Programmer Productivity in the Inscape Environment

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

published in

The Proceedings of GLOBECOM '86.
Houston Texas, 1-4 December 1986
pages 0428-0434 (12.6.1 - 12.6.7)

Abstract

The *Inscape Environment* provides tools for building and evolving large, programmed systems. The primary goals of the environment are to improve communication among developers, improve the effectiveness of the developers, assist in managing the complexity of building large systems, and assist in managing the evolution of large systems. To these ends, Inscape provides a formal module interface specification language (*Instress*) as the basis for program construction and evolution. Specifications, then, provide the basis for communication among programmers and the basis for a practical approach to software reuse. The environment enforces the consistent use these specifications and in so doing prevents a number of classes of errors from occuring. Further, the environment assists in process of constructing programs by recording the minute details of interconnections and by enforcing Inscape's rules of program construction, thereby aiding the developer in determining the completeness of an implementation. Because of the knowledgeable-environment approach, Inscape also assists in managing the process of evolution by tracing the implications of changes, coordinating those changes among multiple programmers, and guaranteeing the consistency and completeness of those changes.

1. Introduction

Programmer productivity issues must be considered in the context of the kinds of systems that are being built. Our approach in the *Inscape Environment* [Perry 85a, 86b] is to provide tools for use in the building of large programmed systems — systems that are built by a large number of developers, that are complex, and that evolve throughout their life. It is with these kinds of systems in mind that the Inscape Environment provides tools to

- improve communication among developers,
- improve the effectiveness of the developers,
- assist in managing the complexity of building large systems, and
- assist in managing the evolution of large systems.

In section 2, we present a summary of the Inscape Environment and discuss the general approach taken to aid the developer with interface specifications, program construction, and program evolution. We then discuss in section 3 four issues of programmer productivity in the light of the Inscape approach. And, finally, we summarize the discussion in section 4.

2. A Summary of the Inscape Environment

The fact that interface problems are significant in the development of programmed systems (cf. [Perry 85b, 86d]) led to the conception of a program construction and evolution environment based on the *constructive use* of module interface specifications. By *constructive use* we mean that the interface specifications are used by the environment in the construction and evolution of programmed systems.

Inscape currently addresses three problems in the process of building programmed systems:

- module interface specifications,
- program construction, and
- program modification.

2.1 Module Interface Specifications

Instress (the module interface specification language) extends Hoare's input/output predicates [Hoare 69] in order to describe the properties of data and the behavior of operations. We introduce the notion of *obligations* to extend the specification of an operation's results. Postconditions by themselves are not sufficient to capture all the side-effects of an operation; they describe what is known to be true after the execution of an operation, but do not specify what the programmer is obliged to eventually satisfy as a result of using the operation (e.g., closing files, deallocating buffers, making data consistent, etc.). *Example 1* illustrates what a specification might look like for the successful execution of reading a record from a file.

```
ReadRecord(<in> fileptr FP; <in> int R; <out> int L; <out> buffer B)
        Preconditions:
                ValidFilePtr(FP)
                FileOpen(FP)
                LegalRecordNr(R)
                RecordExists(R)
                RecordReadable(R)
                RecordConsistent(R)
        Postconditions:
                ValidFilePtr(FP)
                FileOpen(FP)
                LegalRecordNr(R)
                RecordExists(R)
                was RecordReadable(R)
                was RecordConsistent(R)
                Allocated(B)
                RecordIn(B)
                BufferSizeSufficient(B, L)
        Obligations:
                Deallocated(B)
```

Example 1

Further, we provide the notion of multiple results so that exceptions can be precisely and exactly specified in terms of what they mean (i.e., their postconditions) and what is minimally required to handle them (i.e., their obligations). Pragmatic information is also included with the exception specifications to indicate recommended recovery techniques or specific recovery operations. *Example 2* illustrates what an exception specification might look like for the *ReadRecord* operation where the data is readable but not consistent.

Exception:

RecordInconsistent(R)

Postconditions:

ValidFilePtr(FP)
FileOpen(FP)
LegalRecordNr(R)
RecordExists(R)
was RecordReadable(R)
not RecordConsistent(R)
Allocated(B)
RecordIn(B)
BufferSizeSufficient(B, L)

Obligations:

Deallocated(B)

Recovery:

ReconstructRecord(L, B)

Example 2

The environment provides a language-knowledgeable editor (built using the Gandalf environment generation tools — see [Notkin 85] and [Habermann 82]) for Instress that enables the developer interactively to construct module interface specifications. The editor eliminates the problems of syntax errors and specification standards by presenting the user with the templates for each language construct, the possible language commands for each metanode (i.e., each piece of the template that must be filled in by the user) in the template, and a standard presentation of the specifications.

