Inquire: Predicate-Based Use and Reuse

Dewayne E. Perry
Software and Systems Research Center
AT&T Bell Laboratories
Murray Hill, NJ 07974

Steven S. Popovich
Department of Computer Science
Columbia University
New York, NY 10027

Abstract

There are four fundamental aspects of use and reuse in building systems from components: conceptualization, retrieval, selection and correct use. The most important barrier to use and reuse, initially at least, is that of conceptualization. The Inscape Environment is a specification-based software development environment (SDE) integrated by the constructive use of formal interface specifications. The purpose of the formal interface specifications and the semantic interconnections (created and maintained as software is built and evolved) is to make explicit the invisible semantic dependencies that result in conventionally built systems.

The important ingredient provided by *Inquire* in conceptualization, retrieval, selection and use is the set of predicates that describe the semantics of the elements in the interface. These predicates define the abstractions that are germane to the module interface and describe the properties of data objects and the assumptions and results of operations in a module. Use and reuse of components is based on a component's ability to provide needed semantics — either in terms of data object properties or in terms of operation behavior — at a particular point in a system. It is the purpose of *Inquire*, the browser and predicate-based search mechanism, to aid both the environment and the user in the search for the components that will provide the desired predicates — that is, the desired properties or behaviors — that are required to build and evolve an implementation correctly.

1. Introduction

There are four fundamental aspects of use and reuse in building systems from components:

- Conceptualization — understanding what is needed;
- Retrieval — finding possible components to use;
- Selection — determining which of the retrieved components to use; and
- Use — using the selected components correctly.

Compounding these issues is the fact that use and reuse occurs at multiple levels in building systems. Perry and Wolf [13] note that the architecture level is the most appropriate place to begin thinking about reuse of components because the system at that point is constrained the least. The process of design narrows the possible choices because the constraints increase with each design decision. In this paper, we focus on the problems of use and reuse where the components are modules, represented by their interfaces. Independent of the granularity of the component, the most important barrier to use and reuse, initially at least, is that of conceptualization — that is, of understanding the components and their underlying models; in short, understanding what is needed at any particular place in the system.

The Inscape Environment [8,9] is a specification-based software development environment (SDE) integrated by the constructive use of formal interface specifications. These specifications serve two purposes: first, they express the intent of the designer about the externally visible behavior of the operations described in the interface and define the properties of data objects declared and exported in the interface; second, they provide the basis for the construction of semantic interconnections [8] among interfaces and between interfaces and implementations. On the basis of these two uses, Inscape makes explicit the semantic dependencies among the various components in a software system, which are implicit in normally constructed systems, and uses these dependencies to assist in the construction and evolution of these systems.

Inscape uses these explicit semantic interconnections in the process of construction and evolution in several ways:

- to synthesize interfaces from implementations [10],
- to detect semantic errors [10],
- to determine the implications of changes [8,9], and
- to find objects to supply desired behavior and properties (that is, to supply the needed predicates).

It is on this last use that we focus our attention in this paper. The important ingredient in conceptualization, retrieval, selection and use is the set of first-order predicates (with typed parameters) that describe the semantics of the objects in the interface. These predicates define the abstractions that are germane to the module interface and describe the properties of data objects and the assumptions and results of operations in a module. For data objects, predicates represent additional information not provided by the typing mechanism. This is particularly important for a weakly typed language such as C. The amount of semantics provided by the specifier in this form

is a design issue: what are important semantic properties that need to be maintained for the data objects.

For operations, predicates represent the assumptions and results. To some extent, the level of detail provided is a design issue here as it is for data objects: what are the important semantic properties embodied in the operations. However, as the interfaces represent the external behavior of the operation, it is critical that all external state changes be represented in the specification. Only in local objects should state changes not be reported beyond the operation's boundaries. Note that in a language such as C that has only rudimentary abstraction mechanisms, information hiding is really information withholding. Conversely, one has a large degree of latitude in reporting observations about the current state made within the implementation — though it is usually the case that the more information the better.

Use and reuse[1] of a component is based on its ability to provide needed semantics — either in terms of data object properties or in terms of operation behavior — at a particular point in a system. It is the purpose of *Inquire*, the browser and predicate-based search mechanism, to aid both the environment and the user in the search for the components that will provide the desired predicates — that is, the desired properties or behaviors — required to build and evolve an implementation correctly.

