The Inscape Environment

Dewayne E. Perry
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

The Inscape Environment is an integrated software development enviroment for building large software systems by large groups of developers. It provides tools that are knowledgeable about the process of system construction and evolution and that work in symbiosis with the system builders and evolvers. These tools are integrated around the *constructive use* of formal module interface specifications. We first discuss the problems that Inscape addresses, outline our research strategies and approaches to solving these problems, and summarize the contributions of the Inscape Environment. We then discuss the major aspects of the Inscape Environment: the specification language, system construction, system evolution, use and reuse, and validation. We illustrate these various components with examples and discussions.

1. Introduction

The Inscape Environment is an integrated software development environment for building large software systems by large groups of developers. The integration of the components in the environment occurs at two levels: at the lower level, they share a common internal representation (one typical meaning of the term ''integrated environments'') and a common user interface (another common meaning); at the higher level, the components are integrated around the *constructive use* of formal module interface specifications.

Formal specification projects in general have emphasized the formal properties of specifications (as, for example, in Larch [8, 9]) and the problems of verification with respect to those specifications (as, for example, in the OBJ system [4, 5]). Little emphasis has been given to their practical use.[1] Guttag, Horning and Wing [7] state ''We found it difficult to establish a useful correspondence between our elegant formal systems and the untidy realities of writing and maintaining programs''. It is the thesis of this project that there is a useful correspondence between *interface* specifications and the untidy realities of writing and maintaining programs. Illustrating what we mean by ''the constructive use of formal interface specifications'' and its relationship to the ''untidy realities of writing and maintaining programs'' is the purpose of this paper.

A slight digression first into the name of the project. The terms *inscape* and *instress* are taken from the poet Gerard Manley Hopkins because they are metaphorically suggestive. W. H. Gardner [2] states

> *As a name for that ''individually-distinctive'' form . . . which constitutes the rich and revealing ''one-ness'' of a natural object, he coined the word inscape; and for that energy of being by which all things are upheld, for that natural . . . stress which determines an inscape and keeps it in being — for that he coined the word instress.*

As we are concerned with the form of programmed objects, it seems appropriate to call the environment *Inscape*; and as the environment is based on the premise that interface specifications provide the ''glue'' for programmed objects, the specification language is called *Instress*. For obvious reasons, the other components of Inscape all begin with the prefix ''In''.

In the remainder of the introduction, we delineate the problems addressed by the Inscape Environment, the research strategies and approaches that we have taken in Inscape, and summarize the important research contributions made as part of our environmental research. We then discuss various aspects of the Inscape Environment: the module interface specification language, system construction, system evolution, use and reuse, and system validation. Finally, we present a short summary of the Inscape Environment.

1.1 Problems Addressed

Inscape addresses two fundamental problems in building software systems: evolution and scale.

Dividing the life-cycle into two distinct phases, development and maintenance, introduces a distinction that is not born out in practice. The entire life-cycle is iterative: requirements grow and evolve as understanding of the problem increases; the design evolves as design decisions constrain the solution space; these design decisions have backwards-reaching effects on the requirements, both from finding inconsistencies and lack of clarity as well as from indicating costs that may generate reconsideration of the requirements and changes to them; and so on throughout the remainder of the cycle. A system evolves in all its parts from beginning to end. Trying to force this evolutionary process into neat, distinct and independent phases only serves to obscure the reality of software development.

In considering the problems of scale, there are several important subproblems: complexity, programming-in-the-large, and programming-in-the-many. How does one help to manage the problems of complexity, especially as the complexity of systems seems to explode with the increase in their size? What facilities are needed for building systems out of components that are separately developed? How do you manage the problems of ''crowd control''? What policies should be incorporated into the supporting environment to manage the interactions of the various individuals and groups doing a variety of different activities?

These are the problems that we address in our research. The Inscape Environment is the context in which we explore solutions to these problems.

[1] An exception to this lack of practical application is Moriconi's attempt in his Designer/Verifier's Assistant [19] to provide ''a technique for identifying mechanically what formulas must be proved to determine the exact effects of change. This approach proved impractical because the formulas to be proved were in an undecidable theory, causing this problem to be no easier than the verification problem in general'' [20]. Inscape embodies a similar conceptual framework, but attempts to overcome the practical shortcomings of Moriconi's approach by a number of restrictions that are discussed subsequently in this paper.

©1989 ACM 0270-5257/89/0500/0002$00.75

Recommended by: Barry Boehm

1.2 Research Strategies and Approaches

There are three basic approaches that we take in the Inscape Project: we formulate models of various parts of the system; we formalize parts of the process; and we explore various trade-offs to make intractable problems more manageable. Specifically, we

- consider models of software interconnections [26] and propose a new model (semantic interconnections) that forms the underlying basis of Inscape;
- develop a model of software development environments (SDEs) [28] to delineate the important aspects of SDEs and to characterize important aspects of SDEs with respect to the problems of scale;
- formalize various aspects of the software process (specifically those of system construction and evolution);
- make practical use of specification and verification technology by using formal interface specifications constructively; and
- incorporate the understanding of specifications, the development process, and the programming language into the environment.

Our research strategy is to proceed along several dimensions:

- systematically work out the implications of using formal interface specifications as the integrating ingredient in a software development environment that supports the development of large systems with a large number of developers;
- make the use of specifications practical — that is, "dance" around the "tarpit" of verification: restrict the power of the specification language; use shallow consistency checking (that tends towards pattern matching and simple deductions); and automate as much as possible, interacting with the user where automation is not possible;
- use incremental techniques to distribute the cost of analysis — incremental in two senses: incrementally at a fine grain — for example, on a statement by statement basis interactively with the user; and incrementally on a large grain — because of the separation of interface and implementation, we can follow the intuition in Alphard [30], and analyze each implementation independently using interfaces as the basis for analysis.

Our goal, then, is to provide a practical application of formal methods in building large, evolutionary software systems with large groups of people.

1.3 Research Contributions of the Inscape Environment

The novel features of the Inscape Environment are as follows:

- introducing notions of obligations and multiple results in the interface language Instress;
- using predicates as units of interconnection forming a semantic interconnection structure that is the basis for Inscape's analysis;
- enforcing the consistent use of interfaces, thereby detecting and preventing interface errors as they are made (within the constraints of Inscape's shallow consistency checking);
- determining the implications of changes to both interfaces and implementations at the levels of predicate satisfaction and propagation, and exception handling and propagation;
- providing the policies of enforced and voluntary cooperation with supporting mechanisms (change propagation and simulation) and structures (hierarchical databases and workspaces);
- supplementing the static analysis with specification-based test-case description generation and integration-test management;
- using predicate-based browsing and system search techniques; and
- formalizing various important version management concepts.

The discussion of these various aspects of Inscape will clarify their contribution and their effect on the environment and the process of developing and evolving large systems.

2. Module Interface Specifications

The primary purpose for using formal module interface specifications is to provide a means of presenting all the information that a programmer needs to know about using any particular module in a clear and concise way — that is, capturing what the designer had in mind when the module was created so that the module can be used in ways consistent with those intentions. To accomplish this purpose, *Instress* provides facilities for defining predicates and for describing the properties of data and the behavior of operations[2] (in terms of these predicates). In addition, Instress provides facilities to supply kinds of pragmatic information (such as recommendations for recovery from exceptions). Thus, by specifying the semantics of the module interface and including pragmatic information, the designer can define precisely the meaning of a module and indicate ways in which it can be properly used.

