POSITION PAPER
for the International Workshop on
The Software Process and Software Environments

TOOLS FOR EVOLVING SOFTWARE

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

Current State of Affairs in Software Devlopment

There are three properties that are of paramount importance in the design and implementation of software systems:

- most of these systems are complex,
- these systems evolve during their lifetime, and
- the development of these systems is generally a group rather than an individual effort.

Despite our awareness of these properties, we have done very little to provide automated support to aid the developer and ameliorate the effects of these factors.

Further, even though it is intellectually accepted that we have methodological means to combat these factors and provide benefits during the evolutionary process, development managers still choose short term goals over these long term benefits. Given a choice between getting a system out as early as possible and doing a proper design that will evolve gracefully, the first is invariably chosen at the expense of evolutionary problems.

The reasons for this choice of short term gains in exchange for long term pain tend to be complex. However, one central reason given is that early delivery is an extremely important fact of life, both in terms of the marketplace and individual advancement and that the long term benefits are rather more promisory than substantial. It is time to provide tools to make these long term benefits substantial and desirable, even though some of the short term goals such as early delivery may suffer.

Tools for Evolution of Software

The tools proposed in the *High Quality Program Construction* (HQPC) project [1] attack this problem in several ways. Basically, the tools revolve around the constructive use of interface specifications: a formatted (or, if you will, structured) module interface specification provides the basic means of system construction among groups of developers; and the specification (by means of its component parts) becomes the basis for the construction of the system and for its evolution.

The module interface specification language (ISL) is designed to capture as much of the meaning of the interface as is possible so that the interface may later be used in the construction of the encompassing layers of the system. In order to do this, the interface specification delineates the types, constants, data and operations of a module, and by means of a structured format along with first order predicate calculus describes the interrelationships among the operations in the module, the interrelationships between the data and the operations, the side effects of operations on data and parameters, and the exceptions that may occur, the conditions under which they occur and their effects.

An ISL knowledgeable editor has been constructed (based on [2]) to guide the user in specifying a module interface. While the effort to define precisely an interface in this manner is considerably greater that the informal (and error prone) methods currently used, there are a number of advantages. First, the effort necessary is ameliorated by the constructive nature of building the interfaces. The editor environment provides beneficial automation and interaction to reduce the amount of work for the specifier. Second, the environment enforces an uniform standard and structure on the interfaces. Third, the environment may be able to guarantee certain aspects of semantic consistency within the interface. Fourth, some aspects of the implementation can be automatically generated from the specification, thereby reducing the implementation effort.

Given the existence of these specifications, an program construction environment can be built that uses these specifications in the construction of other pieces of the system and interactively constructs both the implementation and the new components interface specification. By moving the knowledge of program construction, the programming language, and the interface specifications into the environment, a number of advantages are gained. First, the environment eliminates a large class of interface errors by prohibiting them (in the same way a language knowledgable editor prevents syntax errors). The environment guarantees the consistent use of data and operations according to the interface specifications. Second, the environment minimizes the amount of programmer effort in constructing new components of software by proceding either automatically or interactively. Third, the environment records the relationships of how data and operations are used and the fine interconnections that are constructed. For example, the way in which pre- and post-processing conditions are satisfied, the way in which pre-conditions are satisfied by post-condition of other operations, the way in which some exception conditions may be precluded, are all recordable by the environment. Fourth, system version control and generation (see [3] and [4]) fall out (with a small amount of extra work) as a side benefit.

Once the system has been constructed and the interconnections and dependencies recorded as part of the construction process, the evolution of the system is more easily managed. The effects of change are determinable with finer distinctions that just the "use" relationship. We have captured more of the meaning of the use of the components and can trace the implications of changes to an interface or the changes to the implementation of a module.

In summary, a program construction environment of this nature can provide the following benefits. First, it can increase programmer productivity through interactive program construction and increased software reuse (a side effect of the interface specifications). Second, it can increase the robustness of the system by eliminating a large class of interface errors and enforcing the consistent use of data and operations and the consistent handling of exceptional conditions. Third, the problem of maintenance and evolution can be managed by making explicit the fine interconnections and interactions between pieces of the system, thereby reconding some of the meaning associated with use, and automatically tracing the implications of any particular change.

REFERENCES

[1] Dewayne E. Perry. **The High Quality Program Construction (HQPC) Environment** *Technical Memorandum Draft, March 1983.*

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

[3] A. Nico Habermann and Dewayne E. Perry. **Well-Formed System Compositions.** Department of Computer Science, Carnegie-Mellon University, March 1980. Technical Report CMU-CS-80-117.

[4] A. Nico Habermann and Dewayne E. Perry. *System Composition and Version Control For Ada.* in **Software Engineering Environments**, Horst Huenke, editor, North-Holland, 1981. pp 331-343.