Technical Report
CMU/SEI-87-TR-7
ESD-TR-87-108

Tool Interface Technology

Joe Newcomer

March 1987

Technical Report

CMU/SEI-87-TR-7

ESD-TR-87-108

March 1987

Tool Interface Technology

Joe Newcomer

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Tool Interface Technology

Foreword

The Technology Identification and Assessment Project combined a number of related investigations to identify:

- existing technology in a specific problem area to review research and development results, and commercially available products;
- new technologies through regular reviews of research and development results, periodic surveys of specific areas, and identification of particularly good examples of the application of specific technologies;
- requirements for new technology through continuing studies of software development needs within the DoD, and case studies of both successful and unsuccessful projects.

Technology assessment involves understanding the software development process, determining the potential of new technology for solving significant problems, evaluating new software tools and methods, matching existing technologies to needs, and determining the potential payoff of new technologies. Assessment activities of the project focused on core technology areas for software engineering environments.
This report is one of a series of survey reports. It is not intended to provide an exhaustive discussion of topics pertinent to the area of user interface technology. Rather, it is intended as an informative review of the technology surveyed. These surveys were conducted in late 1985 and early 1986.

Members of the project recognized that more general technology surveys have been conducted by other investigators. The project did not attempt to duplicate those surveys, but focused on points not addressed in those surveys. The goal in conducting the SEI surveys was not to describe the technology in general, but to emphasize issues that have either a strong impact on or are unique to software engineering environments. The objective in presenting these reports is to provide an overview of the technologies that are core to developing software engineering environments.

1. Introduction

One of the key areas in which project members were interested was tool interface technology. This report discusses the need for tool interfaces and some of the current trade-offs in tool interface technology, emphasizing the trade-offs between homogeneous and heterogeneous tools. By highlighting some of the major issues, this report reflects the state of the technology today.

2. Tool Interface Technology

The fundamental goal of tool interface technology is to make it possible for many independent hardware/software components to share information. While there are many low-level technologies that allow the sharing of information (e.g., object file formats common among many languages), the growing complexity of tooling and information, and the realization that coding is but a small part of the problem indicates that more sophisticated tools are needed.

The notion of software development environment technology implies that information is shared at all levels — not only at the "manufacturing" level, but also at administrative and support levels — and shared at all times during the complete product life cycle. Initial requirements specification, problem analysis, system design, coding, testing, product delivery and distribution, maintenance, and even obsolescence are all activities that need to share complex information in increasingly critical ways.

Part of the technological problem is that many of the tools currently employed at these levels are not designed to work together toward a common goal. Word processing/document processing systems used in the requirements documents do not create structures that can be used to trace design decision. Project planning and management tools do not have interfaces to the actual task tools, e.g., directing the development of the program by direct input of the project plan. Implementation tools do not have provisions to feed information back to the project management tools, e.g., project tracking by direct analysis of the programming environment database.

Even within tasks there is little provision for sharing; for example, some project management tasks can best be handled by a spreadsheet capability, while the output from the spreadsheet might then be used to manipulate the project dependency graph. Currently such independent programs have no connections; one must have "integrated" tools designed to handle the complete task.

A limitation of monolithic integrated systems is the difficulty of incorporating new ideas into the system. New ideas, new tools, and new needs can suddenly make the integrated system a problem rather than a solution.

An alternative approach to the highly integrated monolithic tool sets is the nearly uncontrolled anarchy of some other environments. It is easy to create or add new tools or replace old tools, but there is little control or standardization at the interfaces. Interfaces that are not fully specified can lead to surprising behavior when the valid (but undocumented) output of one tool doesn't fit the specification for the input of a subsequent tool. Also, growth and extension of such anarchic toolsets presents significant managerial problems. The major thrust in tool interfacing over the next few years should be to develop a technology that allows the following:

- controlled but uninhibited growth,
- interfacing between new technologies and existing technologies, and
- interfacing of relevant but independently developed programs — within tasks, across tasks, and at the supra-task levels of project management and administration.