Our approach is based on that of Hoare's input/output predicates [12] (see also [18, 30]) but extended in two ways: a set of obligations is added to the set of postconditions to form a result of an operation; and multiple results are provided to allow the description of both normal and exceptional exits from the operation. We use this predicate approach in Instress rather than an algebraic approach (see [4, 5, 6, 9]) because it seems better suited to specifying exceptions and obligations that may occur and what they mean when they do occur.[3] Further, we demonstrate below how this approach enables us to build a semantic interconnection structure that reflects the intent of the programmmer in building and evolving the system.

The following specification of a file management module, Example 1, provides a sample of the type of specification that can be built in Instress. This example is used throughout the ensuing discussion to illustrate various aspects of Inscape and the uses that can be made of these kinds of specifications. In particular, this example illustrates the essential details about the module interface:

- the vocabulary of the abstraction provided by the module,
- the data objects and their properties,
- relationships among data objects,
- the operations and their effects and side-effects,
- the interrelationships among different operations,
- the interrelationships among operations and data,
- exceptions and their effects, and
- the minimal handling of exceptions.

We first present the example, eliding details that are either repetitive or not essential to the subsequent discussion. Next, we discuss predicates and the logical language used to define them, data objects (types, constants, and variables), their definitions and descriptions, and then the specifications of operations (procedures and functions). Finally, we discuss various views of the specification and the analysis that Instress performs as the specification is built.

Please note that we have taken great pains to make the formality of the specifications accessible to the practicing programmer — that is, both easy to read and easy to understand while still having the necessary formal information.

[2] For the present, specifications are limited to sequential programs. This limitation is imposed to bound the problems that need to be solved. Given the method of describing the behavior of functions and procedures, it does not seem to be too great an extension to include concurrency of the form provided in either Ada [1] or Concurrent C [3] (since the notions of entries and transactions are analogous to functions and procedures, with additional semantics to cover aspects of concurrency).

[3] See [17] for a discussion of the relative strengths and weaknesses of various specification techniques.

2.1 Example 1: The Specification of `FileManagement`

This view of the module specification is a sample of Instress' publication view. Programming language entities are in constant-width font, semantic entities are in italics, and supplementary comments are in regular font.

2.1.1 Predicate Specifications

Predicate specifications are the means of creating the vocabulary necessary to describe the semantics of the data objects and operations in the module — that is, to express the abstraction provided by the module. The general intuition about predicates is that similar to the intuition of algorithmic abstraction: complex logical formulas are encapsulated in predicate definitions to provide a useful abstraction that is both easy to understand and easy to use. This abstractive and simplifying use of predicates is an important aspect of Inscape in trying to make formal methods and techniques available as a tool for the general programmer.

In Example 1, we find illustrated the general form of predicate specifications: a typed predicate declaration with both formal and informal definitions. The informal definition is provided to aid the reader's intuition about the formal definition. It is obvious that only the formal definition can be used by the analysis mechanisms, so that extreme care should be taken to ensure that the informal description exactly matches the formal one.

The logical language of Instress is one of the primary focuses of our current research. The question is how to suitably restrict the language. One possibility is to restrict quantification to range over finite sets of data objects. For example, the definition of the predicate *LegalFileName* contains the quantified expression that describes a property of each character in the file name: each character must be alphabetic; no special characters are allowed. The quantification is limited to the characters in the file name.

Some predicates are defined in terms of other predicates, either in the same module or in supporting modules. For example, the predicate *BufferSizeSufficient* would be declared by predicates defined in the memory management module. Some predicates, on the other hand, are introduced as *primitive* predicates, not because there is not an underlying definition, but because those details are abstracted from the interface and the user. The predicate *FileOpen* is such a predicate. The implementation meaning of the predicate denotes the installation of file details in main memory to make subsequent file operations more efficient. Those details are not relevant to the user. What is relevant, however, is that *FileOpen* and *FileClosed* are mutually contradictory predicates (see the definition of *FileClosed*). These definitional relationships and the subsequent use of the predicates in defining the properties of data objects and the behavior of operations build their meaning within the module interface.

2.2 Data Specifications

There are three kinds of data objects to be found in module interfaces: types, constants, and shared variables. Type specifications define the basic meaning for data objects — that is, they define those properties that are common of all objects of that type. Intuitively, variables are used for particular purposes and have particular meaning in addition to that provided by the base type. Similarly, constants have a specific purpose to denote particular objects or values and, hence, have a specific meaning beyond that of the underlying type. This is the kind of information that the data object specifications are meant to capture.

In Example 1, the type specification for `filename` illustrates only a subset of the kinds of information that might be specified for a type: the representation of the type (in terms of the programming language representation constructs), an informal description of the type (again, care should be taken that this informal part of the specification should be consistent with the formal part), and the formal properties of the type beyond those inherited from the type's representation (for example, each object of type `filename` must satisfy the predicate *LegalFileName* in addition to satisfying the properties of the type `string`). These formal properties specify aspects of the type that are not expressible in the supported programming language, especially if the language has a weak or inexpressive type system (as, for example, C has). The designer may refine the meaning provided by the type's representation as well as express relationships among data objects that are not expressible in a type system. Other aspects of interest in type specifications are initialization specifications with their resulting properties and obligations, and visibility or access restrictions (such as may be found, for example, in PIC [29]).

Constant specifications define the value for the typed object and may refine the meaning of the type by specifying additional properties for that constant.

Variable specifications define the type of the object, give an informal description of the intended meaning of the variable, and may specialize the type information by adding properties to refine the meaning associated with that particular variable and to express relationships with other data items. As with types specifications, initialization specifications, with their resulting properties and obligations, and visibility or access restrictions may be provided.

2.3 Operation Specifications

Operation specifications describe the abstract interface (that is, external) behavior of functions and procedures. In Instress, this interface behavior is described by a set of preconditions and a set of results. Results are divided into successful and exceptional results. For both sets, results are defined in terms of postconditions (predicates that are guaranteed to be true) and obligations (predicates that the user is entailed to satisfy eventually). For exceptional results, the specifier also supplies guidance about recovery with either an informal comment or a recommendations for a particular recovery routine. As in the previously discussed specification entities, formal descriptions are also provided by the specifier (with the attendant problems of maintaining consistency with the formal parts)

We distinguish three kinds of preconditions: *assumed, validated,* and *dependent preconditions. Assumed preconditions* are those that Hoare [13] talks about when he says "if the assumptions are falsified, the product may break, and its subsequent (but not its previous) behavior may be wholly arbitrary. Even if it seems to work for a while, it is completely worthless, unreliable, and even dangerous." These preconditions are assumed to be true and must be explicitly satisfied. In Example 1 in the specification of `CloseFile`, the designer has chosen to assume that the preconditions *ValidFilePtr* and *FileOpen* are true when the operation is invoked. This fact is expressed in the specification by the precondition annotation "<assumed>".

However, when writing robust, fault-tolerant programs, there are some preconditions (i.e., assumptions) whose falsification leads to wholly predictable results. These are assumptions that are tested before proceeding in the implementation. We call these *validated preconditions.* In the specification of `WriteRecord`, the preconditions *LegalRecordNr, Allocated,* and *BufferSizeSufficient,* are validated first before proceeding to write the data to the file. The annotation "<validated>" indicates this status.