In Section 2, we discuss the current state of browsers and retrieval mechanisms and indicate their strengths and weaknesses with resect to the four fundamental aspects of use and reuse. In Section 3, we indicate how Inquire differs from these mechanisms, delineate the shape of Inscape's predicate-based search and browse mechanism, and introduce Inquire's retrieval commands for operations, data objects and modules. In Section 4, we illustrate Inquire with an extended example. Finally, in Section 5, we summarize our research contributions, present the current state of the Inquire prototype, indicate some possible extensions, and discuss some of the future directions we might take.

2. Discovery: Browsing, and Retrieval

There are two general ways of supporting use and reuse: browsing and searching. With browsing, one looks around to see what there is and follows up various clues and leads to slowly build an understanding of what is available. With searching, one retrieves candidate components on the basis of some criteria, perhaps expressed as a user or system generated query.

The first approach is useful for basic or primitive discovery. Browsers enable the programmer to follow various relationships and dependencies among existing components and use analogical clues (for example, in the ways that components are used) to build a conceptualization of the components in their current context. For example, Interlisp's Masterscope [20], Steffen's Cscope [19], and CIA [2] provide static views of the organization of a system and basic means of navigating through that structure to build an understanding of the components. MView [1] provides both static and dynamic browsing capabilities, enabling the user to gain an understanding of the system structure and how the components fit in structurally, and also to gain an understanding how the components fit together executionally. The connections among the various components utilized by both the static and dynamic browsers are syntactic — that is, the dependencies are explicit,[2] but the reasons for the dependencies are not. For the rationale of the dependencies, one must depend on external sources of information. Moreover, the larger the system, the more difficult it is to have any confidence in browsing as anything other than a random attack on the problem. Thus, browsers are useful primarily for building an understanding of the system and its components, but have little to offer in terms of retrieval, selection, or use, except to the extent that the conceptual understanding of the system enables the user to perform those operations independent of retrieval tools.

The problems of retrieval are based, at least in part, on those problems we encounter in naming. Browsing works as well as it does partly because of the semantic clues that we embed in the naming of the various syntactic objects that form the basis of browsing. The success of classification schemes, and their associated retrieval mechanisms, depends on how well the keywords capture the appropriate conceptualizations that are embodied in the objects of interest. In particular, this is a difficult problem. The concepts and their associated names are application specific, even project specific. Conceptualization and naming are exceedingly difficult to generalize across different projects and applications.

Basic keyword-based retrieval schemes [5] (see Frakes and Gandel [4] for a general discussion of reuse library classification schemes), in particular, suffer from these naming problems. Proper utilization of keyword-based retrieval depends on how well the keywords can be used to describe the components and how well they are understood by those who try to retrieve components. The utility of these mechanisms lies in the fact that they provide a means of expressing basic concepts — that is, embedding semantics into the keywords. How well they do this depends on how well they approximate the concepts of importance for a particular application or a particular project. Retrieval is generally efficient, but help with selection and correct usage must be found outside the retrieval system. In addition, these basic approaches provide no means

1. If we look at Krueger's classification of reuse categories [7], we would say that Inscape supports various levels of reuse from design and code scavenging through software schemas, though there are hints of very high level languages and software architectures categories here as well.

2. Of course, there are many ways to make those connections as obscure as possible with pointers. It is in this latter case that an execution based browser is most useful.

of indicating relationships among or between these various concepts represented by the key-words, nor do they provide a mechanism for constructing inferences.

There are several approaches that provide solutions to the latter two problems. Prieto-Diaz' faceted classification scheme [14] and LaSSIE [3] are examples of these approaches. Prieto-Diaz uses a conceptual graph with weighted terms to indicate conceptual distance (or closeness) between the terms of a facet. The weights and the type-supertype relationships provide an ordering of terms by their conceptual distance. This ordering provides a useful approximation of some relationships among concepts and, with the supporting evaluation system, enables a more flexible retrieval of components.

