NORTH-HOLLAND

Modeling Software Engineering Environment Capabilities

Marvin V. Zelkowitz

Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, College Park, Maryland; and Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, Maryland

There is considerable interest today in designing open systems that permit tools to be moved freely among various environments on different hardware platforms. To develop such systems, terms such as open systems, features for open systems such as interoperability, and integration must all be precisely defined. We present a model that is an extension of a service-based reference model for development environments that can be used to formally define these and other related concepts.

1. INTRODUCTION

Today there is considerable interest in developing integrated software engineering environments (ISEEs) that provide the software engineering professional with an effective set of tools for building software products. The purpose of this article is to present formal attributes of an environment that can be used as an aid in designing, building, and analyzing such systems. Starting with a reference model of a software engineering environment, various attributes of an environment can be defined based on that reference model. Terms such as *open systems*, *interoperability*, and *integrated software engineering environments* are often presented in the literature as desirable attributes for a system. It is our goal to address these terms based on the attributes we present here.

In Section 2 we explain the concepts of an open system and an integrated system and compare the two. In Section 3 we summarize a software engineering environment reference model that is the basis of the attributes we wish to develop. In Section 4 we present our model of environment attributes, and in Section 5 give our conclusions about this effort.

Address correspondence to Dr. Marvin V. Zelkowitz, Department of Computer Science, University of Maryland, Institute for Advanced Computer Studies, College Park, MD 20742.

2. OPEN SYSTEMS AND INTEGRATED SYSTEMS

As machines become cheaper, the balance between hardware and software costs is changing. Software often costs more than the price of the hardware. With the advent of relatively inexpensive hardware in the 1980s and the 1990s, hardware is rapidly becoming a commodity item with the corresponding need for software to run independently of the underlying hardware platform. Thus was born the concept of the open system. This can be intuitively described as any system composed of software components with interfaces that adhere to open system standards and should execute on any hardware platform that implements those standards. (We will be more precise later.)

2.1 Open Systems and Telecommunications

The term *open systems* originally developed in the telecommunications industry during the early 1980s. This era was characterized by the breakup of AT & T into several independent telephone companies, the emergence of additional long distance carriers, growth of the ARPANET (forerunner of the Internet), and great advances in communications technology from the earlier 300 and 1,200 bps modems to today's 14.4–38K bps to even higher data rates. Electronic mail was just emerging, as was the advent of the computer bulletin board and protocols such

J. SYSTEMS SOFTWARE 1996; 35:3–14
© 1996 by Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/96/$15.00
SSDI 0164-1212(95)00082-C

as file transfer protocol (FTP). There was great need for standards to be developed to permit products from a diverse set of providers to interoperate.

To meet these needs, the open system interconnection (OSI) model was developed around 1985 to manage information flow among communicating components. At the highest levels of the OSI seven-layer model, information transfers between two locations, and the model specifies the type of standard to describe this information flow. At lower levels, transferring mechanisms describe how information is broken down into data packets, until the lowest levels of the OSI model describe the electronic characteristics of this data communication.

With the creation of standards that meet the OSI requirements at a specific level, open systems has come to mean any product that meets the relevant standards. Being public standards, any vendor is free to build products that conform to those standards.

However, over time, the concept of open systems has broadened to include the computing platforms that are communicating, thus extending the concept of open systems to include the programming environment, which should also be specified by a set of publically defined interface standards. Building environment products for an open standard would allow easier entry for new vendors and foster a greater market for such products. For example, the National Institute of Standards and Technology (NIST) recently changed the name of its OSI Implementors' Workshop to OSE (Open Systems Environment) Implementors' Workshop to reflect the increased emphasis on open system environments.

2.2 Open Systems and Environments

We refer to an *environment* as a system supporting the execution of programs that solve problems in some application domain. We can talk about environments supporting business practices (e.g., support of accounting or payroll programs), environments supporting engineering design (e.g., CAD or computer-aided design environments), or environments supporting real-time applications (e.g., a system supporting programs that control on-line processes such as power plants and automated manufacturing). We are concerned here with environments used for the development of software (ISEEs), but other application domains can be handled similarly.