Generally, validation is performed before committing to some computation that may be difficult or costly to undo. The choice between assuming or validating preconditions is often a tradeoff between safety and efficiency. On the other hand, assumed preconditions are often difficult or impossible to verify. In the case of the predicate *RecordIn*, however, it must be an assumed precondition as there is no way of determining, without detailed knowledge of the particular structure of the record, whether the user has put the desired data into the buffer or not.

The third class of preconditions is that whose truth is not known until some point in the computation when an attempt is made do something, such as read a file. It is impossible to know whether the predicate *RecordReadable* is true until the hardware tries to read the record. *Dependent preconditions* are similar to vauuated preconditions but are dependent on, for example, external events, hardware, or other

```
          ***** Module FileManagement *****
Predicates
    LegalFileName(filename F)
        Definition:  NonNullString(F) and
                     each ( i in 1..length(F) ) { Alphabetic(F[i]) }
        Informally:  A legal file name is a non-empty string of
                     alphabetic characters only.

    FileExists(filename F) . . .
    ValidFilePtr(fileptr FP) . . .

    FileOpen(fileptr FP)
        Definition:  primitive
        Informally:  The file is opened for reading and writing.

    FileClosed(fileptr FP)
        Definition:  not FileOpen(FP)
        Informally:  The file is closed for I/O.

    LegalRecordNr(int R) . . .
    RecordExists(int R) . . .
    RecordReadable(int R) . . .
    RecordConsistent(int R) . . .
    RecordWriteable(int R) . . .
    RecordIn(buffer B) . . .
    BufferSizeSufficient(buffer B; int R) . . .
Data Objects
    Type: filename
        Representation: string
        Properties:  each ( filename F ) { LegalFileName(F) }
        Synopsis:    A filename is a non-empty string that is limited to
                     alphabetic characters.

    Type: fileptr . . .
Operations
    int CreateFile(<in> filename FN; <out> fileptr FP)
        Synopsis:    CreateFile creates a file named by FN,
                     automatically opens it, and returns a handle to be
                     used in all subsequent file operations until the file
                     is closed.
        Preconditions:
            LegalFileName(FN)          <validated>
            not FileExists(FN)         <validated>
        Results:
            <Successful result: CreateFile == 0>
                Synopsis: The file named by FN has been created and
                    opened, and the output parameter FP is the
                    handle for subsequent file operations.
                Postconditions:
                    LegalFileName(FN)  ValidFilePtr(FP)
                    FileExists(FN)     FileOpen(FP)
                Obligations: FileClosed(FP)
            <Exception IllegalFilename: CreateFile == 1>
                Synopsis: The file was not created because the file
                    name in FN was invalid.
                Failed: LegalFileName(FN)
                Postconditions: not LegalFileName(FN)
                Obligations: <none>
                Recovery: Ensure that FN has an appropriate string.
            <Exception FileAlreadyExists: CreateFile == 2> . . .

    int OpenFile(<in> filename FN; <out> fileptr FP) ...

    void CloseFile(<inout> fileptr FP)
        Synopsis:    CloseFile closes the file and trashes the file
                     pointer FP.
        Preconditions:
            ValidFilePtr(FP)           <assumed>
            FileOpen(FP)               <assumed>
        Results:
            <Successful result: assumed>
                Synopsis: Assumes the file is open and FP is a valid
                    file pointer; the file is closed and FP is no
                    longer valid.
                Postconditions:
                    FileClosed(FP)       not ValidFilePtr(FP')
                Obligations: <none>

int ReadRecord(<in> fileptr FP; <in> int R;
                 <out> int L; <out> buffer B) . . .

int WriteRecord(<in> fileptr FP; <in> int R;
                  <in> int L; <inout> buffer B)
    Synopsis:  . . .
    Preconditions:
        ValidFilePtr(FP)             <assumed>
        FileOpen(FP)                 <assumed>
        LegalRecordNr(R)             <validated>
        RecordIn(B)                  <assumed>
        Allocated(B)                 <validated>
        BufferSizeSufficient(B, L)   <validated>
        RecordWriteable(R)           <dependent>
    Results:
        <Successful result: WriteRecord == 0>
            Synopsis: The record R has been written in the file
                denoted by FP.
            Postconditions:
                LegalRecordNr(R)    BufferSizeSufficient(B,L)
                Deallocated(B)      RecordExists(R)
            Obligations: <none>
        <Exception IllegalRecordNr: WriteRecord == 1>
            Synopsis: An invalid record number.
            Failed: LegalRecordNumber
            Postconditions: not LegalRecordNr
            Obligations: <none>
            Recovery: . . .
        <Exception UnallocatedBuffer: WriteRecord == 2>
            . . . Postconditions: not Allocated(B) . . .
        <Exception InsufficientBufferSize: WriteRecord == 3> . . .
        <Exception WriteError: WriteRecord == >
            Synopsis: The record has not been written due to an
                I/O error.
            Failed: RecordWriteable(R)
            Postconditions:
                LegalRecordNr(R)    BufferSizeSufficient(B, L)
                Allocated(B)        not RecordWriteable(R)
            Obligations: <none>
            Recovery: RecoverFileSystem

boolean FileExists(<in> filename FN) . . .
    Synopsis:   FileExists determines whether the file named
                by FN exists.
    Preconditions:
        LegalFileName(FN)            <assumed>
    Results:
        <Successful result: FileExists == TRUE>
            Synopsis: The file exists.
            Postconditions: FileExists(FN)
            Obligations: <none>
        <Successful result: FileExists == FALSE>
            Synopsis: The file does not exist.
            Postconditions: not FileExists(FN)
            Obligations: <none>
          ***** End FileManagement *****
```

Example 1: A sample Specification of FileManagement (concurrent) processes.

We note that there is a special relationship between validated and dependent preconditions on the one hand and exceptions on the other: the failure of each validated and dependent precondition must be found in the specification of some exception. For examples see the exception specifications of `CreateFile` and `WriteRecord`.

2.4 Instress Views

One of the primary benefits of using a prototyping mechanism such as the Gandalf's ALOE editor-generation tools [11] is that one can experiment with various concrete syntax representations. The abstract syntax of the specifications can remain constant, but one can choose various ways in which to view the specification. Example 1 represents a multi-font, formatted, publication view. Other views available are an overview of the objects in the interface, an informal view of the objects (showing only the informal comments and synopses), a formal view (showing only the formal definitions of the objects), and various language-dependent views (such as internal and external views for C `.h` files).

2.5 Instress Analysis

Part of the underlying support that Inscape provides the user is the formalization of the development process. In Instress, the process of specification has been formalized in a set of rules about consistency among various objects in the interface and rules about relationships among some of the interface objects. As the specification is built, Instress enforces various rules about consistency and rudimentary completeness. In particular, Instress checks that

- predicate definitions are consistent;
- precondition, postcondition and obligation lists are consistent;
- validated and dependent preconditions are represented as failed conditions in at least one exception; and
- <in> parameters have preconditions and <out> parameters have postconditions.

In addition, while the specification is being built, Instress creates unit, syntactic, and semantic interconnections structures [26] of the interfaces' dependencies on external modules. These interconnection structures are used by subsequent tools in their analysis.