LaSSIE is really more a conceptual mechanism than a retrieval mechanism. It uses a knowledge representation language (KRL) to construct the conceptual hierarchy that represents the conceptual model of the system. The advantage of using a KRL is that it provides a deductive mechanism as part of the conceptualization mechanism. The KRL's underlying mechanism enables LaSSIE to generalize as well as specialize from one concept to another, thus providing a much richer navigation mechanism than Diaz' faceted scheme. The strength of this approach is that various concepts are linked in various relationships by hand (using *is-a* and *part-of* relationships) thereby providing a logically tractable structure.

Both of these approaches provide a relatively rich mechanism for conceptualization. LaSSIE has references to functions at the leaves of the hierarchy, so that while its navigation facility is quite good, its retrieval mechanism is somewhat primitive. On the other hand, the navigation facility is very useful in selecting a particular component. The faceted approach provides retrieval with good recall and precision characteristics (see [14] for the relevant data). The primary drawback of both is that there is no direct connection with the source code, only an informal and hand-crafted connection. We then have the same update problem between the conceptual hierarchy and the source code that we have between informal architecture, design and implementation prose documents and the source code: changes in the one must be consistently propagated to and reflected by the other. How serious this problem is depends on how serious the conceptual drift is for a particular system. Given the fact that average releases of some large systems replace approximately 15-20% of the code [12], the potential for conceptual drift is significant.

Code-Base [18] incorporates an approach that goes part way towards directly connecting the conceptual structure with the application. The user interactively and incrementally does browsing and querying about the application and the underlying mechanism builds a knowledge base from the results of the browsing and querying. Conceptualization results from discovering pertinent information about the application; retrieval builds on that incremental conceptualization. Selection of components is then somewhat similar to that of LaSSIE but with a knowledge-base that is more closely tied to the application. An important aspect that has only begun to be considered is the evolution of the conceptual structure along with the evolution of the application (this issue is critical if knowledge-based systems are to be practical; in fact, it is a critical issue for all retrieval systems). Moreover, correct use of the retrieved and selected components is outside the view of Code-Base.

Two rather novel approaches in retrieval that depend directly on the source code are those of Rittri [15] and Runciman and Toyn [17]. Both are based on polymorphic type systems and provide a relaxation of the matching condition for functional types. They differ primarily in their ordering relation. Rittri's scheme provides independence of the order of the components in a type; Runciman and Toyn provide independence from the number of arguments. In both cases, with certain restrictions, their ordering relations over polymorphic types correspond well to our intuitive notions of type generalization and specialization. Both provide efficient retrieval mechanisms, but are limited to types as the basis for conceptualization. Selection is then presumably limited by the type orientation as well. Correct use of the components is partially determined by the underlying type system. Zaremski and Wing [21] have subsequently identified a small set of primitive function matches that can be combined a number of useful ways, including ways that implement the two approaches above. Moreover, signature matching is extended to include modules as well.

Independently (but shortly after our initial Inquire prototype [11]), Rollins and Wing [16] experimented with the use of λProlog as the specification and query language, utilizing λProlog's built-in higher-order unification to do specification matching. They specified each Standard ML function or abstract type following Wing's two-tiered approach in Larch [6].[3] Their work, obviously, is the closest in intent to ours.

3. The Shape of Inquire

Inscape provides a different approach to the problem of conceptualization and naming from the approaches discussed above (with the exception of Rollins and Wing). The specification logic of Instress enables the designer to define the concepts (represented by the predicates) in a formal logic notation with various forms of dependencies on other concepts. Thus the complex inter-relationships among concepts are formal logical relationships. This approach to conceptualization as formal logical definitions and relationships is both a strength and a weakness: a strength because of the precision and expressivity that is attainable; a weakness because of the undecidability of the underlying logic. Finding ways of ameliorating the effects of intractability problems by means of engineering trade-offs, to strike a balance between rigor and practicality, is one of the primary research activities in the Inscape experiment.

3. We note that Larch is among the intellectual roots of Inscape. Instress is similar to Larch's interface languages. See [8] and [9] for more details.

A further advantage is accrued in Inscape because of the managed connection between interface specifications and implementations. In contrast to LaSSIE, Inscape provides direct connections between the conceptualizations and the source code. When the concepts themselves are changed, the implications of those changes are propagated to their use sites and the implications of those changes determined at those places. When the interfaces are changed so that the objects behave differently, (that is, they reflect a different conceptualization), the changes are propagated to the use sites and implications of those changes determined there as well. Finally, when an implementation is changed, the implications for the advertised interface and its conceptualization is determined. See [10] for a discussion of the rules for composing fragments and the creation of the composed fragment interfaces.