Open systems concepts applied to software engineering environments have progressed in several areas:

1. Although all UNIX implementations were based on the initial AT & T Bell Laboratories source code, variations and enhancements introduced by most vendors have resulted in incompatibilities among the various implementations. Although all of these systems "spoke" UNIX, the dialects were quite different, and software written for one version was generally not transportable to another. In 1984 the UNIX user group, /usr/group, began to develop a standard that would define what was meant to be "UNIX." This became "IEEE Standard 1003.1—Portable Operating System Interface for Computer Environments—POSIX," known simply as POSIX.1 (IEEE, 1988), which defines the kernel set of operating system services for such systems. IEEE standards committee P1003 is also working on other 1003.x standards that address other distributed system issues such as the shell user interface, real-time services, distributed data access, and other system functions.

2. The process by which the U.S. Department of Defense developed a common language for building embedded applications (i.e., the Ada language) in the early 1980s also increased awareness of the need for a supporting environment in which to build Ada applications. Starting with the "Stoneman" requirements document (Buxton, 1980), the concept of an Ada programming support Environment (APSE) was conceived (Figure 1). The APSE would consist of a kernel set of functions (kernal APSE or KAPSE) and a set of tools that would operate on that kernel.

3. During the early 1980s, European interest in environments took the form of a public tool inter-

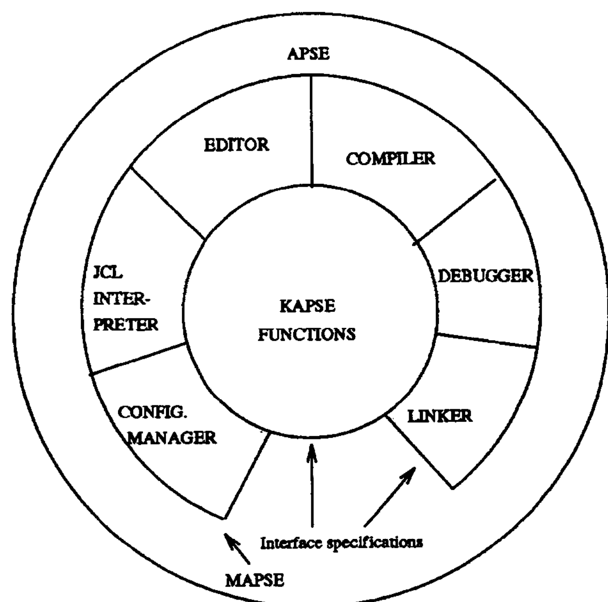

Figure 1. Ada programming support environments.

face (PTI) for specifying the interfaces for tools in an environment. An ESPRIT (European Strategic Programme for Research and Development in Information Technology) project was funded in 1983 to work on such a PTI. This PTI became known as "A Basis for a Portable Common Tool Environment," or PCTE for short (ECMA, 1993). Since 1988, PCTE has been under development by Technical Committee 33 (TC33) of the European Computer Manufacturers Association (ECMA).

4. Development of environments was a popular university research activity during the 1980s. Several ACM SIGSOFT annual conferences were devoted to software development environments (SIGSOFT, 1984, 1986, 1988, 1990, 1992).

Today we would define an *open system environment* as one that is composed of components, each of whose interfaces is fully specified by a public specification that was arrived at via a consensus process of an appropriate standardization body. The standard could be international (e.g., ISO), national (e.g., IEEE, ANSI), or even the result of corporate consortia (e.g., CORBA [Common Object Request Broker Architecture] by OMG [Object Management Group] or COSE [Common Office System Environment] by OSF [Open Software Foundation]).

2.3 Open Systems and Integration

A driving force for open system environments is the desire to permit tools to interoperate when procured from diverse sources. Tools should permit information to be transferred easily among themselves in order for information to be processed more effectively in the environment. This desirable attribute of allowing tools to pass information and control among themselves, or to *interoperate*, is somewhat independent of the concept of an open system. We also want tools to behave in a consistent manner by being integrated with the environment. By *integration* we mean a measure of this interoperability relationship among components of an environment (Thomas and Nejmeh, 1992).

Note that integration is a property between two (or more) components of an environment and not of the environment or tool executing on the environment itself. Therefore, it makes no sense to state "the user interface is integrated." However, one can state that "the compiler is integrated with the user interface," meaning that the user interface functions have a close association with the compiler functions.

Although we do not yet fully understand the appropriate design of an open software environment, we have begun to understand many of the components that make up the architecture for such an environment. An environment usually consists of a framework of core services that provides common facilities used by the other services in the environment. Appropriate end user services are then implemented to provide needed functionality for each application domain. For example, the framework may consist of common facilities such as a user interface (e.g., X Windows), a data repository (e.g., PCTE interface), and a communications facility (e.g., Sun's ToolTalk[1]). End-user services might consist of application tools like editors, compilers, spreadsheets, desktop publishing, etc.