2.6 Benefits: Improved Communication

Formal interface specifications provide improved communication among the developers for a variety of reasons. First, Instress specifications provides a uniform, formal medium for the expression of one part of the design — the module interfaces. A formal medium of the specifications is more precise, specific, clear and complete than current informal methods of documentation. Moreover, there are various views of the specification that are available to the user, ranging from an overview, an informal view, to a strictly formal view of the specifications. Second, the formal document is a better means of negotiation for changes that can then be propagated automatically. Negotiations focus on specific aspects of interface behavior instead of vague descriptions of general functionality. Third, the relationship between the interface specifications and the implementation is managed formally by Inscape. That means that the effects of changes to specifications are automatically projected onto the implementation and that changes to the implementation are projected onto their respective interface specifications. The available information is current. Last, fewer personal interactions are needed because of the formal (and as a result, better) documentation. These means that there is less wasted time because of unsuccessful attempts at personal interaction. Developers are then more independent of each other.

3. System Construction

The program construction editor *Inform* incorporates knowledge of Instress' specifications, the implementation language, and Inscape's rules for program construction to provide interactive, cooperative program construction. The environment maximizes the use of this knowledge to minimize the programmer's effort to construct maintainable systems. During the process of program construction, Inform enforces the consistent use of Instress module interface specifications, enforces Inscape's rules of program construction, and automatically records the dependency relationships and the semantic interconnections determined by the implementation. As the implementation is constructed interactively, the environment determines whether the implementation is complete (i.e., there are no preconditions or obligations that are unsatisfied and unpropagated to the interface) and automatically creates the interface specification from the implementation (which may then be interactively modified according to the requirements of the module's abstraction). If an interface specification already exists, that specification can be compared with the implementation-derived interface specification to determine the correctness of the implementation.

There are two fundamental concepts that are formalized by Inscape in the construction of software systems: the semantic manipulations required in the consistent use of the interface specifications and the constructive satisfaction and propagation of the semantic information in the implementation; and the control flow manipulation required in the handling and propagation of exceptions.

We begin in Section 3.1 by illustrating Inscape's system construction approach with a small example. We then discuss the constructive use of interface semantics in section 3.2, the constructive use of exception specifications in section 3.3, the constructive use of exception specification in section 3.4, generating interface specifications in section 3.5, Inform's views in section 3.6, Inform's analysis in section 3.7, laying the foundation for evolution in section 3.8, and finally discussion some of Inform's benefits in section 3.9.

3.1 Example 2: The Implementation of PutRecord in Inscape

This example shows the publication view of `PutRecord` together with the propagated interface that results from the implementation. Programming language constructs are in bold type, program objects in constant width type, predicates in italics, and annotations in regular type.

Following Example 2 are two graphical views of the implementation: the first is the semantic-interconnection view for the computation's normal (that is, successful) execution; the second is the control-flow view of the implementation with the normal as well as the exception exits.

3.2 Constructive Use of Interface Semantics

The basic building blocks in an implementation are the specified operations. The specifications of these operations are instantiated for each use according to the arguments supplied in the operation invocation. In Example 2, for instance, the specification of `WriteRecord` is instantiated with the arguments `FP, R, L,` and `B` (see the predicates in the list of abbreviations for Figure 1).

The basic unit of implementation is the *sequence*. In Example 2, the basic implementation is a sequence of three statements: the **if** statement and the invocations of `WriteRecord` and `CloseFile`. Figure 1 illustrates how the instantiated interface specifications may be used to satisfy preconditions and obligations, and where the postconditions propagated to the interface come from.

Figure 1 also illustrates how Inform incrementally builds the **if** statement from its component parts and generates the interface for that **if** statement. This constructed interface is then used with the instantiated interfaces to form the interface for the implementation sequence. Thus, Inform builds the implementation in three ways (two of which are illustrated in Figure 1): instantiating interface specifications for operations invoked in the implementation; constructing interfaces from the component parts in complex language statements such as selection and iteration; and interacting with the user where the meaning of programmer actions are not determinable by the environment (as may well be the case in expressions, for example).

3.3 Constructive Use of Exception Specifications

As mentioned in the preceding section on Instress, there is a relationship between some of the preconditions and exceptions: the failure of a precondition may result in an exception. This relationship provides the implementor with some latitude concerning the satisfaction and/or propagation of these preconditions.

We formalize the handling of exceptions and build an understanding of the various ways of handling exceptions into the environment. The following are the formalized ways of handling an exception supported by the environment. An exception may be:

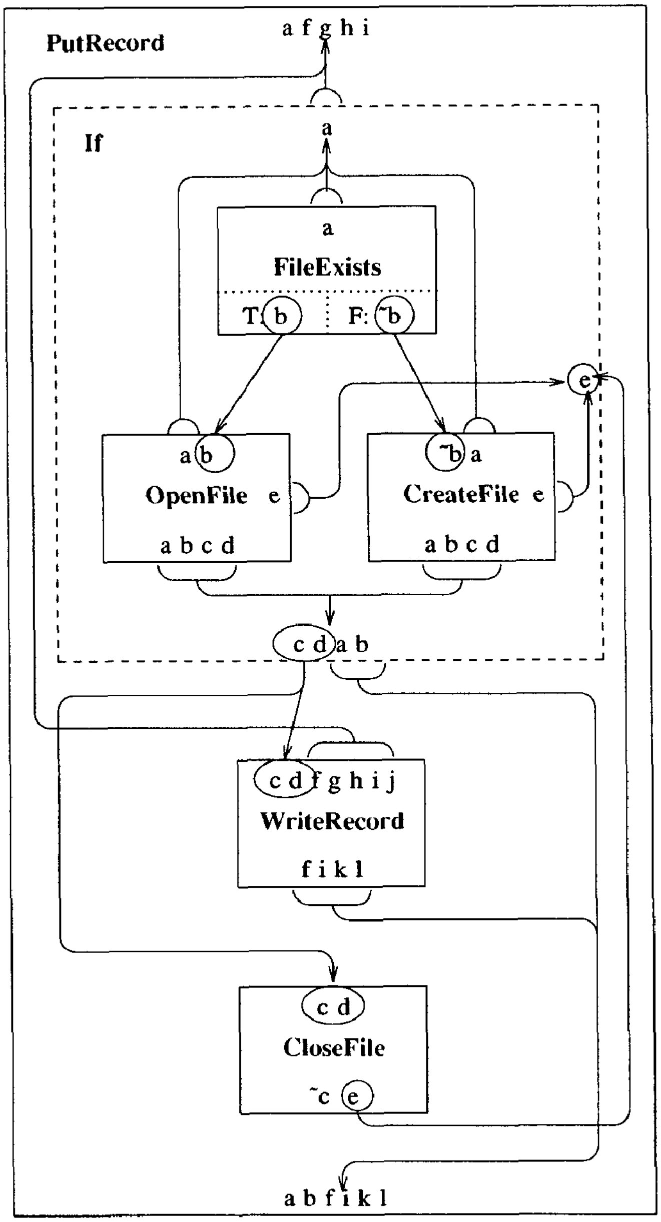
Precluded The associated precondition has been satisfied and there is no need to handle the exception. Figure 2 show this preclusion for the `NonExistentFile` and `FileAlreadyExists` exceptions in `OpenFile` and `CreateFile` respectively.

```
PutRecord(<in> filename FN; <in> int R;
          <in> int L; <inout> buffer B);
  {
  fileptr FP;
  if FileExists(FN)
     OpenFile(FN, FP);
        <exception IllegalFileName pruned>
        <exception NonExistentFile precluded>
  else
     CreateFile(FN, FP);
        <exception IllegalFileName pruned>
        <exception FileAlreadyExists precluded>
  WriteRecord(FP, R, L, B);
     exception IllegalRecordNr        <coalesced>
     exception UnallocatedBuffer      <coalesced>
     exception BufferTooSmall         <coalesced>
        CloseFile(FP);
        return ParameterError;        <propagated>
     exception WriteError
        CloseFile(FP);
        return IOerror;               <propagated>
  CloseFile(FP);
  }
Propagated Preconditions:
  LegalFileName(FN)          Allocated(B)
  LegalRecordNr(R)           BufferSizeSufficient(B, L)
  RecordIn(B)
Propagated Results:
  <normal exit>
     Propagated Postconditions:
     LegalFileName(FN)       BufferSizeSufficient(B, L)
     FileExists(FN)          Deallocated(B)
     LegalRecordNr(R)        RecordExists(R)
     Propagated Obligations: <none>
  <exception ParameterError>
     Propagated Postconditions:
     FileExists(FN)
     not IllegalRecordNr(R) or not Allocated(B)
        or not BufferSizeSufficient(B, L)
     Propagated Obligations: <none>
  <exception IOerror>
     Propagated Postconditions:
     LegalFileName(FN)       Deallocated(B)
     FileExists(FN)          BufferSizeSufficient(B, L)
     LegalRecordNr(R)        not RecordWriteable(R)
     Propagated Obligations: <none>
```

