primer, the interaction between requirements analysis and specification is abbreviated in order to expedite the presentation of formal specification. In practice, such abbreviation should not take place. Thorough consultation with end users should always be an integral part of specification development.

1.4. What Is a “Requirement”, What Is a “Specification”?

There are nearly as many definitions of the terms "requirement" and "specification" as there are authors who use them. A common misconception is that a requirement is an informal statement of user need and a specification is a (more) formal statement of system functionality.

In order for a system to be formally specified, both requirements and specifications need to be formally defined. Furthermore, in order for a system to be understandable to an end user, both requirements and specifications need to be presented in informal, user-accessible terms. Hence, a complete requirements/specification document consists of two views -- a formal system view, and an informal user view.

Given these observations, the terms in question can be defined precisely as follows:

- A *specification* defines the functionality of a system, in terms of the objects and operations of which the system is composed.
- A *requirement* is a verifiable statement of fact made about an object or operation.

This primer uses the term "requirements specification" to refer collectively to both parts of this definition.

2. The Rolodex User Interface

The top-level user interface to the sample Rolodex system is shown in Figure 1. It is a familiar menu-style interface, common to many types of application program. The File menu contains commands to create a new Rolodex file, open an existing file, save the current working file, save the current working file under a new name, print the current working file, and exit. The Edit menu contains commands to undo/redo the previous command, and cut/copy/paste text. Finally, the Rolodex menus has commands to add a new entry into the Rolodex, delete an entry, change an entry, or find an entry.

In a typical scenario of use, the user will open a new Rolodex and proceed to add new entries. Subsequently, entries will be changed, deleted, and searched for as necessary. In response to each of the Rolodex menu commands, an appropriate data-entry dialog box is displayed. For example, a sample dialog for the Add command is shown in Figure 2. In this dialog, the user types the required information and completes the Add operation by pressing the OK

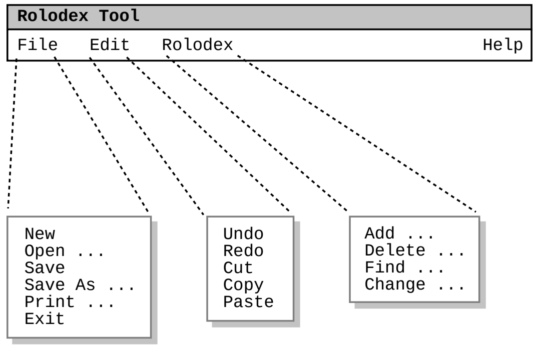

Figure 1: Top-Level Rolodex Interface.

button. The Clear button clears all of the typed information, leaving the dialog empty. The Cancel button cancels the Add operation, and removes the dialog from the display.

For the initial version of the interface, we will assume that Rolodex cards are accessed by name. Accordingly, the commands to delete, change, and find a card use the dialogs shown in Figure 3. For each of these commands, the user initially enters a name, to which the system responds with a further command-specific dialog. The system response to Delete is a dialog that indicates whether a card of the given name is found, and asks for confirmation to delete it. The response to Change is the same data-entry dialog used for Add (Figure 2). Lastly, the response to Find is a display of the card if one of the given name is found, or a "not found" message otherwise.

With this basic interface description, we will proceed to formalize the specification. Even though there are a number of user-level requirements that remain to be addressed. For example, should duplicate cards be allowed in the Rolodex? If so, how should the access commands respond? Developing an initial formal representation will help us address such requirements precisely. After the basic requirements have been covered, we will consider system enhancements and how the enhancements can be precisely specified.

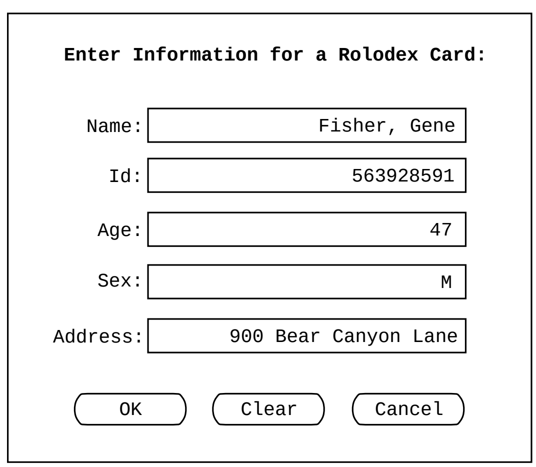

Figure 2: Dialog for Adding a Rolodex Card.

3. Defining Objects and Operations

The initial step in formalizing a specification is to identify the *objects* and *operations* of the system. Almost all Software Engineering textbooks discuss this process in one form or another. It is sometimes called "domain analysis" or "domain modeling". In general, the terms "object" and "operation" are quite commonly used in the same sense as they are used here.

Textbooks describe a number of methods for object and operation identification. Diagraming techniques, such as dataflow and entity-relationship modeling are popular. Another general approach is to apply heuristics that transform a prose description of the system into a more formal notation. This approach begins by deriving objects from nouns or noun phrases and operations from verbs or verb phrases.

The method used in this primer is to derive objects and operations from a prototype user interface. This technique has the advantage of using a necessary artifact of the requirements analysis process, without requiring extra diagrams or prose descriptions to be developed. Another advantage is that it provides the basis for automatically generating portions of a specification from an interface, and for verifying that the interface is consistent with the specification.

3.1. Interface Heuristics

The following heuristics can be used to derive an initial set of objects and operations from a graphical user interface:

1. Function buttons and menu items generally correspond to operations.
2. Data-entry screens and output screens generally correspond to objects.
3. More specifically, data-entry dialogs that appear in response to invoking an operation generally correspond to the input object(s) for the invoked operation.
4. Output reporting screens that appear in response to confirming an input dialog (e.g., with an "OK" button) generally correspond to the output object(s) for the confirmed operation.
5. Interface elements that allow entry of a single number, string, or boolean value correspond to atomic objects.
6. The hierarchical structure of objects is generally displayed in the interface by nested or cascading windows and boxes, with atomic elements at the lowest level of nesting.

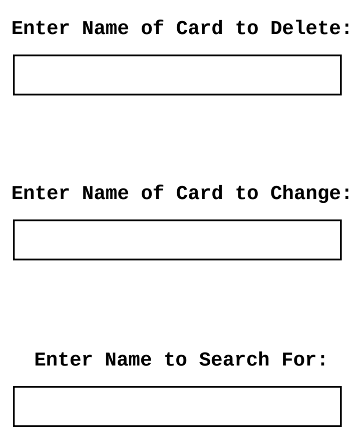

Figure 3: Dialogs for Finding, Changing, or Deleting a Card.

3.2. Heuristic Application

Applying these heuristics to the preceding interface example, we can develop the objects and operations for the Rolodex system. In particular, by the first heuristic we can identify the following operations from the Rolodex menu in Figure 1:

```
operation Add;
operation Delete;
operation Change;
operation Find;
```

For the moment, we will focus on these operations from the Rolodex menu and not yet consider the operations on the File menu. The Rolodex operations are more central to the specific functionality of the Rolodex, whereas the File operations are more general operations that are common to many other applications.

From the second heuristic and Figure 2 of the interface, we can identify the following object:

```
object Card;
```

These basic object and operation definitions are the initial step in the formal specification. They are already specified in SpecL formal notation, in which `object` and `operation` are keywords, names are formal identifiers, and the semi-colon is used as punctuation in much the same way as in a Pascal-class programming language.

The next step is to refine the initial definitions. Specifically, for the operations we need to define the inputs and outputs. In SpecL, operation inputs and outputs are always the names of defined objects. Object refinement entails defining the composite structure of the object, if necessary.