The user's view of the environment is a system consisting of a collection of services that help to solve problems in some domain. With closed proprietary systems where interfaces are not based on open standards, a new embedded tool has to be tailored to execute within the framework provided by the enclosing system. Although the user sees an integrated environment in which to operate, these systems do not allow for easy integration of new tools into the environment, except by the environment developer.

Integration certainly exists to some extent in almost all systems today. One can almost always purchase two tools from a single vendor and have them interoperate effectively. For example, almost all of the major PC vendors (e.g., Microsoft, Borland, Lotus, Apple) provide packages of word processors, spread sheets, communication programs, and graphics packages that interoperate well. We would say that they were all integrated, but would be hard pressed to call all of them open since communication between the tools may be via some interface that is not part of a public specification. However, we are interested in developing open integrated systems where all components share common attributes. The incorporation of integration concepts into an open environment requires understanding of how integration relates to the set of services provided by the environment.

There are several notions of integration that affect the design of an environment (Brown et al., 1994):

[1] Certain commercial products are identified in this paper to specify adequately the applicability of the model. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

Conceptual integration. There is a shared philosophy about how environment components will interact in a consistent manner. "Look and feel" issues, common data formats, and common command sequences are examples of conceptual integration. Use of a common window, mouse, and command structure for all related tools is an example of a conceptually integrated system.

Architectural integration. How components are constructed to interact is mostly an architectural integration issue. Using open standards like PCTE for a data repository or X Windows providing a common window set of services describe architectural choices made in environment design.

Physical integration. This describes the actual components used to build the physical system from its abstract design. For example, although a single standard may be specified (e.g., PCTE for a data repository), it may be necessary to purchase a single instance of that product for all tools to cooperatively use in order to affect good data integration. For example, the use of TrueType fonts in Microsoft's Windows product permits new fonts to be added easily and to readily be made available to all products that use these fonts. This would be a good example of physical integration. On the other hand, on UNIX systems, although Postscript is a relatively common display format for documents, most postscript processors must maintain their own font libraries, resulting in hundreds of millions of bytes of storage being used for duplicate font directories. Although the system may appear to have conceptual integration from the user's point of view, this would not be physical integration.

To date, most integration efforts have centered on architectural integration (with some attention to conceptual integration) in order to design systems that have some degree of interoperability. Key integration areas are generally identified as follows: (Wasserman, 1989)

Data integration. Data intergration is the ability to share information throughout the environment. Different tools and services within the environment have their own requirements to access and share data. A high degree of data integration may mean that the tools in the SEE use a common database with common schema. Other degrees may include using common data formats or using translation mechanisms.

Control integration. Control integration is the ability to combine the functionalities offered in an environment in flexible ways by allowing tools to invoke (or enact) other tools. The combinations may correspond to project preferences and be driven by an underlying software process model.

Presentation integration. Presentation integration is the ability to interact with environment services to provide similar screen appearance and similar modes of interaction.

Process integration. Process integration is the ability to access environment services based on a predefined enactable development process.

In addition, others have written about *method integration* as an extension of process integration into life cycle activities and *platform integration* as the integration of tools to execute on the same hardware, as well as other forms of integration.

We have discussed both open systems and integrated systems as proposed solutions to the underlying problem of interoperability. What is the relationship between these two concepts? Although some claim that they are the same, we have tried to show that they refer to complementary properties of an environment. That is, a system may or may not be open, and at the same time may or may not be well integrated. A system well integrated in one dimension (e.g., all tools use the Motif X Windows interface for presentation integration) may not necessarily be well integrated in another dimension (e.g., each tool has its own proprietary database for data storage).

Openness is a structural attribute of a system. In an open system we are concerned about the interfaces among the system's components. We need to ensure that the interfaces among these components are defined by public standards. On the other hand, integration is a behavioral attribute of an environment. We need to specify how information is passed among environment components and how each component interprets the data it receives. To date, the two concepts have only general definitions. It would help the development of this technology if we could be more explicit about what these terms mean and to be able to determine when we have it and when we do not. This is the major goal for the model presented in Section 4.

Given the need to develop integrated systems, how do we go about the process of defining our requirements for such a system? In the following section we propose that by starting with a service-based environment reference model we can address such requirements.