Example 2: Implementation and Propagated Interface

Pruned This method of handling (or, non-handling) is a conscious refusal to handle the exception. The associated preconditions become *assumed* preconditions and must be satisfied or propagated to the interface. In Figure 2, the exception `IllegalFileName` in both `OpenFile` and `CreateFile` has been handled in this way because failure condition, *LegalFileName*, is already an assumed precondition in `FileExists`.

Reported The exception is propagated, possibly with some repair, to the interface. The exception `IOError` is



a: LegalFileName(FN)
b: FileExists(FN)
c: ValidFilePtr(FP)
d: FileOpen(FP)
e: FileClosed(FP)
f: LegalRecordNr(R)

g: RecordIn(B)
h: Allocated(B)
i: BufferSizeSufficient(B, L)
j: RecordWriteable(R)
k: Deallocated(B)
l: RecordExists(R)

Figure 1: PutRecord's Semantic Interconnections

repaired (closing the file), renamed as WriteError, and reported to the interface.

Recovered
The exception is handled by retrying the statement that generated the exception. There may be some repair to increase the likelihood of success.

Repaired
The results of the exception are fixed or compensated for in some fashion — for example, fixed to match the successful results. The control flow then is merged with the normal results.

Ignored
The results of the exception are satisfactory and are merged with the normal results.

Coalesced
The results of several exceptions are collapsed into one exception and propagated to the interface. The `ParameterError` exception reported in `PutRecord` is this kind of exception (see Figure 2). Some repair may be required.

Introduced
An arbitrary condition is considered to be exceptional and is propagated to the interface. There may or may not be repair included.

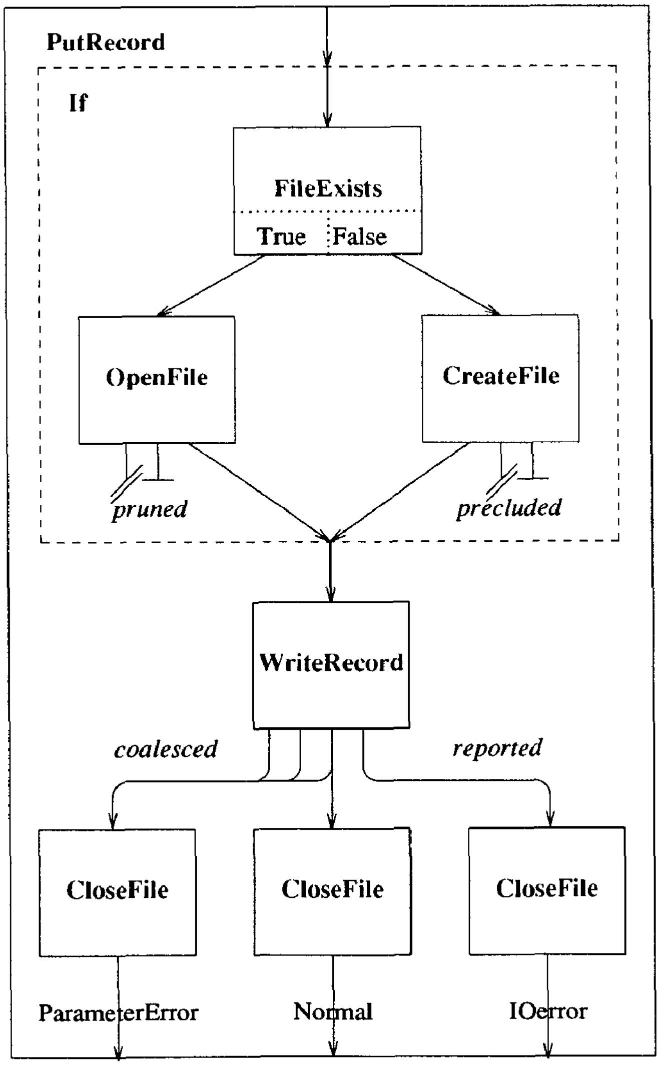


Figure 2: PutRecord's Control Flow

Thus, if a validated or dependent precondition is known to be true (i.e., a postcondition satisfies that precondition), then the associated exception need not be handled because it is known before hand that it will not occur. Correspondingly, if an exception is handled, then the associated precondition need not be propagated to the enclosing interface. Conversely, if the failure of a precondition causes an exception to occur, then that precondition must be exported to the interface as a validated or dependent precondition.

3.4 Generating Interface Specifications

There are various levels of generated interface specifications: the interfaces constructed from the component parts in complex language statements; the interface generated from a sequence; and the operation interface propagated from its implementation sequence interface.

Certain aspects of the interface for the implemented module can be generated automatically by the environment: those preconditions and obligations that have not been satisfied within the implementation and that have been propagated to the interface; and those postconditions that are known to be true at the end of an operation's implementation. However, there are different rules that govern the generation of the different kinds of interfaces: those for generating the interfaces for selection and iteration are more complex than those for generating the interfaces for sequence and operations. For example, in the latter, the interface of the sequence is pruned of all the predicate references to local variables in order to produce the interface for the implementation (Figure 1 shows that *FileClosed* and *not ValidFilePtr* are not propagated to the implementation interface).

However, there are some aspects of the interface that need to be generated interactively with the programmer: the propagation or handling of exceptions; and re-interpretation or introduction of predicates and exceptions at the implementation boundaries to conform with the vocabulary of the abstraction provided by the module interface.

As in the explicit construction of module interface specifications using the Instress editor, the rules governing the completeness and consistency of specifications are enforced here in the generation of interface specifications.

3.5 Inform Views

There are two kinds of views available in Inform: the various contextual views and synthesized views. In the latter category are the two views illustrated in Figures 1 and 2: a graphical semantic interconnection view and a graphical control flow view. The first shows the dependencies and sources; the second shows where and how the various results arise.

The current contextual views are similar to the various views available in Instress — they are various levels of detail of the information contained in the abstract syntax representation of the program (complete with the interconnection information). Inform provides a multiple context view that shows several views simultaneously: the program view, the current specification (instantiated or generated) view (complete with its satisfaction and dependence information), the current-known-postconditions view, the precondition-ceiling and obligation-floor view (those predicates that have not been satisfied and cannot be propagated to the interface), and the propagated-interface view.