Applying heuristics 3 through 6, we can refine the initial object and operation definitions as follows (see Figure 4):

```
operation Add(Card,...)->(...);
operation Delete(Name,...)->(...);
operation Change(Name,Card,...)->(...);
operation Find(Name,...)->(...);

object Card = Name and Id and Age and Sex and Address;
```

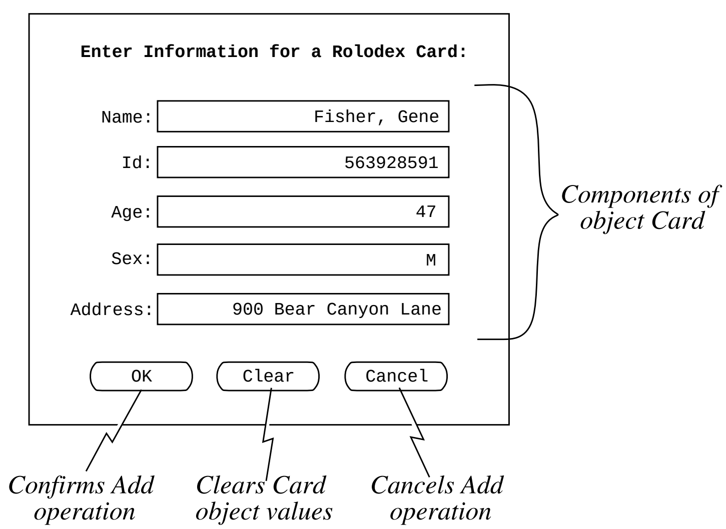

Figure 4: Identifying Object Components from the Interface.

```
object Name = string;
object Id = number;
object Age = number;
object Sex = Male or Female;
object Male;
object Female;
object Address = string;
```

Here we have introduced SpecL notations to specify operation inputs/outputs and object components. The general formats of these notations are the following:

```
operation name(inputs)->(outputs)

object name is component-expression
```

The ellipses in the operation definitions indicate that there are more inputs and outputs to be defined. We will discuss these shortly. Note that the three-dot ellipsis symbol is syntactically legal in SpecL, and may be used anywhere that an identifier is legal. The ellipsis symbol is useful to specify that something is formally missing, with the intention that it is to be expanded in the future.

As discussed in the SpecL reference manual, component expressions can be constructed from four atomic types and five composition operators. Table 1 summarizes these. The composition operators should be reasonably intuitive and are typically easy to recognize in an interface. The table notes common interface forms for each of the basic object types. See the SpecL reference manual for more detailed discussion of object composition.

3.3. Further Refinement

To this point, we have applied heuristics in a straightforward manner. Now we must address some technical details. In particular, we must consider the fact that SpecL is a *functional language*. What this means is that an operation can only operate upon objects that are specified explicitly as inputs and outputs. There are no global data in a functional specification. Programmers who are unfamiliar with functional languages may need some time to become comfortable with functional reasoning. As a concrete example, let us complete the input/output specification for the Add operation, and discuss precisely what it means.

```
operation Add(Rolodex, Card)->Rolodex;
```

This specifies `Add` with two inputs consisting of a Rolodex and a Card, and a single Rolodex for output. Most noteworthy is the introduction of the new object named Rolodex. We obviously need to discuss its origin.

SpecL Type	Meaning	Common Interface Form
integer	integer	string editor for numbers; numeric slider bar or dial
real	real	same as integer
string	string	string editor or combo box
boolean	true/false	string editor for true/false value; on/off button
and	tuple	box containing other types
or	union	radio buttons or string editor with restricted values
*	zero or more	scrollable list
->	function	push button or menu item

Table 1: Basic SpecL Types.

Any system that specifies database-like operations must have on object that is the database itself. Database-like operations are precisely those that we have defined for the Rolodex system: Add, Delete, Change, Find. From a functional perspective, these operations need a database object on which to operate. The structure of such an object is generally a collection of zero or more entries, which is represented in SpecL using the '*' composition operator. Hence, the formal definition of the Rolodex is

```
object Rolodex is Card*;
```

This specifies that Rolodex is a collection of zero or more Cards.

Generally, large collections do not appear in toto on any interface screen. For example, we do not expect the the entire contents of the Rolodex to be displayed on the screen at once. Rather, the point of the system is to store items and retrieve them individually by name. Given this, there is no obvious user interface heuristic to help identify database-like objects. Rather, we can postulate a system-level heuristic for this purpose. Viz., in a system that has database-like operations, a database object must be defined.

Given the Rolodex object, and the requirement that operations must be functional, here are the complete signatures[1] for the remaining Rolodex operations:

```
operation Delete(Rolodex,Name)->Rolodex;
operation Change(Rolodex,Name,Card)->Rolodex;
operation Find(Rolodex,Name)->Card;
```

Let us focus a bit more closely on the nature of a functional operation. The most important aspect is that a functional operation is *side effect free*. This means that an operation cannot use any object unless that object is an explicit input. Further, an operation may effect change only through an explicit output.

The notion that operations *effect change* rather than modify objects is also an important aspect of functional definition. An operation does not modify objects to produce output objects. Rather, a fully functional operation can only *create new objects.*

Consider the Add definition above. When this operation performs its function, it accepts Card and Rolodex as inputs. What it outputs is a *new copy* of the input Rolodex, with a *new copy* of the input card added into the database. In programming language terms, functional specifications have no global variables, no global files, and no call-by-reference parameters. In this sense, SpecL functional definitions are similar to definitions in functional programming languages such as pure LISP and ML.

The fully functional specification of operations is sometimes counter-intuitive, particularly in the case of large objects in a transaction-oriented system. For example, one might consider the explicit input and output of a large database to be unnecessary and/or inefficient. It is necessary since in order to construct a result that contains a new record, the original database must be input. It cannot be assumed that the operation will read from some stored database file or other external storage structure.

With regards to implementation efficiency, this matter is strictly not of concern in an abstract specification.[2] It is almost certainly not the case that an implementation would copy entire databases from input to output. However, such implementation concerns are beyond the scope of the specification. The specification states in functional terms what an operation does, including all inputs and outputs that it uses. A subsequent implementation can use whatever efficient techniques are available, as long as the implementation meets the abstract specification.

While there is some debate as to the utility of functional programming languages, there is little debate for specification languages. There is wide agreement that a formal specification language needs to be functional. This property is necessary to ensure that specifications are formally sound and verifiable. In particular, the predicative style of

---

[1] A *signature* is the specification of the input and output types of an operation.

[2] This is not to say that efficient system performance is to be ignored in a requirements specification. However, performance constraints should be specified in user-level terms, not by algorithmic specification.

specification presented in the next section of the primer relies fundamentally on the functional property of operations.

4. Formal Specification with Preconditions and Postconditions

Having completed the initial phase of specification, we are ready to formalize the object and operation definitions fully. The formal technique used in the primer is based on operation *preconditions* and *postconditions*. A precondition is a predicate (i.e., boolean-valued expression) that is true before an operation executes. A postcondition is a predicate that is true upon completion of an operation. Since pre- and postconditions are predicates, this style of formal specification often called *predicative*.

The pre- and postconditions are used to specify fully what the system does, including all user-level requirements for the system. In practice, formal specification is part of the overall process of requirements specification, which entails:

1. gathering user-level requirements via interface storyboards and usage scenarios;
2. identifying objects and operations;
3. formalizing with pre- and postconditions;
4. refining user-level requirements
5. refining object and operation definitions
6. iterating steps 3-5 until done.

The "until done" step involves two levels of validation. First, we must validate that the specified system is complete and consistent from the end user's perspective. That is, the system meets all end-user needs and does so in a way that is wholly satisfactory to the end user. This is accomplished by continued consultation with the end user, including user interaction with the system prototype.

The second level of validation involves completeness and consistency from a formal perspective. This can be accomplished in a number of ways. In the case of mechanized specification languages, such as SpecL, some completeness and consistency checking is done using a computer-based analyzer. Another valuable validation technique is peer review via formal walkthroughs. Also, there are techniques for formal specification testing, including the postulation and proof of *putative theorems*. Such theorems define properties of the system that we expect to be true, and which can be proved true formally with respect to the given system specification.