It is important to remember that stronger type mechanisms in programming languages or better data description mechanisms in conventional databases will not be adequate. Strong typing is actually an extremely weak form of semantic consistency specification. To interchange information among diverse applications, a stronger approach to semantic consistency is necessary. The database approach is also syntactic, since it provides no intrinsic mechanisms that preserve semantic consistency. Semantic consistency must be maintained by specifications and mechanisms *outside* the applications programs that manipulate subsets of the information; otherwise, the complexity is limited, and growth quickly becomes impossible because every application program must be updated to maintain consistency with each new relation or its equivalent. This suggests that future interface specification development should emphasize more precise semantic specification.

Interfacing diverse tools will become a key problem in constructing sophisticated software development environment technology. Sources of important ideas and programs or the hardware they will use cannot be anticipated; the best or the most appropriate technology should be integrated as it emerges.

Integration may take the form of specifying and adopting standards. Many standards are in place, but many more need to be specified. New tooling can be developed with these interface standards in mind. However, older tooling and tooling that needs to use information in a form different than that for which it was developed (whether non-standardized information, information adhering to an older standard, or information in simply a different but standardized form) must be accommodated. This can be done by providing mapping functions that transform information on input and/or output between the desired forms. In the presence of pervasive information, this again demonstrates the value of handling semantic consistency with data specification in an active database rather than with the mapping programs.

For example, a simple "UNIX[1] pipe" approach to interfacing data in form "A" to data in form "B" suitable for processing via a program "B" might be:

[1]UNIX is a registered trademark of Bell Laboratories.

But the problem is much more serious when the scenario is:

The A format view represents a way to get the information from the database, but it is still in a database-relative format (e.g., a set of relations). The A-to-B transformer must convert this information to a suitable form for B to manipulate, and the output from B must be fed back into the database, possibly updating the information that produced the original A format view. Consistency and correctness of the database must be maintained, even though the A view is a subset of the total information available and other relations might be affected by the updating process as the output from B is fed back into the database.

The technology for dealing with these mappings is new; there is now a product available [5] that handles some of the remapping problems, but the deeper problems remain.

3. Issues

To discuss the problems in tool interfacing, an overview is presented that is intended to capture the essence of some of the problems without going into detail.

The basic goal of tool interface technology is to make it possible to interconnect the components of a system by providing a mechanism for passing information among them. For simple tools and information structures this can be quite straightforward, e.g., a trigonometric routine taking an input value and producing an output value. However, as the nature and complexity of the information changes, simple mechanisms are no longer adequate.

Simple type mechanisms were introduced to attempt to maintain consistency at the interface level; thus, one could not pass an integer to a procedure expecting a double-precision real number. Even this simple type mechanism is not available in many languages that support separate compilation because the type consistency cannot be enforced across compilation boundaries. More modern languages, compilers, and environments provide better support (e.g., Ada, Modula-2 and C/lint), but mechanisms alone do not suffice if they are not used properly; for example, one must actually define and create types. Passing a real number whose units are degrees to a procedure that accepts a real number whose units are radians is often valid, but produces surprising results.

Tool interface technology is well developed in the EDP/Database world. Various mechanisms allow applications programs to run unchanged in spite of changes in the structure and organization of the underlying database, as long as the abstractions required are preserved. Many of these mechanisms, however, cannot be applied generally to the complex information processing that is beginning to characterize programming and project environments.

Conventional EDP databases tend to involve a small number of scalar types, usually of fixed size, and a small number of relations easily expressed in terms of those scalar types. Although one can produce very complex structures in this way, the structures are usually examined along certain restricted dimensions at any one time. A simple characterization is that there are many instances of a few types, and any step in the processing involves a very small number of relationships among these types. The patterns of computation are almost always predictable, occurring at fixed, known intervals (e.g., daily, weekly, quarterly), and careful analysis of time/space costs based on the known transaction style can allow the system architect to predict reasonably accurately the throughput of the system.