3.6 Inform Analysis

Using the predicates defined in the module interface specifications as units of interconnection, Inform enforces the consistent use of the specifications by applying the following basic rule: a precondition or obligation must either be satisfied within the module's implementation, or propagated to the enclosing interface (eventually to be satisfied outside the module's implementation).

It may not be possible, however, to propagate a predicate to the interface. For example, suppose a precondition *P* has not been satisfied (as for example predicate *LegalFileName* has not been in Figure 1) and in the attempted propagation to the interface (intuitively, *up* through the implementation) it encounters a postcondition *not P*. This postcondition forms a logical barrier for the precondition — what we call a *precondition ceiling*. This indicates that the precondition must be satisfied somewhere between where it was incurred and its ceiling. The propagation of obligations proceeds in an analogous fashion (though, intuitively, *down* though the implementation) hitting either an *obligation floor* or making it to the interface.

The interface specification semantics of exception results are those that are known and obligated at the time the result is propagated to the interface. Propagation problems may result during this process. For example, if we had not called `CloseFile` in the reported exception `IOerror`, then we would have had an obligation *FileClosed(FP)* that was not satisfied and could not be propagated to the interface (because it refers to a local variable) resulting in an obligation floor in the exception handler.

The consistency and satisfaction checking that is performed during construction is an augmented version of the checking performed by Instress. Completeness checking varies according to the unit being analyzed.

3.7 Foundation for Evolution

Inform's management of the semantic interconnection detail provides the foundation for Infuse's ability to understand the implications of changes to software constructed within Inscape. While the module is being implemented, Inform records the following details of construction:

- precondition dependence and propagation,
- postcondition satisfaction and propagation,
- obligation dependence and propagation,
- exception preclusion, and
- methods of exception handling.

By recording the ways in which the different elements in the interface are used in the implementation — that is, how exceptions are handled and what functionality (i.e., behavior) is depended on — Inform provides the knowledge necessary for Infuse to understand software changes.

3.8 Benefits: Interface Error Prevention

Because of the semantic analysis performed by Inscape, there is early detection and prevention of interface errors in the construction and evolution of a system. Based on a study of a real-time system (see [22 - 24]) and assuming that this data point is representative, there is the potential for preventing 75% of the interface errors that occur in system test and integration. This amounts to about half of all the errors that occurred in this phase. In addition, interface errors that occur before that phase will be prevented as well. Unfortunately, we do not have data about the magnitude of interface errors that occur before the system test and integration phase.

4. System Evolution

The *Infuse* subsystem [25, 15] manages and coordinates multiple changes by multiple programmers and propagates the changes interactively to guarantee their completeness and consistency. Conformance to Inscape's rules of program construction is enforced throughout the change process.

There are two orthogonal concepts that converge in Infuse:[4] 1) semantic interconnections, and 2) "crowd-control" policies, mechanisms, and structures.

In section 4.1 and 4.2, we discuss the semantic effects of changing interfaces and implementations and ways in which Infuse manages this process. In section 4.3, we describe the basic aspects of Infuse's approach to managing large group interactions. Finally, in section 4.4, we summarize some of the benefits of environmentally-assisted evolution.

4.1 Interface Modifications

There are four basic ways in which Infuse determines the semantic effects of changes to an interface specification. First, Infuse can determine where changes to the interface have no effects. In this case, the implementation is sufficiently rich that it can absorb the effects of the changed interface. For example, if a precondition is added to the interface of an operation, and everywhere the operation is used there exist postconditions known at that point that will satisfy the precondition, then the change will have no effect whatsoever.

Second, Infuse determines where code is no longer needed. For example, if preconditions or obligations are deleted from an interface, portions of code that exist primarily to provide satisfaction for those deleted predicates can be deleted. If postconditions are added to an interface, they may cause other code fragments to become redundant and, thus, candidates for deletion.

Third, where preconditions or obligations (or data properties) have been added or altered, Infuse indicates where new code may be needed or where the encompassing interface may be affected as a result of the change. Similarly, changing or deleting postconditions causes similar behavior on the part of Infuse.

Finally, Infuse determines where and how exceptions and exception handling are affected. Changing the meaning of an exception result is identical in its implications to that of a successful result (that is, subject to the considerations in the three cases above). The results of removing an exception can be automatically determined since the environment knows exactly how the exception was handled and knows what the effects are of removing that exception handling. Adding an exception requires interaction with the user, since there is no way of determining what should be done automatically. One could, however, adopt one or more of the following strategies: automatically determine the implications of *ignoring* the exception, or *reporting* the exception, to determine whether there are any ill effects on the implementations that use the interface.

The implications of changes to interfaces, then, are determined 1) by the functionality depended on and what is known to be true at the point where the changed element is used, and 2) by how the exceptions are changed and now that affects the way they are handled.

4.2 Implementation Modifications

Changes in the implementation have effects on both the implementation and the encompassing interface. For example, changes in the implementation may affect what preconditions, obligations and postconditions may be propagated to the interface of the operation. Further, these changes may affect the predicates involved in re-interpretation at the boundaries of abstraction. With respect to the rest of the implementation, the environment recalculates the relationships of dependence and satisfaction to determine whether there are unsatisfied and unpropagated preconditions, obligations or properties.

Changes in the handling of exceptions will also have an effect on both the implementation and the interface. Internally handling a formerly propagated exception, or propagating an exception that was previously handled internally, changes the interface directly. Changing the form of handling an interface will have a direct effect of the state of the preconditions, obligations and postconditions within the implementation, and indirectly, on the interface.

In both cases, changes to the implementation have an effect on the internal organization of the implementation's semantics and may well

[4] Infuse has been the focus of joint work with Professor Gail Kaiser, Columbia University. In our reporting of this work [15, 25], we have concentrated only on the programming-in-the-many issues because they are separable from the particular interconnection model used in the analysis phases. It has seemed prudent to discuss that work in the context of syntactic interconnections, a model that is widely understood. However, the origin of Infuse is in Inscape as the system evolution component with the semantic interconnection approach as the underlying analysis mechanism.

have an effect on the interface as well.

4.3 *''Crowd-Control'' Issues*

The Inscape Environment is an example of a *city model* environment [28] — that is, it is an environment that is designed to support large projects. Infuse provides two basic policies governing group interactions: that of enforced cooperation and that of voluntary cooperation. The policy of enforced cooperation circumscribes the ways in which programmers interact while making a concerted set of changes to a collection of modules. Interactions among the developers are limited to particular points in the process of integrating the changes, and the changed modules progress through a rigid set of steps in the process from change to complete integration. The structure supporting this policy is the *hierarchical experimental database* (HEDB). Mechanisms that Infuse provides to implement this policy are those that create the hierarchy (by clustering the modules into closely related subsets), enable one to deposit a module into the parent database, and propagate the changes to the siblings in the database.

Several advantages accrue with this policy of enforced cooperation. The HEDB structure limits the effects of change to the confines of the particular experimental database, forms a context for negotiation of changes, tends to limit the extent of iterations in resolving conflicting changes, and imposes a systematic order on the integration of the components of the system that have changed.

The policy of voluntary cooperation allow an arbitrary number of developers (one or more) to form a workspace (the supporting structure) in which to simulate or propagate changes made to various modules in the workspace. For example, one developer might choose to determine the effects of his changes on several other particular modules by forming a private workspace and simulating his or her changes. Alternatively, several developers may choose to see how their changes interact. They collectively form a workspace and simulate or propagate their changes within that workspace.