The formal specification examples presented in the primer have all been run through the SpecL checker, and are therefore complete to the extent guaranteed by that tool. Other forms of formal testing are beyond the scope of the primer.

4.1. Notational Summary

The specification examples to follow use the SpecL variant of formal logic. Available operations include predicate logic, arithmetic, lists, tuples, unions, and strings. These operations are summarized in Table 2. For complete details, the reader should consult the SpecL reference manual. The logic of SpecL is comparable to other formal specification languages. A slight difference between SpecL and a number of contemporary languages is the use in SpecL of lists instead of sets. Formally, both lists and sets can be fully axiomatized, so there no lack of formality in the use of lists. Overall, the use of lists instead of sets results in little difference in a specification. Set notation makes certain low-level specification easier than with lists, such as operations that can be modeled with set union and difference. On the other hand, list notation makes other forms of specification easier than with sets, such as specification of ordering constraints.

Predicate Logic:

Operator	Description
and	logical and
or	logical or
not	logical not
=>	logical implication
<=>	logical equivalence
if-then-else	conditional choice
forall	universal quantification
exists	existential quantification

Arithmetic:

Operator	Description
+	addition
-	subtraction
/	division
*	multiplication
#	length

Lists:

Operator	Description
[e1,...,en]	construction (elementwise)
[e1 .. en]	construction (inclusive range)
L[n]	selection (nth, from 1)
L[m:n]	selection (mth - nth)
+	concatenation
-	deletion
in	membership
#	length

Tuples:

Operator	Description
{e1,...,en}	construction
.	selection

Unions:

Operator	Description
.	selection
?	tag interrogation

Strings:

Operator	Description
"xxx"	construction
L[n]	selection (nth)
L[m:n]	selection (mth - nth)
+	concatenation
in	membership
#	length
explode	convert to list
implode	convert from list

Relational:

Operator	Description
=	equal
!=	not equal
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Table 2: SpecL Notation Summary.

4.2. Formal Specification Maxims

In developing any formal software specification, it is useful to observe the following two maxims:

1. Nothing is obvious.
2. Never trust the programmer.

The first maxim relates primarily to user-level requirements. It is often easy to think that a requirement is sufficiently obvious that it need not be stated formally. The problem with this thinking is that one person's obvious is not always the same as another's. To ensure that a specification is sufficiently precise, stating the "obvious" is necessary.

The second maxim is necessary to avoid nasty surprises in an implementation. In many cases, we might consider an application to be sufficiently simple that we can trust the programmer to get it right. In general, such trust is a bad idea. It is better for the specifier to maintain a respectfully adversarial relationship with the implementor.

4.3. Basic Rolodex Definitions

We are now ready to define the basic formal specifications for the Rolodex system. As a matter of style, we will first state a predicate semi-formally in English, and then refine it to formal logic. The English version can be retained as a comment, to aid in the human understanding of the specification. Let us begin with the Rolodex Add operation:

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    precondition: (* None *);
    postcondition: (* The given card is in the output Rolodex. *);
end Add;
```

Some new SpecL notation has been introduced here. First is the expanded syntax for an operation definition, the general format of which is the following:

```
operation name(inputs)->(outputs)
    attributes; ...
end [name]
```

The *attributes* specify properties of the operation, which can be system- and/or user-defined. The `precondition` and `postcondition` attributes are system-defined.

Other new notation is the addition of names for the inputs and outputs. A full operation signature has the following general format:

```
operation operation-name(input-name:type,...)->(output-name:type,...)
```

As described in the SpecL reference manual, objects define formal *types*, in the same sense as in strongly-typed programming languages. Given this, an SpecL operation definition is analogous to a procedure (or function) definition, such as the following Pascalese version of the Add operation:

```
procedure Add(Rolodex;Card):Rolodex;
```

Such a definition is sufficient to define the type signature of the procedure. However, in a complete procedure declaration, the parameters must be given names so they can be referenced in the procedure body. Hence, a full declaration for the above procedure is

```
procedure Add(r:Rolodex; c:Card):Rolodex;
```

In an SpecL operation definition, the input and output objects must be referenced in the pre- and postcondition predicates. Hence, parameter names in an operation definition serve essentially the same purpose as in a programming language procedure definition. Viz., they provide the means to identify specific input and output objects by name.

There are two major syntactic differences between an SpecL operation signature and a comparable procedure declaration in a Pascal-class language. Unlike most programming languages, the single apostrophe character is legal in an SpecL identifier. By convention, if an operation uses the same type as both an input and output, the name of the output is the same as the input with an apostrophe appended, and the apostrophe is read "prime". The other syntactic difference is the explicit naming of output objects. Most programming languages do not support multi-valued functions, and the output of a function is specified operationally with a `return` statement. In an SpecL specification, the formal specification does not contain an operational "return", so the output object(s) must be explicitly named for reference purposes. In addition, SpecL supports multi-valued operations, which require outputs to be distinguished by separate names.

The final piece of new notation is the syntax for SpecL comments. They are enclosed in the brackets "`(*`" and "`*)`".

Having covered notation, we can return to the main focus of formal specification. The English comment for the Add postcondition specifies the most fundamental property of an Add operation -- upon completion of the operation, the given Card is in the output Rolodex. Formally,

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    precondition: (* None *);
    postcondition:
        c in r'    (* The given card is in the output Rolodex. *);
end Add;
```

The `in` operator is built-in list membership. Its operands are an object and a list of that object[3]. In this case the operands are a Card and a Card* (a Rolodex).

Having no precondition is equivalent to a precondition of true. In general true preconditions are reasonable, given that there is no specific condition that must be met before the operation begins. In the case of the Rolodex Add operation, a true precondition is probably not strong enough, particularly if we need to impose a requirement that disallows duplicate entries in the Rolodex. We will address this requirement a little later.

One of the fundamental questions that must be asked of pre- and postconditions is if they are *strong enough*. In general, adding additional predicate clauses will strengthen the conditions. For example, the true precondition for Add is relatively weaker than one that specifies that the input card is not already in the input Rolodex.

In general, there are two aims to strengthening a specification.

1. Ensuring that all user-level requirements are met (cf. Maxim 1 above).
2. Ensuring that a system implementation works properly (cf Maxim 2).

The former is accomplished via continued consultation with the end user. The latter requires an experienced analyst, who understands the kinds of problems that may arise in a system implementation.

In the case of the Rolodex and similar database applications, an area of potential implementation error is the introduction of spurious entries into the database and/or the spurious deletion of entries. To avoid such spurious operations, the specification of Add can be strengthened as follows:

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    precondition: (* None *);
    postcondition:

        (* The given input card is in the output Rolodex. *)
        c in r'

            and

        (* Any other card is in the output Rolodex
         * only if it is in the input Rolodex *)
        forall (c':Card | c' != c)
            if (c' in r)
            then (c' in r')
            else not (c' in r');
end Add;
```

This specification introduces the use of the universal quantification operator, `forall`. Universal quantification in SpecL has the same meaning as in standard (typed) predicate logic. The general format is the following:

```
forall (x:t) predicate
```

This is read "for all values *x* of type *t*, *predicate* is true" where *x* must appear somewhere in *predicate*. There are also two extended forms of `forall`, shown in Table 3. In general, universal quantification is used frequently when specifying predicates on list objects, as upcoming examples illustrate.

While this example is a good illustration of specification strengthening, there are easier ways to specify the same meaning in SpecL. For example, the postcondition logic can be simplified to the following:

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    postcondition:
        forall (c':Card)
            (c' in r') iff ((c' in r) or (c' = c));
```

[3] The `in` operator is overloaded to accept two list operands, as well.

Extended Form	Reading	Equivalent To
`forall (x:t | p1) p2`	For all *x* of type *t*, such that *p1* is true, *p2* is true.	`forall (x:t) if p1 then p2`
`forall (x in l) p`	For all *x* in *l*, *p* is true.	`forall (x:basetype(l)) if x in l then p`

Table 3: Extended forms of universal quantification..

```
end Add;
```

In general, predicate simplification is beneficial when it helps clarify the specification.