The formal interface specifications, then, are the medium of conceptualization and are the basis for retrieval and selection. A mixture of interactive browsing and retrieval coupled with Inscape's constructive use of the interface specifications will lead to appropriate use.

We present a general view of Inquire's browsing and retrieval mechanism in the next two subsections.

3.1 Syntactic Browsing

Where the browsers described earlier require auxiliary data structures for their support, Inquire requires only the symbol table maintained by Inscape and used to support its facility of change propagation. The symbol table contains references to the local definition and use sites and maintains both an import list of those objects used locally but defined elsewhere, and an export list of external modules that import the local module specification. As a result, syntactic browsing is merely the traversal of the the symbol table reference links.

The primary difference in syntactic browsing in Inscape from that of other systems lies in the fact that predicates are an important part of the syntactic structure. The usual browsing along functional flow and data flow lines is supplemented by that of predicate flow. That is, Inquire's syntactic browsing enables one to navigate along the flow of behavioral properties from one object to another and to investigate the definitions, uses and dependencies of predicates.

There are four basic syntactic browsing commands.

FIND-DEFS — Finds all definitions of a type, constant, variable, predicate, operation, or module[4] with the selected name and displays the list of definitions in an auxiliary window.

FIND-USES — Finds all uses of a type, constant, variable, predicate, operation, or module and displays the names of the operations where the uses occur in an auxiliary window.

SHOW — SHOW works in conjunction with FIND-DEFS and FIND-USES and requires the qualified name (that is *modulename.itemname*) of the item as input. Clicking on one of the names in the auxiliary window (generated by the find commands) supplies the chosen qualified name to SHOW. SHOW then highlights the use site.

VISIT — VISIT is analogous to SHOW, except that clicking on one of the names in the auxiliary window opens a window for that particular object and displays the object in a read-only manner. The user may then perform any of the read-only mode editing commands on the object.

Thus, Inquire supports a coarse-grained discovery and conceptualization process by means of syntactic-based browsing that is similar to standard browsing facilities but extended by the existence of predicates as part of that syntactic structure.

3.2 Predicate-Based Retrieval

Where syntactic-based browsing is envisioned primarily for the benefit of the programmer's discovery process, predicate-based retrieval is envisioned as useful to both the programmer and the environment. Where browsing allows the programmer only the following of a single object at a time, the retrieval mechanism enables the programmer and the environment to look for either single or multiple objects that provide the designated collections of predicates.

For the Inscape environment, Inquire serves a more specific purpose: finding potential objects to be used as solutions to detected problems in implementations. For example, in the construction of implementations,[5] Inscape enforces the following general rule: every precondition and obligation must be either satisfied within the implementation or propagated to the encompassing interface. Inscape determines that the rule has not been satisfied by the presence of non-empty precondition ceilings and obligation floors. These non-empty sets indicate that there are predicates that have not been satisfied but which cannot be propagated to the interface. Precondition ceilings and obligation floors represent semantic errors — they are unsatisfied semantic requirements imposed by the interface specifications. We intend to use Inquire's retrieval mechanism to find objects that provide predicates to satisfy those ceilings and floors.

The advantage of Inquire over keyword retrieval mechanisms is that, like LaSSIE, it provides inferencing as part of the

4. Currently, Inscape only allows a flat name space for predicates and modules — that is, all predicate and module names must be unique. For variables, types, and operations, we allow the duplication appropriate to the specific language. (This is possible primarily because of scope rules).

5. The rules of composition and propagation (see [10] for a complete discussion) use simple statements (assignment and function calls) as the basis for the composition of complex language statements (sequence, selection and iteration). Thus, the interface of an operation is derived from the interface of its implementation sequence which is in turn derived from the interfaces of its sequents, etc.

retrieval. Inquire uses Inscape's underlying inferencing mechanism to retrieve objects that have exactly the predicates desired, objects with predicates that can be inferred from the desired predicates (within the limits of the inferencing mechanism), or objects with predicates from which to infer the desired predicates (again, within the limits of the inferencing mechanism). The advantage of Inquire over existing retrieval mechanisms in general and LaSSIE in particular is that the concepts involved in the retrieval are directly connected to the objects of retrieval and evolve with the objects themselves.