The semantic analysis mechanism used in Instress, Inform and the change analysis described in the preceding section provides the engine for change simulation and change propagation. Thus the change analysis, change simulation and propagation, and the cooperation policies serve to coordinate the change process. In this way the environment manages the change process and can guarantee a high degree of completeness and consistency in the change process.

4.4 *Benefits: Environment-Assisted Evolution*

Inscape's semantic interconnection model, made possible by the formal interface specifications, is the basis for the environment's ability to understand the effects of change at a deeper level than possible with informal methods. When predicates are added, removed or changed in an interface, Inscape determines the effects on the dependencies that have been recorded in the construction process (that is, while building the program) and reports these effects to the user. Similarly, when an implementation is changed, Inscape (by using the semantic interconnections) reports to the user the effects that these changes have on the propagated interface.

Thus the investment in specifications has its primary return in the evolutionary part of the life-cycle that typically accounts for 60 to 80% of the life of a software system. This is particularly important because of the turnover of people that is generally experienced between the development and evolution phases, or for that matter, during any part of the life of any system. The new people are unfamiliar with the design and implementation intent of those that preceded them and are likely to make errors because of this lack of understanding. Inscape provides a way of improving this discovery process with its ability to determine the effects of changes.

It should be noted, however, that this feature of Inscape is extremely important in the development phase as well. System evolution begins very early in the development process in response to the discovery of requirements, design, interface specification, and implementation errors.

5. Use and Reuse

There are two important aspects of reusing components in building systems: finding the components to use and substituting one component for another in the system model. Inquire addresses the first issue; Invariant, the second.

5.1 *Browsing and Use/Reuse*

Inquire provides facilities for navigating through the system base following either unit, syntactic, or semantic interconnections. These facilities are available as browsing facilities to the user and as search facilities to Inscape's components.

As a browser, Inquire enables the user to browse through the system vocabulary (the predicates), the system objects (the modules), and navigate through the system following various types of syntactic and semantic threads. This navigation facility is of importance in the discovery process — that is, the process of developing an understanding of a system. The user may follow various objects dependencies, or follow various semantic dependencies, and thereby acquire an understanding of the system along various dimensions.

Because Inscape's tools are semantics-based, it is the behavior of operations and the properties of data that are important in determining use and reuse. For example, when certain preconditions need to be satisfied, the environment searches the system database and finds one or more operations or objects (whose postconditions or properties) will satisfy those predicates (thereby using the relevant interface information and reusing the various code and data structures).

In addition of finding various objects, it is useful to have information about the cost of those objects. For example, the Example 1 there are various semantic relationships among the operations: if you want to write a record, you first have to open the file, and because you opened the file, you are then obligated to close it. Merely finding that `WriteRecord` provides you with precisely the right postcondition is only part of the problem; there is a cost in chosing to use that operation. An informed choice among possible solutions requires that there is an analyzed cost associated with each possibility. For this reason, Inquire provides a weighting based on the relationships established in the specification.

5.2 *Version Management and Use/Reuse*

Because it is the behavior of operations that is important in the construction of software in the Inscape environment, we achieve a certain independence from the syntactic specifications in determining various properties about different versions. In strongly-typed version control mechanisms such as SVCE [10, 14] and System Modeller [16], we have a notion of equivalence and compatiblity that is tied to the typed signatures of operations, and that notion is both too strong and too weak — it allows versions to be considered equivalent that we know are not equivalent in an intuitive sense, and it declares versions to be incompatible that we would intuitively consider to be compatible. *Invariant* [27] provides us with precise definitions of equivalence and compatibility that coincide closely with our intuitive notions of equivalence and compatibility and thereby provides us with a very useful concept of plug-compatibility in the composition of system versions.

Intuitively, we think of two different implementations of a module as equivalent if their abstract external behavior is identical. For example, two different implementations of the data type `stack` are equivalent if the both provide a stack abstraction, independent of whether they are implemented as arrays, lists, or data in files. That is precisely the notion provided in Invariant: two versions are equivalent if their interface specifications are identical.

Of critical importance in the evolving of systems is the notion of upward compatible versions. Systems evolve more gracefully if extensions do not disrupt previously implemented functionality. However, we believe that there are two distinct concepts that are important here: dependency preservation and functionality preservation. Dependency preservation leads to the notion of *strict compatibility*:

version A is strictly compatible with version B if it requires no more (that is, A's preconditions are a subset of B's), provides no less (that is, B's postconditions are a subset of A's) and obligates equally (that is, A's and B's obligations are identical). Functionality preservation leads to what we call *strict-upward compatibility*: version A is a strict-upward compatible version of B if B's interface is included in A's.

Often there is only partial dependence on an interface. For example, one might start with a double-ended queue, but only put objects on the front of the queue and then take them off the front. One ought to be able, then, to substitute a stack for the double-ended queue with no undesirable effects. This observation is captured in Invariant's notion of *implementation compatibility*: version A is *exactly* implementation compatible with version B if the substitution of A for B in an implementation has no effect on the propagated interface; version A is *strictly* implementation compatible with version B if the substitution of A for B in an implementation I results in a propagated interface PI' that is strictly compatible with the original propagated interface PI.

These formal concepts — equivalence and compatibility — are crucial in the building of systems and the choosing among various versions from which to build those systems. They provide us with a notion of *plug-compatibility* and provide a level of static analysis that is not available in other version management systems. Moreover, they provide us with a formal vocabulary in which to formulate policies about version substitution at various stages in a system development process. For example, at some point in the development process, a manager might make the policy that only exactly implementation compatible versions are allowed as substitutions in the system model.

5.3 Benefits: Data Object and Operation Name Independence

Since the unit of use/reuse search and substitution is the predicate, reuse is independent of the names and numbers of parameters of procedures and functions, and the names of types and data structures (that is, the important aspect is the behavior of the operations or the properties of the data objects, not their particular names). While this makes the reuse mechanism more flexible and concentrates on the more important aspects of reuse, it does not provide a panacea. There is still a basic limitation: that of naming. In Inscape, this limitation is moved from procedure, function, type and structure names to that of predicate names — that is, a uniform vocabulary is needed at the predicate level.

6. Validation

We have discussed various kinds of analysis that Inscape provides in the specification, construction, and evolution of software systems. This analysis is based on a mechanism that makes no claim to provide full-fledged verification. On the contrary, it is one of the design trade-offs to make Inscape a practical but formal environment that we sacrifice this part of semantic checking and settle for a simpler, but useful, form of consistency checking. Because the checking is shallow, there will be inconsistencies that Inscape's analysis mechanism will not find. Our shallow form of checking will catch and prevent a large segment of the logical errors (in the main because a large segment of logical errors are shallow errors), but we will not detect all of them. For this reason, we consider ways of integrating testing into the Inscape Environment.

There are two aspects of testing that we consider: test-case descriptions derived from the interface specifications and the semantic interconnections constructed while the system is built and evolved; and integration test management that uses the city-model facilities of Infuse (this part of the research is being done with Professor Kaiser). The value of unit testing varies widely with the quality of the individual programmer. Problems that should have been caught in unit test are often not found until integration testing occurs. For this reason, we investigate the development of test descriptions that choreograph unit test drivers and stubs in order to better exercise individual modules before they are integrated into more complex, less easily testable pieces.