J. SYSTEMS SOFTWARE 1996; 35:3–14

3. ENVIRONMENT REFERENCE MODEL

As already described, there is a growing trend away from proprietary solutions for computer-based problems and toward standardized open systems solutions for such problems. Providing a framework of services to support applications leads to the obvious question of what are the set of services and what interfaces are needed to support user applications in an open system environment. What is the underlying model for an environment?

After ECMA/TC33 began to develop the PCTE specification in 1988, there was considerable interest in determining how well PCTE met its objectives. However, there were no models in 1988 by which to measure PCTE. TC33 created a Task Group on the Reference Model (TGRM) to create such a model. In 1991, NIST joined with TC33 to develop this model, and Edition 3, known as the NIST/ECMA[2] software engineering environment frameworks reference model is the current edition (NIST, 1993).

The NIST/ECMA model, also known as the "toaster model" based on a graphic that was developed by George Tatge of Hewlett Packard (Figure 2), defines the underlying infrastructure set of services for supporting tools executing on a software engineering environment. The model consists of 66 services catalogued according to the classification of Figure 2 plus a seventh operating system set of services that supports the other six categories (see Appendix A). Software products, called tools, are added to the environment by being written to use the services from the seven classes of infrastructure services.

For each of the 66 services, a set of operations may be defined. For example, there are 21 services for data repository functionality. These include data storage (persistent data), back-up services, archive services, relationship among data object services, query services, and metadata (schema) definition services, plus others. For the persistent data storage service, for example, we would need the operations of access data, store data, modify data, delete data, and query data. How these are implemented is not specified by the model; each implementor desiring this service is free to implement it in any feasible manner.

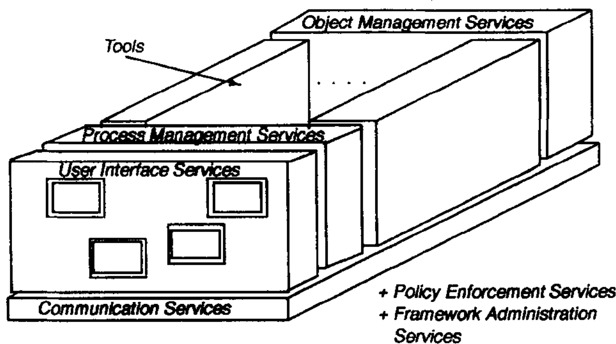

Figure 2. Framework reference model service groupings.

[2] ECMA/NIST in Europe.

To address the functionality that tools provided to aid users in solving software development tasks, the U.S. Navy's Next Generation Computer Resources (NGCR) program set up the Project Support Environment Standards Working Group (PSESWG). PSESWG developed the Project Support Environment (PSE) reference model of the set of services needed to support users of software engineering environments (Brown et al. 1993)[3]. This model included the NIST/ECMA framework model as the core set of framework service and puts a structure on the "tool slots" of the framework model.

This reference model is a description of the functionality that may be provided by an environment. This general description is not bounded by either a particular application domain or, unlike other models [e.g., Perry and Kaiser (1991)], by any process model for a specific project. It is a serviced-based model defining the catalog of functions appropriate for software engineering environments. Services are abstract capabilities of the environment, tasks make use of and provide context for those capabilities, and tools are the actual executable software components that realize environment services. (Appendix A briefly summarizes these services.)

Services are either end user or framework. The former services directly support the execution of a project (i.e., services that tend to be used by those who directly participate in the execution of a project, such as engineers, managers, and secretaries). The latter services generally pertain to the operation of the computer system itself (e.g., a human user performing such tasks as tool installation) or are used directly by other services in the environment. End-user services are further subdivided into technical engineering, technical management, project management, and support service categories. The first three of these groups partition the execution of a project into engineering, management, and a middle cate-

[3] The PSESWG report uses the term "Project Support Environment." We can assume it means essentially the same thing as an ISEE.

gory consisting of services used by both, and generally cover the set of tasks that are applicable for the development, management, and maintenance of software. The fourth group, support services, is complementary to the other three, since it includes capabilities potentially used by all other users, such as word processing, mail, and publishing, and should apply to essentially any computer-based application domain.

Figure 3 represents an intuitive view of the various service groups. Framework services form a central core with a potential relationship to all other services in the environment. Support services underlie the other end-user services. The remaining three groups generally are implemented with interfaces to the framework services and make use of the support services. The boundaries in the figure are not intended to be interfaces between functionalities in an environment. The reference model is a conceptual, not actual, model, and no architectural choices are intended by this figure.