In the succeeding three subsections, we consider three kinds of retrieval entities, each of which has its own structure: operations, data objects and modules. For each kind of entity, we divide the retrieval into three distinct phases: query, retrieval, and sorting (either independent of, or dependent on, some context). We present first the query and retrieval commands, and then discuss the possible orderings of the results. In the interest of brevity, we present the commands for the retrieval of operations in full detail and elide the discussions of the commands for data objects and modules.

3.2.1 Operations

The specifications of operations in Inscape define the externally visible behavior in terms of a set of assumptions[6] (defined by a set of preconditions) and a set of results, some of which are considered successful and some of which are considered exceptional. Results are defined by a set of postconditions (predicates that are true as a result of executing the operation) and a set of obligations (predicates that must eventually be satisfied; for example, allocating a buffer entails eventually deallocating it). Inquire thus focuses on the external behavior as the conceptual means of finding operations.

OPERATION-QUERY-START — This command creates a window containing a query template, into which the user will enter preconditions, postconditions, and/or obligations to specify the behavior of the operations to be found. Relationships between objects and predicates are indicated by using the same argument in multiple predicates. Multiple occurrences of the same argument will trigger unification in the query processing. After entering the query, the user must choose one of the two commands below to perform the query.

SHOW-OPERATIONS — This command is given after entering a query into the window created by the OPERATION-QUERY-START command. It finds all of the operations that require all of the preconditions and provide all of the postconditions and obligations specified in the query. It finds only cases where a single operation provides all of the requested behavior.

SHOW-OPERATION-SETS — This command, like SHOW-OPERATIONS, is given after entering a query. It finds all ''minimal'' sets of operations that require all of the preconditions and provide all of the postconditions and obligations specified in the query. In this case, ''minimal'' means that for each set, the removal of any operation from the set would result in a set that does not satisfy at least one of the specified predicates.

If, however, there is no way to satisfy the query completely, resulting sets will be returned that satisfy the maximum number of the designated predicates.

After the search has been completed, one may sort the list of operations in various orders. Each ordering is, again, intuitively based on some simple software engineering principle and is intended to allow the programmer, or the Inscape Environment itself, to examine possibilities ordered by a desired ''productiveness criterion''.

MIN-OPERATIONS — *Minimize the number of operations needed to supply the desired set of predicates*. This ordering finds the ''simplest'' way of satisfying the desired behavior.

MIN-PREDICATES — *Minimize the total number of predicates*. This ordering finds the alternative that comes closest to providing exactly what we asked for.

MIN-IMPORTS — *Minimize the number of imports*. This ordering finds the alternative that capitalizes on currently imported modules, or that requires the minimum number of modules.

MIN-EXTRANEOUS — *Minimize the number of extraneous operations*. This ordering finds the alternative that carries the least ''baggage''. MIN-EXTRANEOUS differs from MIN-OPERATIONS in that the latter is concerned only with minimizing the number of operations to satisfy the query while the former is concerned with minimizing the number of operations implied or entailed by the operations that satisfy the query. For example, because of preconditions and obligations, one operation that satisfies the desired set of predicates may require two other operations to work properly while another may require only one; the latter may be preferable to the former.

MIN-CHANGES — *Minimize the number of changes to existing code*. This ordering finds the alternative that requires the least disturbance to the context in which it is to be used — that is, it tries to make as much use of what is already known in the existing code.

The first two orderings are independent of any context other than the search criteria while the last three require contexts of various sorts. Minimizing the imports requires knowledge of the context of the intended use; minimizing the operation ''baggage'' requires knowledge of the retrieved operation's context; and minimizing the number of changes requires the

6. Some of these preconditions may represent assumptions that must be true (designated in the example below by [A]), some may represent assumptions that are validated within the implementation (designated by [V] and which result in exceptions if not true), and some may represent dependent assumptions (represented by [D]) which are not known to be true until some action is performed (such as a record being readable from the disc; these may also result in exceptions).

context of the intended use site.