Another way to simplify this specification is to use a constructive list operator, as follows:

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    postcondition:
        r' = r + c;
end Add;
```

where '+' in this context is the list append operator. As a matter of style, this primer presents *analytic* rather than *constructive* specifications. Generally, a constructive specification is one that describes the output of an operation using a constructive operation on the inputs, whereas an analytic specification describes output without using constructive operations. In SpecL, the constructive operations are those described as "construction" in Table 2. There is debate among software engineers as to the relative merits of constructive versus non-constructive specification, but further discussion of this topic is beyond the scope of the primer. In this primer, all specification examples are analytic.

Given the development of Add thus far, we can provide a comparable level of formal specification for the other three Rolodex operations. In order to do so, we observe that the remaining operations each takes a card name as input. To formalize these operations, we must be able to refer to the `Name` component of a card. This leads to a very common occurrence in the process of formalizing a specification. Namely, we need to update the definition of an existing object, based on the need to specify a requirement precisely. The update required here is to provide names for the components of the Card object, so that the components can be individually referenced. Here is the updated definition:

```
object Card is name:Name and id:Id and age:Age and sex:Sex and addr:Address;
```

This definition uses name/type pairs in the same manner as in a full operation signature. In a predicate, the named fields of a Card can be referenced using the '.' operator, which has a comparable meaning to its use in a programming language. That is, an SpecL tuple defined with `and` composition is essentially the same as a record structure in a programming language. Given this, the '.' is used to select a field of the tuple.

With this update to the Card object, here are the initial formal specifications for the Delete and Change operations. These specifications include "no spurious data" requirements.

```
op Delete(r:Rolodex, n:Name)->(r':Rolodex)
    pre: ;
    post:
        (* Cards in the output Rolodex consist of those
         * in the input Rolodex, except for those with
         * the same name as the given input name. *)
        forall (c':Card)
            (c' in r') iff ((c' in r) and (c'.name != n));
end;

op Change(r:Rolodex, n:Name, c:Card)->(r':Rolodex)
    pre: ;
```

```
    post:
        (* The given card is in the output Rolodex *)
        (c in r')

            and

        (* No other cards of the same name are in the Rolodex *)
        forall (c':Card | c' != c)
            (c' in r') iff ((c' in r) and (c'.name != n));
end;
```

Careful examination of the Change specification reveals a behavior that may not be appropriate at the user level. Specifically, the Change operation takes a name and a new card, and replaces *all* cards of the given name with the given card. Resolution of this potential problem involves deciding whether or not two or more cards of the same name may be present in a Rolodex. We address this in the next section of primer.

Note that we have not yet specified the Find operation. This requires some additional user-level analysis, which we now pursue.

4.4. Basic User-Level Requirements

To this point, we have formalized the most basic specifications of the Rolodex operations. It is now appropriate to consider the formal definition of basic user-level requirements. To start, there are a number of "obvious" user-level requirements, including the following:

1. Duplicate entries are not allowed in the Rolodex
2. Input values are checked for validity.
3. If the Find operation outputs more than one card, the output should be sorted in some appropriate order.

An historical note is of interest with regards to such requirements. In the not-so-distance past, the process of formalizing a specification entailed formalizing the English with which the specification was stated. For example, the first of the above requirements could be stated "formally" as follows:

A Rolodex shall not contain duplicate entries.

While this may not seem to be a substantial improvement to the original statement of the requirement, it represents a seriously-proposed approach to formalization. With this approach, a number of possible forms of natural language are standardized with a restricted vocabulary. For example, all formal requirements are expressed using "shall" instead of other comparable English words such as "should", "ought to", or "allowed to". This idea of formalizing English is noteworthy because it has been widely used in practice, and significant documents have been "formalized" in this manner. While such rules can indeed help with the formalization process, they fall well short of a fully formal basis for requirements specification.

4.4.1. No Duplicates

Analysis of the no duplicates requirement provides fine support for the "nothing-is-obvious" maxim. While we may expect reasonable people to understand what "no duplicates" means, there are in fact a number of plausible interpretations. Three such interpretations are the following:

1. No two cards in a Rolodex have exactly the same values for all card fields.
2. No two cards in a Rolodex have the same name.
3. No two cards in the Rolodex have the same unique key, such as the Id field of the card.

Which of these interpretations to choose is categorically *not* a matter for a programmer to decide. Rather, it should be decided at the user specification level, by the analyst in consultation with the end users. We could even grant that most programmers are reasonably smart, so in this case we might safely assume that a programmer could make the correct decision, or know enough to consult with the user to resolve the problem. Suppose, however, we were specifying data records in a much more complicated application domain, such as aeronautics. In this domain we might have a data object such as an anomaly list, with record fields like PreFlight, Taxi, InFlight, Approach, and Landing. What does it mean to disallow duplicates in an anomalies database? Which field, if any, could be used as a unique key? The point is that such questions need to be answered by end users and/or application domain experts. Such questions should most certainly not be left unanswered when the programmer begins work, since the programmer may well not know how to answer them.

Let us assume for our example that we have decided on the third of the alternative interpretations above. This means that cards in the Rolodex need only differ in the Id value. In particular, there may be multiple cards with the same name. This fact has a significant impact on the operations that take a Name as input. These operations use the Name as a search key, which may not refer to a unique card. We will address the effects of this as we continue to refine the specification.

The basic strategy for disallowing duplicates is to define a precondition on Add that checks for an entry of the same Id as the card being added. Here is the refined specification for Add. For brevity, the postcondition is omitted:

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    precondition:
        (* There is no card in the input Rolodex with the same Id
         * as the given input card. *)
        not (exists (c' in r) c'.id = c.id);

    postcondition: (* Same as above *);
end Add;
```

Here we have introduced the second form of quantification in SpecL -- existential. Table 4 summarizes the formats.

A discussion of the exact nature of any precondition is in order. By definition, failure of a precondition means that the operation is prevented from executing. More precisely, precondition failure means that the operation fails and produces a value of nil. The nil value in SpecL is defined for all object types, and is a distinct value from any other value of a given type.

The abstract meaning of precondition failure does not define how operation failure is perceived by the end user. Generally, the end-user should see an appropriate error message when an operation fails. The details of such error messages are typically abstracted out of the formal specification.

4.4.2. Input Value Checking

There are a number of possibilities for input value checking. As a basic example, consider the following updated version of Add, where the input value constraints are defined formally with accompanying comments.

Form	**Reading**	**Equivalent To**
`exists (`*x*`:`*t*`)` *p*	There exists *x* of type *t* such that predicate *p* is true.	
`exists (`*x*`:`*t* `|` *p1*`)` *p2*	There exists *x* of type *t*, such that *p1* is true and *p2* is true.	`exists (`*x*`:`*t*`)` *p1* `and` *p2*
`exists (`*x* `in` *l*`)` *p*	There exists *x* in *l* such that *p* is true.	`exists (`*x*`:`*basetype(s)*`) (`*x* `in` *s*`) and` *p*

Table 4: Forms of existential quantification..

```
operation Add(r:Rolodex, c:Card)->(r':Rolodex)
    precondition:
        (#(c.name) <= 30)  (* The length of the name is <= 30 characters *)
            and
        (#(c.id) = 9)  (* The length (i.e, number of digits) of the id is 9 *)
            and
        ((c.age >= 0) and (c.age <= 200))  (* Age is a reasonable range *)
            and
        (#(c.addr) < 40)  (* The length of the address is <= 40 chars *)

        and not (exists (c' in r) c'.id = c.id); (* No dups condition from above *)

    postcondition: (* Same as above *);
end;
```

The SpecL '#' operator is the built-in length operator for lists, strings, and integers (for integers it computes the number of digits). Later when we consider interface enhancements, additional input value checking will be specified.