The boundary between service groups, particularly the boundary between end-user and framework services, is dynamic and changes over time. There is a general tendency for greater functionality to be gradually assumed by the underlying framework. For instance, historically, most operating systems have included a directory structure and file system for data storage; a relational database is only occasionally added to a basic operating system. In the future, however, relational database functionality may be part of every operating system. Similarly, operating systems usually had only primitive display management operations. Since 1988, the MIT Consortium X Window System has become a popular graphical user interface (GUI) so that today it is usually included in every environment framework as a standard GUI to use. In the not too distant future, it probably will be a primitive operating system set of services included with every hardware platform.

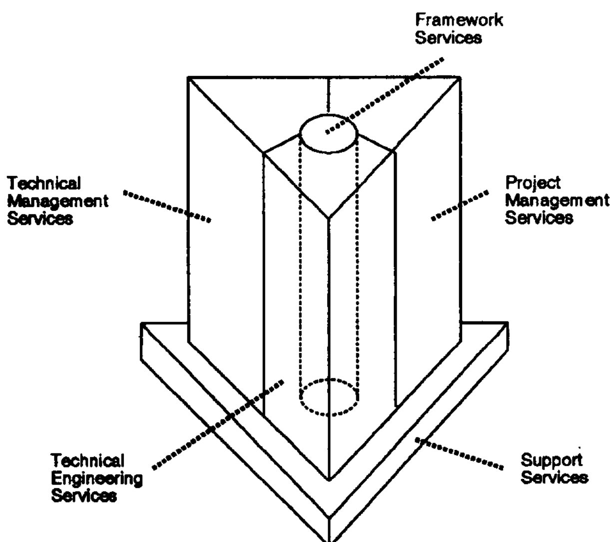

Figure 3. PSE reference model service groups.

The model has been used to map (e.g., describe and contrast) the functionality of various products or standards in order to determine how the functionality they provide compares to the functionality present in the model (Brown et al., 1992; Zelkowitz, 1993). It is our object to extend this concept as a way to define certain environment properties, as given in Section 4.

In what follows, we use the classification of services in the PSE reference model as a means to characterize the functionality of a software engineering environment. However, all that is really required is that we have a service-based model of an environment. In the discussion that follows, we could easily replace the formalism of the PSE model with any other appropriate model; however, the PSE model seems sufficient for our purposes at this time.

4. SOFTWARE ARCHITECTURES

We can use the PSE reference model of the previous section to develop the requirements for a software engineering environment. The first stage in system design is to specify the functionality that is desired by indicating which services of the reference model are to be included in the environment. This provides a taxonomy for describing the functionality desired and provides a mechanism for comparing alternative tools or standards for providing those services.

In describing architectures, we need to know how well two tools, standards, or products address the set of services that are required. The domain of services we consider is defined by the following two vectors:

The *reference model mask* (RMM) is a bit vector listing all services in the reference model. Similarly, the *reference model operations mask* (RMOM) is a vector listing all possible operations, remembering that each service can be implemented by a set of operations. The sample set of operations in the reference model is an initial guide to this set, and may be extended as necessary. X_i refers to the ith component of vector X.

A *specifications mask* is a bit vector that is a mapping onto RMM giving a subset of the services of the reference model (e.g., a "1" signifying that the service is so indicated) needed to support some

environment requirement. The specification mask may be the entire model, one service, or a set of services (e.g., the framework user interface services). This provides the context we are interested in. If we are looking for an appropriate design of the user interface, we might use a specification mask consisting of the user interface services; if we want to describe software development practices, we might use a specification mask of the software engineering end-user services. In the definitions that follow, there will always be a specifications mask that limits the context of what we are discussing.

A *service mask* is the functionality provided by a given product. It, too, is a bit vector mapping onto RMM. It provides the gross functionality of a given tool.

Similarly, we describe an *operations mask* as a mapping of the operations of some product onto RMOM. This permits us to describe the set of operations provided by a product in order to discuss functionality at a lower level of granularity than just using the service mask. We may also develop a *specifications operation mask* giving the set of operations needed in a specification.

Table 1 summarizes these masks in terms of vectors that define the requirements for a product and vectors that define the functionality provided by a product.

Table 1. Reference Model Functionality Vectors

Granularity	Requirement	Product functionality
RMM	Specifications mask	Service mask
RMOM	Specifications operations mask	Operations mask

4.1 Equivalence of Products

Functionally equivalent. Two products (e.g., standards or tools) are functionally equivalent with respect to a specifications operation mask (i.e., the operations of interest) if they support the same set of operations with respect to that specification mask.