3.2.2 Data Objects

We use objects of the same type in varying ways and with differing intent. As most type systems provide at best a rough approximation to the intent with which objects are used, Inscape provides for the addition of arbitrary properties to be used as further type constraints and as specifications of relationships to other data objects. Inquire thus focuses on the specified properties of types, variables, and constants as the means of retrieving data objects.

The retrieval operations OBJECT-QUERY-START, SHOW-OBJECTS and SHOW-OBJECT-SETS are analogous to those retrieval operations described above for operations. Similarly, the sorting functions MIN-OBJECTS, MIN-PROPERTIES, MIN-IMPORTS, MIN-EXTRANEOUS and MIN-CHANGES are analogous to those on operations.

3.2.3 Modules

Finding modules can be done one of two ways: first, by doing a query for either a data object or an operation and then using the browsing mechanism to find the modules where those objects are located; second, by using a query facility that combines the two previous facilities.

Again, the retrieval operations MODULE-QUERY-START, SHOW-MODULES and SHOW-MODULE-SETS are analogous to those for operations. Similarly, the sorting functions MIN-MODULES, MIN-CONCEPTS and MIN-IMPORTS are also analogous to those on operations.

4. An Example of Browsing and Retrieval

Suppose that we want to write a program that creates some data and saves it on secondary storage. We will walk through how one might do this using Inquire's browsing and retrieval facilities. While the example is extremely simple and would in reality only require perusal of the specification of the module `FileManagement`,[7] it is indicative of how one would use Inquire in the context of building and evolving a large system.

To get started, we use the command FIND-DEFS and SHOW to look at the specification for `WriteRecord`.

```
WriteRecord(
            <in> fileptr fp,
            <in> recordnumber rn,
            <in> unsigned int n,
            <inout> buffer b
            ) returns int status
```

Synopsis:
n bytes in the buffer b are written to record r in the file reference by fp and the buffer is then deallocated.
Preconditions:
[A] ValidFilePtr(fp)
*[A] FileOpen(*fp)*
[A] BufferAllocated(b)
[V] BufferSizeSufficient(n)
[V] LegalRecordNumber(rn)
*[D] RecordWriteable(*fp, RecordNumbered(rn))*
{Successful result}
Determined by: `status' == 0`
Synopsis:
The record was written successfully to the file.
Postconditions:
BufferSizeSufficient(n)
BufferDeallocated(b)
b' == 0
LegalRecordNumber(rn)
*RecordWritten(*fp, RecordNumbered(rn))*
Obligations:
<none>

At this point we might well browse through the definitions and uses of some of the predicates used in the specification using FIND-DEFS, FIND-USES and SHOW. For example, let us consider *LegalRecordNumber*. Using the browsing commands we find the definition of that predicate.

LegalRecordNumber(recordnumber rn)
Definition:
and: 0 <= rn
rn <= MaxRecordNumber()

By chaining through definitions and uses of predicates we build our conceptual model of the computational elements at our disposal.

Another way in which we might find out about a particular predicate (or domain abstraction, if you will) is to query for objects that use that predicate as a property for either a type, constant or variable. The OBJECT-QUERY-START command produces a template to fill out with the desired properties.

Properties:
LegalRecordNumber(NR)

We then issue the command SHOW-OBJECTS. Inquire returns a list of data objects containing the types `file`, `record`, and `recordnumber`. *LegalRecordNumber* is part of the definition of *RecordNumberUnique* which is a property of both `file` and `record`, and is the property of `recordnumber`.

Type: `primitive file`
Properties for each file f:
FileNameUnique(f)
each record r:
RecordNumberUnique(f, r)
Synopsis:
The structure of the type file is abstracted away for the interface, hence its primitive definition. An abstract notion of a file is needed to correlate the references of filename and fileptr.
Type: `unsigned int recordnumber`
Properties for each recordnumber rn:
LegalRecordNumber(rn)
Synopsis:
A record number is the subrange 0 `. .` MaxRecordNumber of unsigned integers, and

7. See [9] for an elided version of the module specification. Full versions of the module specification are available from the first author.

denotes the corresponding record in a file.