The more complex information of lifecycle-pervasive environments — those that try to support all aspects from the requirements assessment through post-deployment support — involves a small number of instances, each of many types, whose fine structures are complex and not of predetermined length; and there are many relationships among the types. Furthermore, the usage patterns are not *a priori* determinable, since they depend upon particular project management strategies, needs for information, and events that are neither regular nor frequent.

A new property that these systems introduce to software development is the presence of *persistent information*, a property well-known in the EDP community. Over the lifetime of the project, the database must not only support a heterogeneous collection of information (including graphs, program source, documentation, test data, customer reports, etc.), but also must be available for new tooling as it is introduced. The classical collection of text files organized by file name or directory name is not capable of coping with this class of problems, largely because of the unstructured nature of such information.

As more structure is imposed on the information, the needs of unanticipated new technology must be addressed. This technology will also deal with the information and its relations in ways far more complex than the currently available, simple relational database models can support; and they must do so efficiently. Imposing a new structural layer onto an existing database system has the potential of incurring unacceptably high performance costs. Nonetheless, this may be the most effective way (given current technology) to explore the deeper issues of such a structure.

4. Issues in Interfacing

The following subsections represent a list of issues in interfacing. For each issue, motivations for their choice and cost/flexibility trade-offs are given. Some promising new approaches will then be discussed.

4.1. Memory Resident Interfaces

These interfaces are characterized by data that typically have a brief existence ranging from a few microseconds (e.g. a stack frame) to hours (a long program run). Only a few anomalous cases occur, e.g., operating system data structures that may persist for months if the hardware and software are reliable. However, the interfaces usually are not transmitted external to the program's address space. They are usually recreated when the program starts execution and do not persist beyond the (normal or abnormal) termination of the program. Memory interfaces are also seen as highly reliable interfaces at the bit level; there is rarely any error in the transmittal of the physical data. The interpretation of that data, of course, is a different problem; strongly-typed languages are an approach to syntactic correctness of the information, but not sufficiently powerful to guarantee its semantic correctness.

These interfaces are not particularly flexible; once an instance of such an interface strategy is determined (usually by a compiler), a strong commitment is made to its representation. It usually cannot be changed without regenerating the system, e.g., recompiling and relinking in the simplest case. The cost of a change can be unacceptably high when a complex set of interactions encompassing several modules, plus acceptance testing, is involved.

4.2. Message Passing Interfaces

These interfaces are characterized by a brief existence (the transmittal time of the message), but are usually transmitted external to the program's address space. Messages are created, transmitted, received, and destroyed. Significant considerations here include the fact that the information may be transmitted in a heterogeneous environment and is frequently very simple in structure. However, it is usually assumed that the transport mechanism is unreliable; and at some level of abstraction, it is no longer safe to assume that a message sent is a message received.

The need for portability often places a limitation on the complexity of the information passed through a message. Pointers to other data structures are classically hard to encode, so what is usually passed consists of one or more records or sequences of scalar values. However, scalars also have their limitations (see 4.4).

Remote Procedure Call (RPC) mechanisms are an interesting extension, and one that is becoming more important in modern distributed computing. In RPC, the parameter passing mechanism may have to pass complex information structures; if it passes them by reference instead of by value, additional complications occur; and if the structures contain pointers to other structures, even more elaborate mechanisms must be included in the RPC mechanism. RPC also has all of the complications engendered by message loss, receiver failure, etc., with additional complications of recovery. However, the power and flexibility of RPC are making it a potentially important replacement for some of the more limited message passing systems in distributed environments. A significant contribution of RPC to programming methodology is that it frees the user from the task of determining the site of the activation. In a fully general system this may involve scheduling resources, such as finding an idle processor, in a manner that is completely transparent to the user.

4.3. Persistent Interfaces

Persistent interfaces are those where the information being passed along the interface has an existence quite independent of its creator. File systems and databases are classical instances. The lifetime of such data is not only independent of the creating process, but in fact often exceeds the useful lifetime of the code that constituted the creating process. Revised programs must be able to access this data without requiring reorganization of the information. Reorganization may incur either prohibitive cost or simply be impossible.[2] Thus, representation independence, data dictionaries, and similar mechanisms have arisen in the EDP community in response to a very real set of problems. These problems have been largely ignored in the computer science community, where persistent data may have a lifetime of only weeks or months.