4.4.3. Ordering of Multi-Card Lists

Given that more than one card of the same name can be in a Rolodex, the Find operation should produce a list of outputs rather than a single Card. Hence, the initial signature of Find needs to be updated. This is an example where a clarification of user-level requirements leads to a fundamental refinement of the formal specification.

With the original Find signature, a formal specification would look something like this:

```
op Find(r:Rolodex, n:Name)->(c:Card)
    pre:
        (* There is a card in the given Rolodex
         * with the given name. *)
        exists (c' in r) c'.name = n;
    post:
        (* The output card is in the given Rolodex
         * and has the given name. *)
        (c in r) and (c.name = n);
end;
```

The problem here is that specification does not indicate which of possible multiple cards of the same name is found. An updated formal specification for Find is:

```
op Find(r:Rolodex, n:Name)->(cl:CardList)
    post:
        (* Cards in the output list consist of those
         * in the input Rolodex with the given name *)
        forall (c:Card)
            (c in cl) iff ((c in r) and (c.name = n));
end;

obj CardList = Card*;
```

An important question to consider is the order of the card list output. The list structure in SpecL does not guarantee any content ordering[4]. Hence, in the immediately preceding specification of Find, no order for the CardList output can be assumed. Ordering of outputs from an operation such as Find is exemplary of a requirement that is easy to

[4] A list is an indexed collection, such that the nth item can be located. However, no content ordering can be assumed in a list, unless explicitly specified.

overlook. As with other requirements, we should not trust that a programmer will do the right thing in the absence of a formal statement. In this case, the programmer may not even think that there is problem if an output list is displayed in some internal order, such as the order cards were stored in a hash table.

In order to specify card list ordering, we must strengthen the Find postcondition. In consultation with our Rolodex users, suppose we have determined that a list of cards with the same name should be ordered by Id. That is, we are specifying that the output of Find is sorted by the Id field of a card. The formal specification of sorting is a more advanced application of logic than we have seen thus far. Here it is, in the context of the Find definition:

```
op Find(r:Rolodex, n:Name)->(cl:CardList)
    post:

        (* Cards in the output list consist of those in the input Rolodex with
         * the given name. *)
        forall (c:Card)
            (c in cl) iff ((c in r) and (c.name = n))

            and

        (* The output card list is sorted in ascending order by card id. *)
        forall (i:integer | (i >= 1) and (i < #cl))
            cl[i].id < cl[i+1].id;
end;
```

An English translation of this forall logic is the following:

> For each position i in the output list, such that i is between the first and the second to the last positions in the list, the ith element of the list is less than the i+1st element of the list.

The reader should study this logic to be satisfied that it specifies sorting satisfactorily.

There are two further points of discussion to be addressed with regards to the specification of sorting: unbounded quantification and the use of auxiliary functions in SpecL. These are covered in the next two subsections of the primer.

4.4.4. Unbounded Quantification

What would happen to the meaning of the sorting predicate if the restrictions on the range of i were not present? I.e., if the sorting logic in the postcondition were changed to the following:

```
forall(i: integer)
    cl[i].id < cl[i+1].id
```

The meaning here is an *unbounded quantification*. That is, the quantifier operates over the infinite range of all integers. In principle, there is nothing wrong with unbounded quantification. For example, the original anti-spurious requirements for the Add operation were expressed using unbounded quantification. One might argue for range restrictions on the grounds of efficiency, but as noted earlier, efficiency of this nature is not of concern in an abstract specification.

The potential problem with unbounded quantification is that the body of the universal quantifier may not have the correct value in an unbounded range, and hence the value of the entire quantifier expression may be false when we expect it to be true. This is the case in the unbounded quantification used in the sorting predicate just above. Here is specifically what goes wrong in this logic:

- when i is outside of the range of [1..#cl], then the value of the expression cl[i] is nil. This is the case by the language rules of SpecL, that define the value of an index expression to be nil if the value of the index is outside of the bounds of the indexed list.
- When the value of cl[i] is nil, the value of cl[i].id goes to nil. This again is by the rules of SpecL, that define the value of a selection expression (containing '.') to be nil if the value of the object being selected from is nil.

- This in turn leads to the following expression as the body of the forall:

  nil < r[i+1].id

  the value of which is false. This result is due to the SpecL rule that defines the value of '<' to be false if either or both of its operands is nil.
- Finally, any value of false in the body of the forall drives the value of the entire forall to false, by the normal rules of forall. Viz., the value of the forall is false if any one or more values of the body is false.

To some extent, the exact outcome of the unbounded quantification above is due to the particular semantic rules of SpecL. In general, however, unbounded quantification is potentially problematic under any logical semantics. The point is that one needs to be careful when using unbounded quantification to ensure that the body of the quantifier has a well understood value over the entire unbounded range of quantification.

4.4.5. Using Auxiliary Functions

The postcondition in the most recent definition of Find is a little lengthy. In practice, predicates significantly longer than this can appear in the specification of a complex operation. When pre- or postconditions become unduly long, it is useful to use *auxiliary functions* to organize the logic. An auxiliary function is much the same as function definition in a programming language, with the restrictions of functional semantics discussed earlier in the primer. The purpose of an auxiliary function is modularize a piece of logic, give it a mnemonic name, and allow that logic to be invoked in one or more places.

As an example, here is the last definition of Find using two auxiliary functions.

```
op Find(r:Rolodex, n:Name)->(cl:CardList)
    post:
        CardsFound(r,n,cl)
            and
        SortedById(cl);
end;

function CardsFound(r:Rolodex, n:Name, cl:CardList)->boolean =
    (* Cards in the given card list consist of those in the given Rolodex with
     * the given name *)
    forall (c:Card)
        (c in cl) iff ((c in r) and (c.name = n));

function SortedById(cl:CardList)->boolean =
   (* If the the given card list has more than one card, it is sorted in
    * ascending order by card id. *)
    if (#cl > 1) then
        forall (i:integer | (i >= 1) and (i < #cl))
            cl[i].id < cl[i+1].id;
```

Semantically, there is no difference between an auxiliary function and an operation. They both define objects of a function type. In fact, the keyword "operation" can be used in place of the keyword "function". The separate keywords are provided to suggest different usages within a specification. By convention the keyword "operation" should be used to define an operation that is directly visible to the end user, i.e., an operation that can be traced directly to some user interface element. In contrast, the keyword "function" should be used for functions that are not normally visible to the user, but which are used solely to clarify the logic of a formal specification.

5. User-Level Refinements and Enhancements

In this section we consider a number of refinements and enhancements to the user interface of the Rolodex system and how these can be specified formally.

5.1. Pattern-Based Search

Suppose we would like to locate Rolodex cards by keys other than just the name. To be fully general, we could allow search by patterns for any one of the keys. At the interface level, the Find operation would present the same dialog used for Add. In the case of Find, entries in the dialog box would be patterns rather than just strings or numbers. For example, Figure 5 shows a search dialog that will find all cards with age less than 40 and sex male. An entry for any one of the five card fields can contain a single instance of one of the following patterns:

Operator	Meaning
x	matches the value x (a string or number)
< x	matches all values less than x
> x	matches all values greater than x
x - y	matches all values between x and y

For simplicity, we assume that exact match is necessary for strings, but this requirement is probably too strict for a practical user interface. E.g., a user should be able to enter "Smith" in the name field to find all cards with "Smith" somewhere in the name. The reader is invited to enhance the specification that follows with a feature for such partial matching.