After having explored various domain-specific abstractions represented by the defined predicates, we return to finding out how to use `WriteRecord` properly. The first two predicates in the precondition list refer to the first parameter and indicate that one needs a valid file pointer and an open file for the operation to work properly. The quickest way to find out how to satisfy these two predicates is to retrieve all the operations that provide them. Using the command OPERATION-QUERY-START, filling in the two predicates as postconditions,

Preconditions:
<None>
Postconditions:
ValidFilePtr(FP)
*FileOpen(*FP)*
Obligations:
<None>

and issuing the command SHOW-OPERATIONS, Inquire returns a list with only one operation, `OpenFile`. Browsing `OpenFile` as we did `WriteRecord` provides the following specification.

```
OpenFile(
        <in> filename fn,
        <out> fileptr fp
        ) returns int status
```

Synopsis:
If a file exists with the file name in fn, then the file is opened for I/O and a pointer to the file is returned for all further I/O operations.
Preconditions:
[V] LegalFileName(fn)
[V] FileExists(FileNamedBy(fn))
{Successful result}
Determined by: `status' == 0`
Synopsis:
The file named in fn is open for i/o and fp references that file.
Postconditions:
LegalFileName(fn)
FileExists(FileNamedBy(fn))
*FileOpen(*fp')*
ValidFilePtr(fp')
*FileNamedBy(fn) == *fp'*
Obligations:
*FileClosed(*fp')*

We note that we need to have a legal file name for an existing file for `OpenFile` to work properly. We then query for these two predicates, *LegalFileName* and *FileExists* using SHOW-OPERATION-SETS (as a variation in our exploration). This results in a list of three operations: `FileExists`, `OpenFile`, and `CreateFile`. We already know about `OpenFile`, so we browse through `CreateFile` and find out that it requires that the file not exist as a condition of proper execution.

```
CreateFile(
          <in> filename fn
          ) returns int status
```

Synopsis:
If no file exists with the file name in fn, then a file is created with the file name specified by fn.
Preconditions:
[V] LegalFileName(fn)
[V] not: FileExists(FileNamedBy(fn))
{Successful result}
Determined by: `status' == 0`
Synopsis:
A file has been created with the whose unique name is the value of fn.
Postconditions:
LegalFileName(fn)
FileExists(FileNamedBy(fn))
FileEmpty(FileNamedBy(fn))
FileNameUnique(fn)
Obligations:
<none>

We then query for the predicate *not FileExists* and get back the list of one operation, `FileExists`. Thus, we see that we must determine by means of `FileExists` whether the file exists or not and either open the file if it does, or create and then open the file if it does not exist.

There is one remaining problem. Since we have opened the file, we are obligated to close it. Querying for the postcondition *FileClosed* yields the solution `CloseFile`.

5. Summary

Supporting the use and reuse of software components is a difficult problem. Existing systems have, in general, emphasized efficient retrieval at the expense of conceptualization and provided little help with selection and correct use. In our experiment in Inquire we have taken a different approach to these tradeoffs: we have emphasized conceptualization considerations with the attendant advantage of significant support for selection and correct use.

The contributions of our work in Inquire are tightly coupled with those of Inscape and are dependent on them for its leverage in providing a solution to the problems of component use and reuse. Inquire's novel approach derives from the facts that

- The formal module interface specifications are the medium for conceptualization of system components;
- Conceptualization evolves as the system evolves because of the environmentally managed interconnection between the interfaces (that is, the conceptualization of the components) and the implementation;
- Retrieval, selection and correct use are based on the interface specifications (that is, on the conceptualization medium); and
- Coarse-grained discovery and conceptualization are extended via browsing along not only data and control flow, but also predicate flow.

The browsing and operation query, retrieval, and default sorting facilities (for the successful case,[8] not for exceptions)

have been implemented in the Inscape Prototype. The query, retrieval and sorting facilities for data objects and modules have not yet been done because the simplified prototype specification language has only primitive facilities for data objects (their type names, but no properties). The query and retrieval mechanism works well because of the prototype's simplified and decidable specification logic. This simplification of the specification logic does however limit the expressiveness of the predicates in providing conceptualization.

The next step is to port the Inquire prototype to the full Instress (module interface specification language) Editor. This porting entails extending Inquire to include the full set of data object, operation (with exceptions), and module facilities. For Inquire to work using the full specification logic, we will need to improve the efficiency and richness of the inference mechanism in Inscape.