In programming environments, the information has the quality of the persistent interface. In a 5-year project, it should be possible not only to access the requirements documents from which the project was created, but also to provide annotations, communication, feedback, traces, etc., of the current system relative to those initial documents. Change log histories from the beginning of the project may be needed and should be accessible. However, the environment itself may change over time, because there are new releases of tools or completely new tools introduced in the environment, or new hardware that requires porting of the environment. None of these events should cause critical information to be lost.

A number of factors seem to preclude the use of conventional DBMS technology from maintaining this information. They include: structures such as graphical data structures; program sources of indeterminate length; annotated post-semantic syntax tree representations (such as structure editors use); and the need to establish relations at levels finer than the gross "file" level (for example, forming a relation between a field bug report or feature upgrade request and the line or two of code which performs it, or the paragraph in the revised requirements document that would reflect a change in the specification.)

4.4. Structural Interfaces

As information becomes more complex, it is no longer possible to encode it effectively as simple scalars. Some mechanisms which now exist are text encodings of trees, dags, or general cyclic graphs. While allowing a general encoding, these mechanisms can be costly. Notably, the cost of encoding as text, writing text, reading and parsing text, and encoding text as binary data can be quite high. When text is used as a communication mechanism between tightly coupled components of a system, significant performance costs can be incurred. However, such mechanisms

[2] The last machine that can read the magnetic tapes from the 1960 census is due for decommissioning soon. There is apparently no way to transfer those tapes to a more modern medium, so they will be completely inaccessible.

allow communication in heterogeneous environments; for example, if the text is limited to the 95 "printable" characters of the ASCII set, plus "newline" or other equivalent punctuation, such structures can be interchanged between various 8, 16, and 32-bit architectures.

Such changes do not occur without management and development costs. The readers and writers must agree on the format of the information; unless a fully general mechanism (such as extended S-expressions) can be used, each writer and reader must be individually handcrafted. Changes in the structure will then require changes in all associated readers and writers. This can be a formidable management task. Even with a fully general mechanism, the form and content of the resulting data structure must be agreed upon. Ideally, existing code should be reasonably impervious to change in the presence of upward-compatible changes.

Regardless of these problems, the importance of structural interfaces is increasing as more complex information must be passed among system components. An example of a highly-structured interface with a textual representation is the PostScript[3] system [1], an interface designed for the transmittal of complex multifont documents.

4.5. Impact of Interface Considerations on Programming-in-the-Small

There is, as usual, a trade-off between flexibility and other parameters. For example, a data structure access of the form

```
A.B.C         (Ada)
A^.B^.C       (Pascal)
A -> B -> C   (C)
```

encodes very strongly the notion that the B field is a component in the record referred to by A and is found at a distinct offset within that record. Pascal and C are even more problematic, since the programmer must also encode the fact that A is a pointer to a record instance, and B is a pointer found within that record (at a specific offset) which refers to a record that contains a C field. Such programs contain no representation independence. Mechanisms that create record definitions from a data-dictionary-like specification and require recompilation of the programs with the new definitions help only a small part of the problem, since there is still a commitment to a representation at the source level.

Procedural interfaces introduce a level of abstraction; for example, the interface

```
C(B(A))
```

simply constrains the B operation to provide a piece of information when applied to the name A, and the C operation, when applied to this value, delivers the desired result. There are those who, with significant justification, argue that this approach provides entirely too much representation information, and that the correct access is

```
C(A)
```

where the implementation decides (via its data dictionary) exactly how to find the C information when given the object A.

Procedural interfaces are extremely clumsy to write, and they only solve the right-hand-side

[3]PostScript is trademark of Adobe Systems Incorporated.

(RHS) value; languages like Ada and Pascal do not permit procedures to return left-hand-side (LHS) values. This means that LHS values require some other mechanism, e.g., using ’store’ procedures for assignment. These do not usually work when ’var’ (Pascal) or ’out’ or ’inout’ (Ada) parameters are required, and the result is some fairly distorted code.