Given below are selected excerpts of the formal specification for the enhanced Find operation. The specification focuses on searching with patterns in the Age field of a card. Formal specification of searching by the other card fields (name, id, sex, and address) is very similar. The gist of the pattern-search specification is the definition of the object SearchInfo and the additions to the postcondition of Find. Notationally, the SpecL `description` attribute is used to describe the major objects and operations. The value of the description attribute is simply a comment.

```
object SearchInfo = np:NamePattern and idp:IdPattern and
        ap:AgePattern and sp:SexPattern and adp:AddressPattern
    description: (*
        Each component of SearchInfo is a search pattern that corresponds to
        one of the fields of a card.
    *);
end SearchInfo;
```

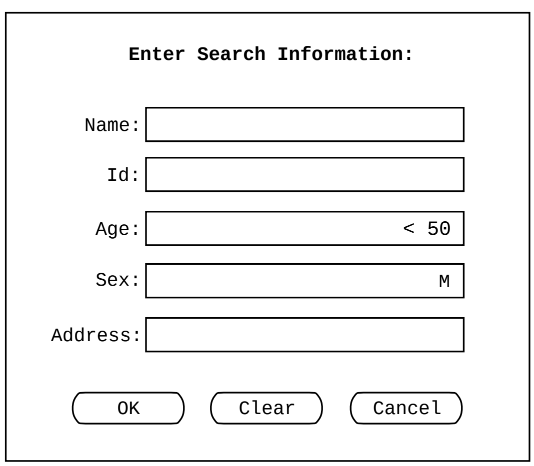

Figure 5: A Pattern Search Dialog.

```
obj NamePattern =  ...;
obj IdPattern =  ...;

object AgePattern = lessp:AgeLessThan or gtrp:AgeGreaterThan or
        rangep:AgeRange or eqp:Age
    description: (*
        An AgePattern allows the user to search for cards with an age value
        less than a given age, greater than a given age, between a range of
        given ages, equal to a specific age, or with a specific age.
    *);
end AgePattern;

obj SexPattern = ...;
obj AddressPattern = ...;

object AgeLessThan = lts:LessThanSymbol and age:Age
    description: (*
        This pattern specifies all ages less than a particular age.
    *);
end AgeLessThan;

object AgeGreaterThan = lts:GreaterThanSymbol and age:Age
    description: (*
        This pattern specifies all ages greater than a particular age.
    *);
end AgeGreaterThan;

object AgeRange = age1:Age and rs:RangeSymbol and age2:Age
    description: (*
        This pattern specifies all ages in a range between age1 and age2.
    *);
end AgeRange;

object LessThanSymbol;
object GreaterThanSymbol;
object RangeSymbol;

operation Find (r:Rolodex, si:SearchInfo)->(cl:Card*)
    pre:  ... ;
    post:
        (*
         * All cards in the output list must be found according to the given
         * search info, and the output list must be sorted by Card id.
         *)
        CardsFound(r,si,cl)
            and
        SortedById(cl);
    description: (*
        Find zero or more cards that match the constraints specified in the
        given SearchInfo.
    *);
end Find;

function CardsFound(r:Rolodex, si:SearchInfo, cl:Card*)->boolean =
    (* Cards in the given card list consist of those, and only those, in the
     * given Rolodex that match the given search info. *)
    forall (c:Card)
        (c in cl) iff ((c in r) and Match(c,si));
```

```
function Match(c:Card, si:SearchInfo)->boolean =
        MatchName(c.name, si.np) and
        MatchId(c.id, si.idp) and
        MatchAge(c.age, si.ap) and
        MatchSex(c.sex, si.sp) and
        MatchAddress(c.addr, si.adp);

function MatchName(Name, NamePattern)->boolean =  ... ;
function MatchId(Id, Idpattern)->boolean =  ... ;

function MatchAge(age:Age, ap:AgePattern)->boolean =
    if (ap?.lessp) then
        age < ap.lessp.age
    else if (ap?.gtrp) then
        age > ap.gtrp.age
    else if (ap?.rangep) then
        (age >= ap.rangep.age1) and (age <= ap.rangep.age2)
    else if (ap?.eqp) then
        age = ap.eqp.age
    else if (ap = nil) then
        true;

function MatchSex(Sex, SexPattern)->boolean = ... ;
function MatchAddress(Address, AddressPattern)->boolean =  ...;
```

5.2. Historical Dialogs

It is typical in a graphical user interface that a dialog retains values from the last time it was displayed. For example, when the Add dialog is displayed for the second time and beyond, it could contain the last values entered. Some users may find such historical dialogs undesirable, so the system could contain an option that turns the feature on or off. With historical dialogs on, each dialog box displays the previously entered value(s). With historical dialogs off, each dialog box is empty when displayed.

Given below are selected excerpts of the formal specification for optional historical dialogs. The gist of the specification is the definition of the SystemState object and the decomposition of the main Rolodex operations into two operations. For example, Add is decomposed into InitiateAdd and ConfirmAdd. From the user interface perspective, InitiateAdd is invoked by selection of Add in the Rolodex menu; ConfirmAdd is invoked by selection of the OK button in the Add dialog.

```
operation InitiateAdd(r:Rolodex)->(cd:CardData)
    post:
        (* If the historical dialog option is on, then output the previously
         * entered card data, else output empty card data. *)
        if r.state.options.showprevdata
        then cd.c = r.state.lastadd
        else cd.c = nil;
end;

operation ConfirmAdd(r:Rolodex, cd:CardData)->(r’:Rolodex)
    pre: (* Same precondition as original Add,  but replace all occurrences of
          * variable c with cd.c *);
    post: (* Same postcondition as original Add, with same replacements as in
           * precondition *);
end;

obj CardData = c:Card and flag:boolean
    description: (* See discussion below *);
```

```
end;

object Rolodex = state:SystemState and cards:Card*;
object SystemState = lastadd:LastAddInput and LastDeleteInput
        and LastChangeInput and LastSearchInput and options:Options;
object LastAddInput = Card;
object LastDeleteInput = Name;
object LastChangeInput = Name and Card;
object LastSearchInput = SearchInfo;
object Options = showprevdata:ShowPreviousData  and ... ;
object ShowPreviousData = boolean;
```

It is important to note the change to the Rolodex object, which is now a tuple. The ramifications of this change are that all references to a Rolodex variable, for example *r*, must be changed to *r.cards* throughout the specification.

The CardData object has no other purpose than to constrain a Confirm operation to take input from the companion Initiate operation. The flag component of CardData has no other purpose than to ensure that Card and CardData are distinct types. This specification borders on too operational, since its purpose to to specify an order of operations. In general, the specification of such ordering should be done judiciously, since it constrains an implementation to a particular style of interaction.

5.3. Check Pointing

It is common for database systems, such as our Rolodex, to save the contents of the Rolodex at regular intervals. Such "check pointing" saves are a service to the user in case some catastrophic failure occurs when a user has not recently performed a manual save operation.

Given below are selected excerpts of the formal specification for check pointing. Note the addition of a FileSpace object, which is a model for an external (operating) system file storage. This is used in the specification of the Rolodex File operations, where the definition of the Save operation is found.

```
op Add(r:Rolodex,fs:FileSpace,c:Card)->(r':Rolodex,fs':FileSpace)
    post:
      (* previous postcond and *)

      if r.state.options.chkpton then
          if r.state.chkpt = 0 then    (* Time to do checkpoint save *)
              r'.state.chkpt =
                  r.state.options.chkptinterval and
              fs' = Save(fs,r)
          else                         (* Not time yet, decrement counter *)
              r'.state.chkpt = r.state.chkpt - 1 and
              fs' = fs
      else
          true;
end;

obj SystemState = (* previous components and *) chkpt:CheckpointCount;
obj Options = (* previous components and *)
    chkpton:CheckpointOnOff and chkptinterval:CheckpointInterval;
obj CheckpointCount = integer;
obj CheckpointInterval = integer;
obj CheckpointOnOff = boolean;
```

Note carefully the "else true" clause in the postcondition. If-then-else is not a programming language control construct, though it may seem like one most of the time. Rather, if-then-else is a choice expression that has one of two values depending on the value of the if test. If the else clause is missing and the value of the if test is false, then the value of the entire if-then-else is false, which is not necessarily the desired value. In this case, for example, we do not want the Add operation to fail if checkpointing is off.