2.2 Program Construction

The predicates used in the specifications of the properties and constraints of data and of the behavior of operations provide the basis for semantic interconnections (see [Perry 86a] for a discussion of the semantic interconnection model) used in constructing programs. Knowledge of the interface specifications, the implementation language, and the rules of program construction are incorporated into Inscape's program construction editor in order to maximize the effectiveness of the environment and to minimize the effort of the programmer. While constructing a program interactively, the environment 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. Further, the environment determines whether an operation's implementation is complete (i.e., there are no preconditions or obligations that are unsatisfied and unpropagated to the interface) and automatically creates the operation's interface on the basis of its implementation.

2.3 Program Modification and Evolution

The details recorded as a part of the system-building process form the basis for extending Inscape's program construction editor to handle program modification as well. Because of the semantic interconnections, Inscape is able to determine the implications and extent of changes, both to the interface specifications and to their implementations. The Infuse subsystem [Perry 86c] manages and coordinates multiple source changes by multiple programmers, propagates these changes interactively, and guarantees their completeness and consistency. Basically, the implications of a change are determined by the functionality depended upon and by what is known to be true at the point of the change. During the change process, the environment enforces Inscape's rules of program construction and the consistent use of the interface specifications in order to preserve the basic integrity of the implementation.

3. Issues of Programmer Productivity

There are four basic areas in which Inscape affects programmer productivity:

- writing module interface specifications,
- preventing classes of interface errors,
- assisting in the process of program construction, and
- assisting in the process of program modification and evolution.

We discuss each of these areas and indicate the ways in which Inscape improves the productivity of the programmer.

3.1 Module Interface Specifications — a Front-Loaded Process

With the emphasis on the formal specification, the Inscape approach puts a much heavier burden on the developer during the design and specification phase of the system building process. Much more thought is required to supply the amount of detail required in the module interface specifications than is currently required for the informal design and specification documentation that is generally provided as part of current system development. Note, however, that this difference in effort is due to the incompleteness and inadequacy that is accepted in the current, informal approaches.

Thus, the Inscape approach with its formal specifications requires more work, and, in a general sense, results in lower productivity in the design and specification phases of the building process. However, a distinction between short term and long term productivity is in order here. In the short term, there is indeed more work required in order to produce the specifications. To combat this short term decrease in productivity, there are some aids that the environment provides to ease the amount of effort required of the developer. The Instress editor makes it easy to copy sets of predicates so that the specification of exceptional results does not require tedious and repetitive typing. Some of the rules of interface specification are concerned with the relationships between preconditions and exceptions; the editor aids the specifier in determining what ought to be the minimal set of exceptions, given the set of preconditions. Further, the editor provides type and consistency checking so that the specifications are internally consistent. And finally, the editor checks for a minimal form of specification completeness.

Within the Inscape Environment, there are some long term benefits: the specifications provide the basis for construction and evolution. Without the specifications, the environment would not be able to assist in the construction process nor provide assistance in the modification process. We will discuss these aspects further in sections 3.3 and 3.4.

One important long term benefit of the language-knowledgeable specification editor is that the specifications have a uniform format and become the common means of communication between designers and programmers. While this does not reduce the amount of effort to produce the specifications, it does reduce the amount of effort needed to consume (i.e., read and use) the specifications. We agree with [Guttag 80] that *formal specifications ... proved to be a useful communication medium*.

The specification language coupled with the uniformity of the specifications provide an important basis for software reuse. Because the meaning and proper use of a module is captured in the specification and is understandable by the user, there is more of an increased incentive to reuse, rather than to rewrite, software.

Another important benefit, though largely unquantifiable, is that the resulting design is better because of the larger effort required to write formal specifications. [Guttag 80] states:

> *The most difficult part of this exercise was deciding on the abstractions we wished to have and on the functionalities of the operators associated with these abstractions. This is not surprising. Anyone who has spent time designing software knows that dividing it into appropriate modules is a difficult task. Fortunately, the process of trying to axiomatize the abstractions we had provisionally chosen proved to be a great help. Whenever we discovered that the axioms specifying the type were getting overly complex, we took this to mean that we had not achieved a proper separation of concerns, and consequently revised our choice of abstractions.*

Further, [Guttag 82] notes that:

> *The process of formal specification encourages prompt attention to inconsistencies and incompleteness and ambiguities in understanding. Each of our efforts in program specification has clarified our understanding . . . — whether the attempt came before or after the construction of the program. In many cases, such improved understanding has been the major result of the specification effort.*

Some aspects of improvement in productivity with respect to interface errors are discussed in the next section.