Procedural interfaces are typically very expensive at runtime. They are usually implemented by compilers as the most general procedure-call mechanism. Very few compilers allow the user to specify that the procedure (defined in a separate module) should be compiled inline or perform such optimizations automatically.

Many of these problems do not arise in the DBMS/EDP community because the notion of pointers is encoded as keys in relationships. The more general mapping shown above is a common pattern: given a tuple A and another relation R, retrieve the corresponding C data. However, as indicated earlier, the costs associated with these are quite different, and attempting to use such a mechanism to manipulate the abstract syntax tree in real time in a screen-based structure editor would not give adequate performance. This is because the nature of the relations and the usage patterns of classical DBMS systems involve coarser-grained interaction on large quantities of structurally identical information.

A mechanism that provides data independence and works naturally within the language, and does not incur severe cost is necessary at the programming level. Although there are some candidates for this, they do not respond to all of the problems.

5. Flexibility Requirements

There are two extreme positions of tool integration. In one model, the tool does everything; new features are added by integrating new components into the tool. This rapidly becomes self-limiting. By analogy, few people use a Swiss army knife, which includes knife blades and scissors. If one adds a torque wrench, an oil filter removal tool, and a small astronomical telescope, even fewer people would use it. Further, adding a new tool to the knife becomes increasingly difficult.

The alternate extreme is analogous to selling an empty toolbox and providing a tool catalog. While it allows customization, there is substantial overhead involved in identifying the right tools, and significant problems occur if the toolbox does not have a space for them (adding a chain saw or two-man crosscut to the average toolbox does pose certain technical difficulties). Connecting the tools together, where that analogy applies, is substantially more complicated than buying a 3/8"-to-1/2" socket wrench adapter.

Somehow, new tools that interact in ways not yet predicted must be accommodated. In the case of future computing systems, a variety of tools from all phases of the project lifecycle must be integrated into something that actually supports a project across its lifetime: project planning tools, documentation tools, accounting and cost tools, program construction tools, testing tools, maintenance support tools, and many others.

A single vendor, tool, or machine cannot be expected to support all of these requirements or even a subset of them effectively. As technology becomes software driven, it will be less important which hardware is chosen since software costs are now already dominating hardware costs (e.g., today, it is possible to install a $40,000 CAD software package on a $12,000 computation engine.) Thus, preparations must be made to integrate programs into a computer and also to integrate the computers that run those programs into an assemblage of other, heterogeneous, computers that support various aspects of the project lifecycle.

6. Potential New Technologies

A system recently coming into use within the Ada community and elsewhere is the Interface Description Language (IDL) data structure notation, used to specify the Diana representation for Ada compiler intermediate representation [7]. IDL provides both a language-independent structure specification (allowing interface to multilanguage environments) and an interchange format specification; however, for the latter case the specification does not preclude highly optimized representations for tightly coupled systems. With certain careful engineering considerations taken into account, an IDL support system incurs no more time or space overhead than conventional language-specific record systems; and it can support upward-compatible changes with automatically generated or fully generic readers and writers for a variety of interchange representations.

More speculative, but also more promising, are systems based on the *object-oriented model*. Such systems include Smalltalk [4, 8, 6, 3], Actors, and Flavors. In these systems, one does not so much act upon data as request data to act. This shift in emphasis allows richer structures to be built and enhancements to be made over time while maintaining a consistent interface to the user. In addition, the notion of *active databases*, in which the database has responsibility for maintaining its consistency and integrity relations rather than the (distributed) applications code, allows much more complex structures to be built. Systems such as CAIS demonstrate that this is a highly promising direction for future development. CAIS (the node model) currently rests somewhere between the simpler structure representations and the fully general active database model.