Also note that the equality operator, '=', is *not* an assignment operator, as much as it may look as such. What the equality operator means in an expression such as

```
r'.state.chkpt = r.state.chkpt - 1
```

is *not* that `r'.state.chkpt` is assigned the value `r.state.chkpt - 1`. Rather, this is an expression that specifies the output value of `r'.state.chkpt` is equal to one less than the input value of `r.state.chkpt`. In an implementation of this operation, an assignment of some form will most likely take place, in order for the postcondition to become true. However, the postcondition by itself *does no assignment*.

5.4. Undo

It is quite common for interactive systems to have an Undo function that reverses the effects of the most recent operation. A more advanced undo/redo capability is sometimes available, where a number of recent operations can be undone or redone.

Given below are selected excerpts of the formal specification for a simple one-level undo facility. The specifications for Add and Undo operations are given. A complete specification would include updates to Delete and Change, comparable to those for Add.

```
obj Rolodex = state:SystemState and cards:CardList; (* Same as above *)
obj SystemState = (* previous components and *) prev:CardList;
obj Card = (* as usual *);

op Add(r:Rolodex, c:Card)->(r':Rolodex)
    post:
        (* The following is the original postcond, with substitution
         * of r.cards for r and r'.cards for r' *)
        forall (c':Card)
            (c' in r'.cards) iff ((c' in r.cards) or (c' = c))

            and

        (* Save the cards of the input Rolodex as the previous card list. *)
        (r'.state.prev = r.cards);

end Add;

op Undo(r: Rolodex)->(r':Rolodex)
    post:
        (* If the input Rolodex has a previous card list,
         * then the cards of the output Rolodex are that list
         * and the previous of the output is nil (so only one
         * level of undo is possible).
         * Otherwise, the output Rolodex is the same as the
         * input Rolodex. *)
        if (r.state.prev != nil)
        then (r'.cards = r.state.prev) and (r'.state.prev = nil)
        else r' = r;
end;
```

Based on this example, the reader is invited to specify a more advanced undo/redo feature that would allow a selectable number of operations to be undone and redone.

5.5. Security

For some database systems, security is necessary. In the case of our simple Rolodex, a plausible security requirement would be to allow only privileged users to perform the Add operation. Given below are selected excerpts of the formal specification for such a security scheme. The gist of the specification is to define a UserTable containing UserInfo records. The records are tuples of a UserId and privilege Level. Each user is assigned a system Id (potentially different from a card id) and a privilege level. When a user logs in to the system, the user id is supplied, whereupon the privilege level is extracted from the user table. The specification below does not define the operations necessary to maintain a user table, i.e., adding and deleting user records.

```
obj SystemState = (* previous components and *)
        users:UserTable and addok:boolean;
obj UserTable = UserInfo*;
obj UserInfo = uid:UserId and level:Level;
obj UserId = string;
obj Level = priv:Privileged or nonpriv:Nonprivileged;
obj Privileged;
obj Nonprivileged;

op Login(uid:UserId,state:SystemState)->(r':Rolodex)
    post:
        (* Adds are OK in the output Rolodex if the given user
         * is privileged, otherwise adds are not OK. *)
        if FindUser(state.users,uid).level?.priv
        then r'.state.addok = true
        else r'.state.addok = false

            and

        (* The output Rolodex has no cards. *)
        (r'.cards = nil);

    description: (*
        The operational model here is that the login operation creates an
        initial empty Rolodex, with the appropriate value for the addok flag.
        This model affects the specification of the File Open operation, in
        that the Open will read Rolodex cards from a file, but maintain the
        Rolodex state established by Login. *);
end;

op FindUser(ut:UserTable,uid:UserId)->(uinfo:UserInfo)
    pre:
       exists (uinfo' in ut) uinfo'.uid = uid;
    post:
       (uinfo in ut) and (uinfo.uid = uid);
end;
```

6. Rolodex File Operations

There are number of approaches for defining the file operations of the Rolodex system, or other comparable system that uses external file storage. The approaches vary in how abstractly versus how concretely the external file storage medium is modeled.

6.1. Abstract File Operations

In the most abstract approach, we can view the external file storage medium as simply a collection (i.e., SpecL list) of Rolodex objects. With this view, the file operations can be defined as follows:

```
obj FileSpace = Rolodex*;
obj Rolodex = name:Name and cards:Card*;
obj Card = ...;
obj Name = string;

operation New()->r':Rolodex
```

```
    post: r' = nil;
end New;

operation Open(fs:FileSpace, n:Name)->r':Rolodex
    post:
        exists (r in fs) (r.name = n) and (r' = r);
    description: (*
        Open is essentially a find operation, with the same form of
        postcondition as a basic find.
    *);
end Open;

operation Save(fs:FileSpace, r:Rolodex)->fs':FileSpace
    post:
        (* The given rolodex is in the output filespace *)
        (r in fs')

            and

        (* No rolodexes of the same name are in the filespace. *)
        forall (r':Rolodex | r' != r)
            (r' in fs') iff ((r' in fs) and (r'.name != r.name));

    description: (*
        Save is essentially a change operation, with the same form of
        postcondition as a basic change.  There is no precondition, since a
        file for the given rolodex need not already exist in the file space.
    *);
end Save;

operation SaveAs(fs: FileSpace, r:Rolodex, n:Name)->fs':FileSpace

    pre: ;
        (* Note there is no precondition that prevents unintentional
         * overwriting of a rolodex of the same name in the input fs.  This
         * could be added if necessary. *)

    post:
        (* There is a rolodex, with the same cards as the input rolodex and the
         * given name, in the output filespace. *)
        exists (r' in fs') (r'.name = n) and (r'.cards = r.cards)

            and

        (* No other rolodexes of the same name are in the filespace. *)
        forall (r':Rolodex | r' != r)
            (r' in fs') iff ((r' in fs) and (r'.name != r.name));

    description: (*
        SaveAs differs from Save in that the name of the saved rolodex must be
        that given, rather than the original name of the input rolodex.  This
        difference is reflected in the first clause of the postcondition of
        SaveAs vis a vis Save.  The second postcondition clause is the same in
        both Save and SaveAs.
    *);
end SaveAs;

operation Print(r:Rolodex)->r':Rolodex
    post: SortedByName(r');
```

```
    description: (*
        Printing produces a sorted rolodex.  It could be augmented with an
        Options input that would specify different sorting orders.  Note that
        this abstract view of printing does not formally specify print
        formatting details.  This issue is addressed in the more concrete level
        of file operation specification.
    *);
end Print;
```

This view of file operations abstracts out all details of a file system, other than the fact that it stores Rolodexes. The signature of the print operation is also of interest in this view. In its most abstract form, a Print operation merely changes the order (and presumably physical medium) of a Rolodex, without performing any concrete formatting of the card data.

