The Inscape Environment: Knowledge-Based Synthesis of Large Systems through the Evolution of Program Interfaces [Extended Abstract]

Dewayne E. Perry

AT&T Bell Laboratories, 3D-454
600 Mountain Ave.
Murray Hill, New Jersey 07974
(201) 582-2529

The problems of Building Large Software Systems

Our research focuses on the problems of building a software development environment within which developers can construct and evolve large software systems. There are a number of fundamental problems that plague large systems; we address two of them in this paper. The first is that the majority of the life cycle is spent in evolution and maintenance, not development. [Boehm 81], page 533, states *Estimates of the magnitude of software maintenance costs range from slightly over 50% to 75% of overall life-cycle costs.* Second, as the size of systems increases, the problems of effective communication among developers increase even more rapidly. The complexity of systems, the interactions between pieces of the system, and the interactions between the developers grow in proportion to the size of the system.

It is our position that "classical automatic programming" is currently not mature enough to solve the problems inherent in producing large systems. Further, we assume that it will not become mature in the near future. We, therefore, take a half-way position between traditional software development and fully automatic programming in order to determine the kinds of benefits that may be derived from formal specification and verification on the one hand and automatic programming on the other while trying to avoid the hard problems that occur in both.

Inscape: an Interactive, Semi-Automatic Programming Environment

The Inscape Environment [Perry 86a] is centered around the constructive use of module interface specifications (expressed in the language *Instress*). Instress specifications formally describe application-specific knowledge in a declarative fashion — that is, Instress provides a means of reifying program interfaces. As interfaces are a major factor in the correctness proofs of large systems, we want to be able to reason about them explicitly. Further, Instress specifications are constructed by means of a specification-knowledgeable editor that provides consistency and completeness checking in addition to general semantic checking such as type checking.

The interactive, semi-automatic program construction process is automated by the *Inform* program construction editor which is knowledgeable about the specification language, the programming language and Inscape's logic of program construction. The knowledge provided in the specifications is instantiated for each specific use and additional knowledge about the software is captured as the program is being constructed.

Some of this acquired knowledge is obtained from the management of the details of constraint propagation [Stefik 81] — that is, Inform keeps track of constraint satisfactions so as to provide a basis for the understanding of behavioural dependencies between pieces of the system. Constraints are provided in several different ways: data constraints are given in the specifications as part of the description of the properties of the types, variables, and constants; constraints on operations are provided by preconditions and obligations (preconditions are predicates which must be satisfied before the operation's invocation; obligations are predicates that must be satisfied subsequent to the operation's execution — see [Perry 86a]); constraints on exceptional results are expressed for each operation as well (enabling the environment to understand the nature of these conditions and how to handle them — a much more conservative approach than automatic generation of error detection code as in RIP [Kelly 86]). These constraints must either be satisfied within the implementation or be propagated to the interface of the piece of software being constructed.

However, because of the logic of program construction, the environment may not be able to propagate the unsatisfied constraints to the interface. In this case, the implementation is considered to be incomplete. To handle this problem of incompleteness, Inform maintains an agenda of unsatisfied preconditions and obligations, selects possible techniques for satisfying them, and presents the user with a prioritized list of these possibilities. Prioritization is based on user-supplied pragmatic information and the number of constraints that can be satisfied by each possibility.

An important aspect of the evolution process is automated by the *Infuse* subsystem [Perry 86b]. The primary goal of Infuse is to maintain the consistency of all the interfaces and their uses while a system evolves. The construction process has guaranteed the satisfaction or propagation of the constraints expressed in the specifications; the evolution process must eventually preserve these goals of satisfaction or propagation. One of the tools used by Infuse in determining the effects of changes to existing software is a truth maintenance system. Infuse combines a backtracking form of truth maintenance with a weak form of assumption-based truth maintenance [de Kleer 86] to perform shallow searches of spaces of possible changes.

A notion common to the various tools underlying the Inscape Environment is that of *consistency*. We have identified several strengths of consistency, the different logics associated with each form of consistency, and the relative costs of implementing those logics. At this point, we provide a relatively weak form that yields a practical environment at the expense of allowing the possibility of errors in the implementations. We are currently investigating the implementation of a slightly stronger form.

Summary

Inscape provides mechanisms for ameliorating two fundamental problems in the life-cycle of large software systems: communication and evolution. First, the knowledge base acquired in the form of

application-specific knowledge (the interface specifications) and the construction of the program (acquired in the construction and evolution process) provide a useful means of communication among the developers in a large project. Second, the interactive, semi-automatic building and evolving of software provide a gain in productivity over traditional methods, with the environment assuming a much more intelligent and active part of the construction and evolution process.

References

[Boehm 81] Barry W. Boehm. **Software Engineering Economics**. Prentice-Hall, 1981.

[de Kleer 86] Johan de Kleer. *An Assumption-Based TMS*. **Artificial Intelligence**, 28:2, March 1896, pp 127-162.

[Kelly 86] Van E. Kelly and Deborah L. McGuinness. *Automatic Re-Programming for Robustness*. To appear in GLOBECOM 86.

[Perry 86a] Dewayne E. Perry. *The Inscape Program Construction and Evolution Environment*. Computer Technology Research Laboratory Technical Report, AT&T Bell Laboratories, April 1986.

[Perry 86b] Dewayne E. Perry and Gail E. Kaiser. *Automatically Managing and Coordinating Sources Changes in Large Systems*, Computer Technology Research Laboratory Technical Report, AT&T Bell Laboratories, June 1986.

[Stefik 81] Mark Stefik. *Planning with Constraints (MOLGEN: Part 1)*, **Artificial Intelligence**, 16:2, 1981, pp 111-139.