References

[1] D. G. Belanger, R. J. Brachman, Y. F. Chen, P. T. Devanbu, and P. G. Selfridge. ''Toward a Software Information System'', *AT&T Technical Journal*, 69:2 (March/April, 1990). pp 22-41.

[2] Y. F. Chen, and C. V. Ramamoorthy. ''The C Information Abstractor'', *Proceedings of COMPSAC*, October 1986, Chicago Il.

[3] Premkumar Devanbu, Ronald J. Brachman, Peter Selfridge, Bruce W. Ballard. ''LaSSIE -- A Knowledge-Based Software Information System'', *Proceedings of The 12th International Conference on Software Engineering*, 26-30 March 1989, Nice, France.

[4] W. B. Frakes and P Gandel. ''Representing Reusable Software'', *Information and Software Technology*, November 1990.

[5] W. B. Frakes and B. A. Nejmeh. ''An Information System for Software Reuse'', *Proceedings of the Tenth Minnowbrook Workshop on Software Reuse*, 1987. pp 142-151.

[6] J. V. Guttag, J. J. Horning, and J. M. Wing. ''The Larch family of specification Languages'', *IEEE Software* 2:5 (September 1985). pp 24-36.

[7] Charles W. Krueger. ''Software Reuse'' *ACM Computing Surveys* 24:2 (June 1992). pp 131-184.

[8] Dewayne E. Perry. ''Software Interconnection Models'', *Proceedings of The 9th International Conference on Software Engineering*, March 30 - April 2, 1987, Monterey CA. pp 61-69.

[9] Dewayne E. Perry. ''The Inscape Environment'', *Proceedings of The 11th International Conference on Software Engineering* 15-18 May 1989, Pittsburgh PA. pp 2-12.

[10] Dewayne E. Perry. ''The Logic of Propagation in the Inscape Environment'', *Proceedings of SIGSOFT '89: The Third Testing Analysis and Verification Symposium*, December 1989, Key West FL. *Software Engineering Notes* 14:8 (December 1989).

[11] Dewayne E. Perry and Steven S. Popovich. ''Inquire: Predicate-Based Use and Reuse''. Specification Driven Tools Conference, AT&T Bell Laboratories, October 1989.

[12] Dewayne E. Perry and Carol Stieg. ''Software Faults in Evolving a Large, Real-Time System: a Case Study'', *Proceedings of the European Software Engineering Conference — 1993*, Garmisch, Germany, September 1993.

[13] Dewayne E. Perry and Alexander L. Wolf. ''Foundations for a Study of Software Architecture'', *ACM SIGSOFT Software Engineering Notes* 17:4 (October 1992). pp 40-52

[14] Ruben Prieto-Diaz. ''Classification of Reusable Modules, in *Software Reusability. Volume I. Concepts and Models*, edited by Ted J. Biggerstaff and Alan J. Perlis. New York: ACM Press, 1989. pp 99-123.

[15] Mikael Rittri. ''Using Types as Search Keys in Function Libraries'' *FPCA '89. The Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, 11-13 September 1989. pp 174-183.

[16] Eugene J. Rollins and Jeannette M. Wing. ''Specifications as search keys for software libraries'', *Conference on Functional Programming Languages and Computer Architectures*, September 1989. pp 174-183.

[17] Colin Runciman and Ian Toyn, ''Retrieving re-usable software components by polymorphic type''. *FPCA '89. The Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, 11-13 September 1989. pp 166-173.

[18] P. Selfridge. ''Integrating Code Knowledge with a Software Information System''. *Proceedings of the 5th Annual KBSA Conference*, Syracuse, NY, Sept. 1990.

[19] Joseph L. Steffen. ''Interactive Examination of a C Program with Cscope'', *USENIX Winter Conference Proceedings*, Dallas 1985, pp 170-175.

[20] W. Teitelman. *The INTERLISP Reference Manual*, Bolt, Beranek and Newman, 1974.

[21] Amy Moormann Zaremski and Jeannette M. Wing. ''Signature Matching: A Key to Reuse'', *ACM SIGSOFT'93: Foundations of Software Engineering, 1993 (FSE93)*, Redondo Beach, CA, December 1993.

8. In the prototype, there is only a single result specified. In the full specification language, one may specify any number of successful and exceptional results.