3.2 Error Prevention

In [Perry 85b], we provided an initial analysis of the interface faults in a system test and integration phase of a large, real-time operating system. An operational definition of interface faults (i.e., changes affecting more that one file and changes to header files — the system was written in **C**) yielded an error rate of approximately 36% interface errors. This set of errors was analysed and classified in detail. A preliminary look at the single file error set (representing about 64% of the total error set) yielded about 51% interface errors, for an overall percentage of 68.6% interface errors for the entire error set. In [Perry 86d], we completed our analysis of the single file interface faults and integrated the results of the two analyses.

There are a number of error classes which are basically design errors — errors that better design would have reduced or eliminated. These are the errors we referred to in the preceding section that would benefit from the formality of the specification phase. The amount by which they would be reduced is impossible to determine; however, we think that there would be a reasonable reduction in the number of errors. These error classes are as follows (and represent 24.4% of all the interface errors):

Inadequate functionality,
Disagreements on functionality,
Changes in functionality,
Added functionality,
Changes in data structures, and
Additions to error processing.

There are however, a number of error classes which are directly addressed by the Inscape environment and are largely prevented from occurring (in much the same way that syntax-directed editors prevent syntax errors from occurring — the editors do not allow them to occur). These errors classes are:

Data initialization/value errors,
Violation of data constraints,
Inadequate error processing,
Misuse of interfaces,
Inadequate interface support,
Hardware/Software interfaces,
Inadequate postprocessing,
Coordination of changes, and
System construction.

These error classes account for approximately 73% of all the interface faults (and approximately 50% of all the errors reported in this study). We discuss in the following paragraphs how Inscape prevents these errors from occurring.

The *data initialization/value* (5.7%) and *constraint-violation* (4.1%) errors are prevented by the environment because 1) they are explicitly defined for each type, constant and variable, and 2) the environment enforces the satisfaction of the data properties and constraints (as part of enforcing the consistent use of the interface specifications).

The *inadequate error processing* errors (17.7%) are prevented because 1) the exact meaning and minimal handling requirements of each exception is explicitly given in each operation interface specification, and 2) the environment enforces the satisfaction of those specifications as part of the program construction process.

The *misuse of interface* errors (6.7%) are prevented for the same reasons as the classes in the previous two paragraphs: the environment enforces the consistent use of the interfaces in the construction process.

Inadequate interface support errors (5.6%) are prevented by the environment because module interfaces are constructed automatically on the basis of the implementation of the module. A comparison can then be made between the specified and the constructed interfaces to see if the constructed interface supports (that is, in some reasonable sense it matches) the specified interface.

Given a hardware interface specification of the sort provided for software module interfaces, the environment can prevent the *Hardware/Software interfaces* errors (4.9%) in the same way that it prevents the misuse of interface errors.

Inadequate postprocessing errors (10.4%) are exactly those errors which occur because obligations are not specified in either formal or informal methods. By adding obligations to the result specifications, the environment can then enforce their eventual satisfaction.

Coordination of changes errors (10.3%) are eliminated because the environment provides a change management and coordination component in which the implications of the changes are first determined and then propagated in a complete and consistent manner.

Finally, *system construction* problems (7.7% — in this case endemic to building systems in **C**: essentially #include problems) are eliminated because the environment has a detailed understanding about the dependency structure of the system and generates the appropriate dependency information for the correct compilation and linking of the system pieces.

It is important to note that these problems that we have been discussing have occurred at the system and integration phase — that is, rather late in the development cycle. These same kinds of errors occur much earlier in the development cycle and while we have no explicit data about the frequency and the numbers of these errors, we feel (based on our experience developing large systems) that they constitute a significant problem in the early part of the implementation cycle as well. Hence, Inscape would prevent these errors from occurring when they would normally first appear, thus increasing the productivity of the programmer even further.