If S is a specifications operations mask (i.e., S specifies the set of operations in a requirements document), and if A and B are the operation masks for two products, then A is functionally equivalent to B relative to S if $S \wedge A = S \wedge B$, where $\wedge$ stands for the bitwise "and" operation.

Conceptually equivalent. Two products (e.g., standards or tools) are conceptually equivalent with respect to a specifications mask S (i.e., the services of interest) if they support the same set of services with respect to that specification mask (e.g., $S \wedge A = S \wedge B$ for service masks A and B).

Conceptual equivalence is weaker than functional equivalence because each product may support a different subset of the operations of a service and so may be conceptually, but not functionally, equivalent. In general, competing products tend to be conceptually equivalent because they generally provide similar functionality, but replaceable components need to have identical interfaces and be functionally equivalent. For example, most word processors support functions of right justification, pagination, and font alterations (conceptual equivalence), but may implement these using different user commands (e.g., not functionally equivalent).

We can further subdivide functional equivalence into two interesting subsets:

Semantic equivalence. Two operations are semantically equivalent if they have the same or different syntax, but perform the same function. For example, there may be a differing number of parameters, or the parameters may be in a different format. The Ada and C bindings for PCTE should be semantically equivalent because they should support the same functionality on the PCTE repository, but the Ada and C function calls will have a different syntax.

Syntactic equivalence. Two operations are syntactically equivalent if they have the same syntax but may have differing effects (e.g., semantics). For example, many languages have read commands as *read(filename, object)*, with differing semantics imposed by those languages.

4.2 Interfaces

When we develop specifications for a tool, we generally provide two sets of interfaces—the services the tools provide to the environment and the set of services needed to implement the tool:

Functional interface. The functional interface defines the set of operations that the tool implements. It is a specification operations mask on RMOM. In general the mask will define operations from the set of end-user services of the reference model.

Development interface. The development interface defines the set of operations needed to implement the tool. Although it is also a specification operations mask on RMOM, in general it defines a set of operations from the framework set of opera-

tions needed to implement the tool. For tool A, we refer to the development interface as D_A.

These two concepts are significant when we discuss integration. Two tools can interoperate if the development interface of one is compatible with the functional interface of another. That is, one tool provides the framework needed to support the execution of another tool. However, integration is concerned with two tools using similar development interfaces (e.g., accessing similar functional interfaces from the supporting framework).

Given specification mask S and development interface T, two tools, A and B, are *interchangeable* with respect to masks S and T, if services specified by S are conceptually equivalent, and operations specified by T are semantically and syntactically equivalent. That is, (1) $S \wedge A = S \wedge B$ (conceptual equivalence in the functional interface); (2) $T \wedge D_A = T \wedge D_B$ (functional equivalence in the development interface); and (3) if $(T \wedge D_A)_i = 1$ then the operation specified by D_{Ai} is semantically and syntactically equivalent to the operation specified by D_{Bi} (Figure 4).

4.3 Open Criteria

This provides a way to address the concept of an open system. We want two tools A and B to be interchangeable with respect to their interfaces with other tools under open criterion C, an operations mask. T provides an *open interface* with respect to interface C if A and B are interchangeable with respect to development interface $T \wedge C$ (and some specifications mask S). That is, the development interface T restricted to the operations in C are the "same" in A and B.

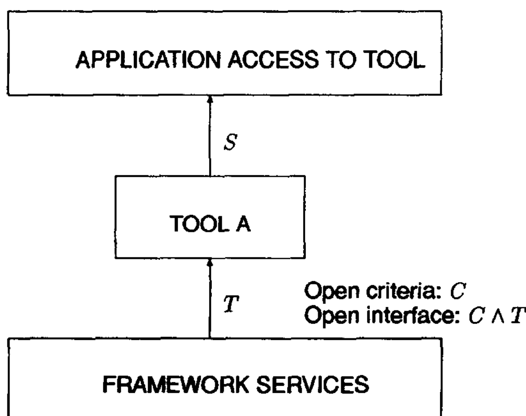

Figure 4. Interchangeable tools and open systems.

Because "open" implies a consensus-based standard, this definition only addresses the technical issues of defining such an open system standard. It does not address the social process of developing such a consensus-based standard within an appropriate standards body.