For control, the notion of *remote activation*, of which RPC is but one instance, becomes important. Active databases, which are themselves distributed, must be able to initiate activities on other machines. When they are combined with object models and active databases, more flexible and general paradigms can be developed. The concentration on costs of these mechanisms thus becomes a structural and organizational issue (where, when, and how to act upon information) rather than a construction issue (the cost of building these mechanisms). Certainly the various DBMS systems have accomplished this for their problem domain; a philosophically similar approach of developing basic mechanism packages with general applicability needs to be followed.

7. Rethinking the Problem

Some of the interesting implications of long-term information storage are the requirements of *traceability*, *accountability*, and *re-creatability*. These requirements move away from the "file" model in which a small number of files are "updated" (effectively changed-in-place). They also move away from the database model in which records, relations, or values are "updated" (usually changed-in-place) toward a model in which nothing is ever "discarded"; all versions of all information are preserved. Logistically, this would entail consuming nearly infinite amounts of disk space; pragmatically, one must regularly remove information to a long-term archive store. However, it is currently the case that such "checkpoints" are determined by administrative action. It must become the case that checkpoints and consistency are automatically maintained by the system, and administrative choices for baseline points must be validated for consistency. As systems grow in complexity, it becomes increasingly difficult for any one person or group to maintain the consistency requirements across thousands of modules and millions of lines of code. New approaches to the problems of information storage, independent of all other considerations, must be considered. Thus, a multiversion, pervasive store may be an active database, an object database, or a passive information database; it may be accessed via hand-coded interfaces, automatically generated interfaces, message interfaces, or whatever — but it must be a new way of thinking about the problem. Approaches such as HyperText/HyperData [2, 9] address many of these questions, but by no means all of them. Considerable work remains.

8. A Taxonomy of Issues

The previous sections have discussed some of the topics in tool interface technology that are presently important to software development environment technology. In summary, these topics involve the following issues:

- transient vs. persistent data,
- data vs. control issues (local procedural communication vs. remote activation),
- strong typing (syntactic) vs. interpretation (semantic), and
- information structures passed (single word, fixed length text, variable length text, structures, pointers, objects).

Each of these represents a different kind of interface problem. Many of the problems have characteristics that make them appear to be information *management* problems rather than simple information *communication*.

9. Conclusions

Tool interfacing is one of the core technologies that must be understood and treated properly for software development environment technology to continue to evolve. Unfortunately, tool interfacing is not well understood, and the trade-offs between alternative interfacing methods are not easily evaluated. A homogeneous system is attractive for obvious reasons, but system homogeneity inhibits the ability of software development environment technology to evolve quickly as new technologies emerge in heterogeneous forms. Heterogeneity, on the other hand, poses significant communications problems. The ideal solution should combine the strengths of both approaches while bypassing the weaknesses. To date, there is no readily available technology that captures these characteristics. Although preliminary research results are encouraging, much more work is needed before the important issues are resolved.

References

[1] Adobe Systems.
PostScript Language Reference manual (ISBN 0-201-10174-2)
Addison-Wesley Publishing Company, Reading, MA, 1985.

[2] Steven Feiner, Sandor Nagy, and Andries van Dam.
An Integrated System for Creating and Presenting Complex Computer-Based Documents.
ACM Computer Graphics :181-189, August, 1985.

[3] Adele Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, 1984.

[4] Adele Goldberg and David Robson.
A Metaphor for User Interface Design.
In *Proceedings of the 12th Hawaii International Conference on System Sciences*, pages 148-157. Conference on System Sciences, 1979.

[5] Imperial Software Technology.
ISTAR: Integrated Project Support Environment.
Internal Paper.
August, 1985

[6] Ralph L. London and R. A. Duisberg.
Animating Programs Using Smalltalk.
Computer 18(8):61-71, August, 1985.

[7] John R. Nestor, William A. Wulf, and David A. Lamb.
IDL - Interface Description Language.
Formal description.
1981

[8] S.K.Warren and D. Abbe.
Rosetta Smalltalk: A Conversational Extensible Microcomputer Language.
In *Proceedings of the Second Symposium on Small Systems*. October, 1979.

[9] N. Yankelovich, N. Meyrowitz, and A. van Dam.
Reading and Writing the Electronic Book.
Computer , October, 1985.

Table of Contents