3.3 Environmental Assistance in Program Construction

We have alluded to a number of ways in which Inscape's assistance in program construction improves the productivity of the programmer. While enforcing the rules of program construction with respect to the Instress specifications, Inscape records the interdependencies that occur in the implementation — that is, the preconditions and obligations that are satisfied and the postconditions upon which they depend. When preconditions or obligations are not satisfiable within the implementation as it currently exists, an attempt is made to propagate them to the encompassing interface. However, it may not be possible to do this. Preconditions may conflict with previous preconditions or postconditions which form a ceiling that stops their outward propagation. Similarly, obligations may conflict with subsequent obligations which form a floor beyond which the obligation may not penetrate. When these precondition ceilings and obligations floors occur, the implementation is incomplete and further code must be inserted at appropriate points to satisfy these conditions. The environment determines when and where these conditions occur and provides assistance in determining how to resolve their satisfaction. For example, if two *ReadRecord* operations (see Example 1 above) occur one immediately after the other, the obligation to deallocate the buffer denoted by *Bufptr* incurred by the first occurrence will conflict with the obligation incurred by the second occurrence (that is, the pointer to the buffer will be lost and the buffer will not be deallocated). Hence, the second occurrence will form an obligation floor for the first obligation.

One way that the environment does this assistance is to search the system database to find operations which supply the desired postconditions and present the list to the user. Another way the environment helps is to provide a database browser for the programmer to determine what is available and what might be usable to build or complete an implementation.

Once the implementation is complete, the environment-constructed interface can be compared against the desired specification of the module to determine whether the implementations of the components are correct with respect to their specifications. (Note that because the form of consistency that we provide in the Inscape environment is relatively weak, there still may be errors in the implementation even though the implementation is complete and provides just what is required by the specifications — there may exist subtle logical inconsistencies that cannot be caught by the environment.)

Thus, Inscape improves the programmer productivity not only by preventing errors from occurring, but by assisting the programmer in building a complete, consistent, and correct implementation (within the limitations of the environments consistency checking).

3.4 Environmental Assistance in Program Modification

The Inscape Environment improves the productivity with respect to program modification in three ways: it determines the implications and the extent of changes; it coordinates the change process among the programmers making changes to the system; and, it ensures the completeness and the consistency of the changes.

By automatically determining the implications of any given change (by determining the affects of removing/adding various predicates from/to the interface specifications or the implementation), the environment improves the efficacy of the programmer and removes some of the iterations that result from incompletely understood changes. Further, programmers may simulate the effect of their changes on other modules to determine the resulting problems induced by those changes. Thus, the programmer’s understanding of a change’s effects is increased in a very practical way.

The Infuse component of Inscape provides an automated forum for coordinating sets of changes among multiple programmers. As changes are made and committed to, Inscape propagates them in a systematic way (through deposits of the modules into the parent experimental database) and determines the consistency of the changes among those modules in that particular experimental database. This process of deposit, propagate and consistency checking is repeated until all the changes have been processed and the changed modules deposited back into the base system. Thus the environment assumes a major role in coordinating the change process and ensuring the completeness and consistency of the changes made to the system. The programmer thereby is relieved of managing the detail of making changes and can concentrate on the functionality and the complexity of the changes.

4. Summary

We have discussed the different ways in which the Inscape Environment can improve the productivity of programmers in building large programmed systems.

- Instress formal specifications provide a uniform and common means of communicating the intent and use of the modules that comprise the system and, hence, improve the communication among programmers.

- Programmer effectiveness is improved in both measurable and non-measurable ways: Inscape prevents a substantial set of errors from occurring — potentially 73% of the interface errors (and thus 50% of all the errors) according to the study reported in [Perry 86d]; and the module interface specifications provide a useful basis for software reuse (enhanced by environmental assistance while building and changing the system).

- Inscape, by incorporating knowledge of the specifications, the programming language, and the rules of program construction, assists in managing the complexity of building systems by enforcing the consistent use of the specifications and by managing the details of interconnections and dependencies as the system is being built.

- Because of this knowledgeable-environment approach, Inscape assists in managing the process of evolution by tracing the implications and extent of changes, coordinating those changes among multiple programmers, and guaranteeing the consistency and completeness of those changes.

It is worthwhile noting that the improvements in productivity result from a set of coordinated tools and concepts, each attacking a portion of the problem of building systems. Further, the improvements are attained by paying the price of increased formality — by investing a large amount of effort in the formal module interface specifications at the beginning of the implementation cycle. Without this initial effort, the desired benefits would not be forthcoming.