The term *open system* has often been discussed as a binary decision—a system is either open or it is not open. However, a system has many interfaces, and it makes sense to talk about the degree of openness with respect to some of these interfaces. If we wish to specify a tool (e.g., an editor) and a repository as separate components in an environment, then the editor's development interface (i.e., the open criterion C mentioned above) will not include repository functionality. This permits the repository to be an embedded component of this tool, and the system will correspondingly not be open with respect to this repository. However, if we specify in the editor's interface the appropriate repository functions as part of open criterion C, then we force the repository interface to be open and replaceable by another interchangeable repository component.

For example, if E is the operations mask for an editor and R is the operations mask for a repository, then a system will have an open editor interface if it meets open criterion E, an open repository interface if it meets open criterion R and it will have an open editor and repository interface if it meets open criteria $E \wedge R$.

4.4 Integration Criteria

On the other hand, integration demands other attributes from our reference model. For two tools to be integrated, we need to determine how well they interoperate with respect to some specification.

Let A and B be two tools. If we want to determine how well integrated they are with respect to some operations specification mask T (e.g., user interface functions, data sharing), consider D_A and D_B, which are the development interfaces for A and B, respectively. $D_A \wedge T$ and $D_B \wedge T$ give the set of functions needed by both tools. We would like D_A and D_B to be semantically equivalent with respect to T (Figure 5).

This is not quite sufficient, however. We need to ensure that data are used in the same way by both tools. Standards that define interfaces can be used in three ways: (1) a service interface (e.g., a set of operations such as the POSIX function calls in defining the UNIX kernel); (2) a data format (e.g., the ASCII character codes); or (3) a protocol (e.g.,

the X.25 mail protocol). The reference model mainly addresses the first of these interface classes, although the second of these is partially addressed via the metadata and data interchange services of the object management services of the framework reference model.

For integration we need to specify the semantics of data passing through these interfaces as well as the protocol governing the sequence of operation calls between the two tools. Modeling these protocols is still an important research topic.

We can also precisely define the domain of interest when we discuss integration. If we are looking at two editors that access the repository, then our specification mask S will be the set of object management services needed to maintain edited objects. Two editors may be interchangeable (e.g., vi and emacs both support the same interface to the UNIX file system) but provide very different sets of user commands, because the user interface was not of concern. On the other hand, if we wish to be concerned about user interfaces, our operations specifications mask would consist of the user command set.

The relationship between open and integrated can be demonstrated by comparing Figures 4 and 5. For integration we need the technical aspects of an open interface, and we need to add the semantic equivalence of two tools that need to interoperate. Openness is a property of the interface between two tools (Figure 4), whereas integration is a property between two tools (Figure 5).

Transportability is the ability to move a given tool among a large number of systems. Define the transportability mask T to be the operations mask of (generally, but not limited to, framework) operations on system **T** (e.g., the specifications needed to support tools on system **l**). Let I be the development interface needed to implement tool **I**. **I** will be *transportable* to system **T** if $I \wedge T = I$ (i.e., **I** uses only a subset of the available framework services). Or stated another way, **I** is transportable to system **T**

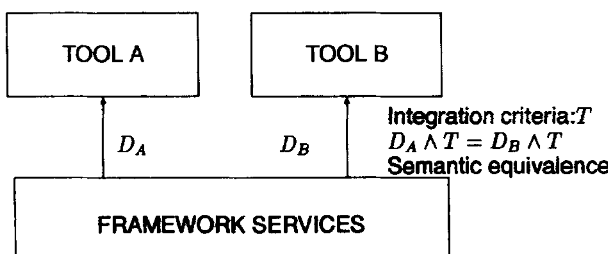

Figure 5. Integration interfaces.

if system **T** is open with respect to open criterion I. If $I_i = 1$ and the operation represented by I_i is syntactically and semantically equivalent to the operation represented by T_i, then the tool **I** can be moved virtually unchanged to system **T**. Otherwise, tool **I** must be modified so that I_i executes on system **T** as appropriate.

This definition delineates the scope of transportability. Only those operations defined in the transportability mask are under consideration. For example, consider the case of a PASCAL compiler. If we view the transportability mask as simply the operation of producing object code, then any PASCAL compiler that accepts the same source program and produces an equivalent object code format would be interchangeable with this compiler. All internal data formats would still be proprietary to that compiler.

On the other hand, if the transportability mask includes operations for all of the compiler switches (e.g., preprocessor, list symbol table, optimize code, produce listing), then only products that implement these additional operations would be interchangeable with the original compiler. This more detailed mask has "opened up" the PASCAL compiler so that additional products could replace parts of the compiler without the need to replace the compiler as a whole.