6.2. More Concrete File Operations

Making the file operations more concrete is primarily a matter of defining more concrete objects. The fundamental meaning of the operations does not change, but the formatting details do. Consider the following definition of a more concrete FileSpace, and the associated file operations.

```
obj FileSpace = File*;
obj File = type:FileType and name:FileName and data:FileData;
obj FileType = a:ASCIIType or b:BinaryType or rolo:RolodexType or ... ;
obj FileName = Name;
obj FileData = ascii:ASCIIData or bin:BinaryData or rolo:RolodexData or ... ;
obj ASCIIData = string*;
obj BinaryData = number*;
obj RolodexData = Rolodex;
obj ASCIIType;
obj BinaryType;
obj RolodexType;

obj Rolodex = name:Name and cards:Card*;
obj Card = ...;
obj Name = string;

operation New()->r':Rolodex
    post: r' = nil;
end New;

operation Open(fs:FileSpace, n:Name)->r':Rolodex
    post:
        (*
         * There is a file of the given name in either ASCII or Rolodex format
         * in the given file space.  If the latter, then the cards of the
         * output rolodex contain the converted ASCII data.  If the former
         * (i.e., file is in Rolodex format), then the cards of the output are
         * those in the file.
         *)
        exists (f in fs)
            (* File and rolodex names are the same *)
            (r'.name = f.name)

                and

            (* File data are the same as rolodex cards, parsed if ASCII *)
            if (f.data?.ascii) then
```

```
            r’.cards = ParseRolodexText(f.data.ascii)
        else if (f.data?.rolo) then
            r’ = f.data.rolo
        else false; (* Neither ascii nor Rolodex file format *)

    description: (*
        Open is essentially a find operation, with the same underlying form of
        postcondition as a basic find.  This more concrete version of Open must
        parse ASCII text data files, if that is the format in which a rolodex
        was saved.

        The postcondition relies on a non-trivial auxiliary function,
        ParseRolodexText, that would specify precisely the correspondence
        between the textual and abstract representations of rolodex card data.
    *);
end Open;

operation Save(fs:FileSpace, r:Rolodex, type:FileType)->fs’:FileSpace
    description: (*
        Save operates the same as the more abstract version of save, except
        that it uses the FileType input to determine the format of the file
        in the output filespace.
    *);

end Save;

operation SaveAs(fs: FileSpace, r:Rolodex, n:Name)->fs’:FileSpace

    post: ... ;  (* Comparable change to Save. *)

end SaveAs;

operation Print(r:Rolodex)->rp:RolodexPrintout
    post:
        exists (r’:Rolodex) (
          exists (c:Card) (
            ((c in r’.cards) iff (c in r.cards)) and
            SortedByName(r’.cards) and
            (rp = GeneratePrint(r’))
         )
        );

    description: (*
        Printing produces the ASCII text for a sorted Rolodex.  The
        postcondition here relies on a non-trivial auxiliary function,
        GeneratePrintText, that would define textual formatting details
        precisely.  GeneratePrintText is the complement of the earlier
        ParseRolodexText function.

        Note that the postcondition specifies sorting on the rolodex, rather
        than on the printout, as would be the case with the following logic:

            forall (c in r)
                (GenerateCardPrint(c) in rp) and SortedByName(rp)

        This logic for sorting a printout would be considerably more tedious,
        since it would have to perform parsing to extract the name field from
        the text representation of a card.
    *);
```

```
end Print;

obj RolodexPrintout;

function ParseRolodexText(ASCIIData)->Card*;
function GeneratePrint(Rolodex)->RolodexPrintout;
function SortedByName(Card*)->boolean;
```

The major new aspect of this definition is the modeling of different file formats. This is typical of many data processing applications. For example, most text processors allow the user to select between plain text format versus an application-specific format. In the case of an application such as a word processor, there may be some information loss in the plain text format (such as font types, etc). In the case of a database system, such as the rolodex, there should be no data loss in a text format. Rather, choosing the application-specific format of rolodex file is a matter of efficiency versus external readability.

6.3. Considering a Rolodex System Workspace

In the preceding file space specifications, no consideration was given to a user workspace that contains one or more active files. The purpose of defining such a workspace is to specify requirements such as a limit on the number of rolodex files that can be open simultaneously. Consider the following definitions:

```
obj FileSpace = File*;
obj File = name:FileName and data:FileData;
obj FileName = Name;

obj WorkSpace = name:Name and r:Rolodex
    description: (*
        A workspace models the use work area, which can contain at most one
        open rolodex file.
    *);
end;

operation New()->w':WorkSpace
    post: w'.name = "Untitled" and w'.r = nil;
end New;

operation Open(fs:FileSpace, n:Name)->w':Workspace
    post:
        exists (f in fs) (f.name = n) and (w'.r = f);
    description: (*
        Open is essentially a find operation, with the same form of
        postcondition as a basic find.
    *);
end Open;

operation Save(fs:FileSpace, w:Workspace)->fs':FileSpace
    post:
        (* The rolodex in the given workspace is in the output filespace *)
        exists (f in fs') (
            (f.name = w.name) and (f.data = w.r)

            and

            (* No files of the same name are in the filespace. *)
            forall (f':File | f' != f)
                (f' in fs') iff ((f' in fs) and (f'.name != w.name))
        );
```

```
    description: (*
        Save is essentially a change operation, with the same form of
        postcondition as a basic change.  There is no precondition, since a
        file for the given rolodex need not already exist in the file space.

        NOTE: in this workspace version of Save, there is a subtle scoping
        change in the postcondition compared to the earlier versions of Save.
        Specifically, the universal quantification is nested within the
        existential quantification.  This is because the File variable f is
        defined in the scope of the exists, not as an input parameter.
    *);
end Save;
```

In this model, a rolodex is a purely abstract object, without even a name. The workspace carries the name, which is transmitted to and from files when the user invokes save and open operations.

7. Considering Other Interface Styles

An important property of an abstract specification is to be as free of concrete interface details as possible. To illustrate this point, we can consider two different forms of concrete user interface, both of which map to the same abstract specification that has been developed in the primer.

Figure 6 shows a pushbutton-style interface to the Rolodex. It is much the same as the original interface, except the pulldown menus have been replaced with pushbuttons. In addition, the data entry area does not change. Rather, the user simply types in the card data fields, and selects a desired operation when done. Additional dialogs will pop up as necessary to request further inputs or display results.

The only significant difference for the specifier with this interface is that the both the Card and SearchInfo objects are displayed in the same physical screen area, rather than in separate dialogs. Initially, it might have been slightly more difficult to recognize Card and SearchInfo as separate objects.

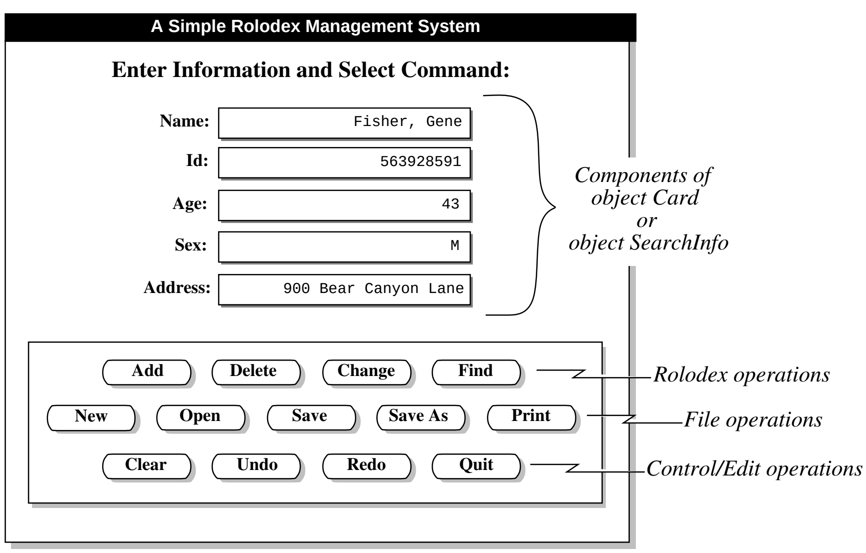

Figure 6: A Pushbutton-Style UI.

It is worth noting that to the extent the interface is confusing to the specifier, it may well be the end user as well. The notion of "form follows function" is an important one in software specification. That is, a system that is coherent and easy to specify for the analyst will be equally coherent and easy to use for the end user, and vice versa.

In either of the graphical interfaces for the Rolodex system, keyboard shortcuts could be available for the menu and/or pushbutton operations. Such shortcuts should have absolutely no effect on the formal specification.

Figure 7 shows a plain text-based Rolodex interface. This interface style is typical of that used with DOS or a UNIX shell. The same abstract Rolodex specification applies equally well to the textual interface as it does to the other two graphical interfaces.

Command	**Arguments**
a[dd]	name, id, age, sex, address
d[el]	name
c[hange]	name, id, age, sex, address
f[ind]	name
n[ew]	
o[pen]	file
s[ave]	[file]
[p]rint	[1]
[u]ndo	
[r]edo	
[q]uit	

Figure 7: A UNIX/DOS-Style Text UI.