Integration test management is integrated into the Infuse subsystem and provides semi-automatic integration-test-harness generation from component test harnesses, and intelligent regression testing as a part of the integration of component parts. Integration of the component test harnesses may require interaction with the users to resolve redundant stub conflicts and create integrated test drivers. Parts of the component test suites may be rerun depending on the effects of the partial integration of the modules in the experimental data base. New test sets may be needed for the integrated unit. The test manager oversees the integration, creation process and regression testing.

7. Summary of The Inscape Environment

Inscape is an integrated environment that provides tools that are knowledgeable about the process of system construction and evolution and work in symbiosis with the system builders and evolvers. Integration occurs at two levels: an underlying common representation used by the various tools, and module interface specifications that supply the integration for the various kinds of analysis that Inscape provides.

The state of the environment varies depending on the particular part. The specification language has been defined and a syntax directed editor is being built. The form and scope of the logical language is under investigation. Inform is in prototype form (currently providing manipulation of sequence, selection, iteration and exception handling) and is being incrementally expanded to include more of the specification language and programming language. A prototype version of Infuse incorporating a syntactic model has been built by Professor Kaiser and her students at Columbia University. Their prototype will be ported to Inscape and retrofitted with Inscape's semantic model. The concepts for browsing and version management have been defined and designed at a high level. Work is currently focused on augmenting Inscape's analysis with testing. Eventually, we will have a complete prototype of Inscape.

Acknowledgements

The following contributed to the Inscape project in various ways. Bill Schell, AT&T Bell Laboratories, has helped to build various parts of the Inscape prototypes; Peggy Quinn, AT&T Bell Laboratories, made a significant contribution to the design of Instress during her internship; and Professor Kaiser, Columbia University, has, of course, contributed significantly to the crowd-control aspects of Infuse that are extended and incorporated into Inscape. Professor Kaiser's students have contributed in part to work on Infuse: Yoelle Maarek (also at Technion, Israel) provided a clustering algorithm; Bulent Yener directed the implementation of a syntactic version of Infuse. Professor Kaiser's and her students' work has been supported in part by the AT&T Foundation and AT&T Bell Laboratories.

Peggy Quinn, Alex Wolf, Mark Moriconi, and the conference reviewers all provided useful comments on the style and content of this paper.

References

[1] *Reference Manual for the Ada Programming Language. Draft Revised MIL-STD 1825*. United States Department of Defense, July 1982.

[2] W. H. Gardner. *Gerard Manley Hopkins: A Selection of his Poems and Prose*, Harmondsworth, Middlesex: Penguin Books, 1953.

[3] N. H. Gehani and W. D. Roome. "Concurrent C." To appear in *Software — Practice and Experience.*

[4] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types." in *Current Trends in Programming Methodology. Volume IV*. (ed Raymond T. Yeh), Prentice-Hall, 1978.

[5] J. A. Goguen and J. J. Tardo. "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications." *Proceedings of a Conference on Specifications of Reliable Software,* IEEE Computer Society, April 1979. pp 170-189.

[6] J. V. Guttag and J. J. Horning. "The Algebraic Specification of Abstract Data Types." *Acta Informatica* 10:1 (1978). pp 27-52.

[7] J. Guttag, J. Horning and J. Wing. "Some Notes on Putting Formal Specifications to Productive Use." *Science of Computer Programming* 2 (1982). pp 53-68.

[8] J. V. Guttag and J. J. Horning. *Preliminary Report on the Larch Shared Language*. Xerox PARC Technical Report, CSL-83-6, December 1983.

[9] John V. Guttag, James, J. Horning, and Jeannette M. Wing. "The Larch Family of Specification Languages." *IEEE Software*, 2:5 (September 1985). pp. 24-36.

[10] A. Nico Habermann and Dewayne E. Perry. "System Composition and Version Control for Ada." In *Software Engineering Environments*. H. Huenke, editor. North-Holland, 1981. pp 331-343.

[11] Nico Habermann, et al. *The Second Compendium of Gandalf Documentation*. Department of Computer Science, Carnegie-Mellon University. 24 May 1982.

[12] C. A. R. Hoare. "An Axiomatic Approach to Computer Programming." *CACM* 12:10 (October 1969). pp 576-580, 583.

[13] C. A. R. Hoare. "Programs are Predicates." In *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985.

[14] Gail E. Kaiser and A. Nico Habermann. "An Environment for System Version Control." *Digest of Papers Spring CompCon '83*, IEEE Computer Society Press, February 1983. pp 415-420.

[15] Gail E. Kaiser and Dewayne E. Perry. "Workspaces and Expermental Databases: Automated Support for Software Maintenance and Evolution." *Proceedings for the Conference on Software Maintenance — 1987,* Austin TX, September 1987. pp 108-114.

[16] Butler W. Lampson and Eric E. Schmidt. "Organizing Software in a Distributed Environment." *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems. SIGPLAN Notices*, 18:6 (June 1983).

[17] B. H. Liskov and S. N. Zilles. "An Introduction to Formal Specifications of Data Abstractions." in *Current Trends in Programming Methodolgy. Volume 1*. (ed Raymond T. Yeh), Prentice-Hall, 1977.

[18] David Luckham and Friedrich W. von Henke. "An Overview of Anna, A Specification Language for Ada." *IEEE Software,* 2:2 (March 1985). pp. 24-33.

[19] Mark S. Moriconi. "A Designer/Verifier's Assistant" *IEEE Transactions on Software Engineering,* SE-5:4 (July 1979). pp 387-401.

[20] Mark S. Moriconi. Personal communication.

[21] David S. Notkin. "The Gandalf Project." *The Journal of Systems and Software,* 5:2 (May 1985). pp. 91-105.

[22] D. E. Perry and W. M. Evangelist. "An Empirical Study of Software Interface Faults." *International Symposium on New Directions in Computing* (IEEE), Trondheim, Norway, August 12-14, 1985.

[23] Dewayne E. Perry. "The Inscape Environment and Programmer Productivity." *Proceedings of the IEEE Global Telecommunications Conference,* Houston TX, December 1986. pp 428-434 (12.6.1 - 12.6.7).

[24] Dewayne E. Perry and W. Michael Evangelist. "An Empirical Study of Software Interface Faults — An Update." *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences,* Kona, Hawaii, January 1987. Volume II, pp 113-126.

[25] Dewayne E. Perry and Gail E. Kaiser. Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems." *Proceedings of the 1987 ACM Computer Science Conference,* St. Louis MO, February 1987. pp 292-299.

[26] Dewayne E. Perry. "Software Interconnection Models." *Proceedings of the 9th International Conference on Software Engineering,* Monterey CA, March 1987. pp 61-69.

[27] Dewayne E. Perry. "Version Control in the Inscape Environment." *Proceedings of the 9th International Conference on Software Engineering,* Monterey CA, March 1987. pp 142-149.

[28] Dewayne E. Perry and Gail E. Kaiser. "Models of Software Development Environments." *Proceedings of the 10th International Conference on Software Engineering,* Raffles City, Singapore, April 1988. pp 60-68.

[29] A. L. Wolf, L. S. Clarke, and J. C. Wileden. "Interface Control and Incremental Development in the PIC Environment", *Proceedings of the 8th International Conference on Software Engineering,"* London UK, August 1985. pp 75-82

[30] W. A. Wulf, R. L. London, M. Shaw. "Abstraction and Verification in Alphard: Introduction to Language and Methodology." *IEEE Transactions on Software Engineering,* Vol. SE-2:4 (December 1976). pp 253-265.