Note that this demonstrates that transportability and interoperability are different concepts. Interoperability requires transportability properties as well as the semantic equivalence of the transportable operations with another tool.

4.5 Example Use of Model

Table 2 presents an example use of this model. The table presents the specification operations mask for two spreadsheet products (S_1 and S_2). Services of the reference model, which were not present in either tool and not part of the following analysis, were deleted from the table to conserve space. An x in a "Services" row means that all operations under that service are provided.

Columns R_1 and R_2 represent two functional interfaces (specifications) for the spreadsheets (e.g., what functions are needed), and columns D_1 and D_2 represent the development interface for each (e.g., what functions they need in order to be implemented). DI is a hypothetical specification for an environment framework, which could represent a new platform on which both spreadsheets are to be transported.

According to Table 2, spreadsheets S_1 and S_2 are functionally equivalent with respect to R_1 because

Table 2. Example: Spreadsheet Mappings

Reference Model Services and Operations	S_1	S_2	R_1	R_2	D_1	D_2	DI
Estimation service	x	x	x	x			
Risk analysis service	x			x			
Text processing service							
Operation: create, edit, save, import text	x	x					
Operation: format or print text	x	x					
Operation: create and manipulate textual table	x						
Numeric processing service	x	x	x	x			
Figure processing service	x	x	x	x			
Audio and video processing service	x						
Presentation preparation service	x						
Calendar and reminder service	x						
Tool installation service							
Operation: customize tool	x						
Operation: create a test environment	x						
PSE status monitoring service							
Operation: log actions and events							
Operation: produce report on PSE usage	x						
Operating system services	x	x			x	x	x
Object management services	x	x			x	x	x
Policy enforcement services	x	x			x		
User interface services	x	x			x	x	x
User command interface services	x	x			x	x	x

they both implement the same subset of R_1 operations, but they are not equivalent with respect to R_2 because S_1 provides a risk analysis interface, but S_2 does not.

S_1 and S_2 are potentially interchangeable with respect to interface *DI* because they both have the same set of services in common with *DI*. One still must check, however, whether those operations are syntactically and semantically equivalent in order to have interchangeable components. S_2 will be easily transportable if operations in $S_2 \wedge DI$ in the new system are semantically and syntactically equivalent to the similar functions in the original system. However, transporting S_1 to D_2 would require adding policy enforcement services to support S_1 because such services are not provided by development interface *DI*.

5. CONCLUSION

We have tried to clearly differentiate between the concepts of an open system and an integrated environment. Both attributes are highly desirable in system design, yet each is separate and can be provided independently of the other. Our major goal was to bring a degree of precision to describing which kind of system we have and to be able to develop objective measures based on which we can decide if our systems have these attributes.

Starting with a service-based functional model of a software engineering environment, we developed definitions of many commonly used terms, such as open systems, interoperability, transportability, and integration. By using a vector of services based on this reference model of the application domain, several key ideas, such as functional equivalence, interchangeability, and transportability were given more precise definitions. Using these definitions, we were able to define both open and integrated environment concepts.

Using the concept of service masks and the service-based model, we have defined concepts such as open interface and transportability to be attributes of the interface between a tool and the framework, whereas concepts such as integration, interchangeability, and interoperability are attributes that reflect the interaction of two tools with respect to an underlying framework.

The services included in this reference model tend to be discrete, describing functionality that is explicitly invoked to provide a unique stand-alone service. There are, however, some services that tend to be *ubiquitous*, in which case their functionality is implicitly invoked and whose influence permeates many other services. A significant example of ubiquitous services can be seen in policy enforcement (e.g., security) services. Although these services are described in a separate section, their operation can affect most other services in the model, even though that interaction may not be documented in the service descriptions. For example, mandatory access control (i.e., who may access data) permeates every service that accesses the data repository.

A different example of ubiquitous services lies in integration services and is at the heart of the problem in defining the degree of integration in an environment. Sharing information among the services of an environment is directly related to the degree of integration that the environment exhibits. Although integration is recognized as an important aspect, there are few actual integration services that can be separated as discrete, and the role of the model presented here is to try to ferret out such integration ideas. Data interchange within the framework's object management services and some of the life cycle process engineering services are known to be related to integration, but the complete set of services needed to appropriately develop a fully integrated environment is still an important research topic and not fully understood today.

The use of reference models as a tool toward environment design is still a relatively new concept. Although we limited our scope here to software

engineering environment design, there is nothing in the design process described in Section 4 specifically for ISEEs. Most of the framework services of the reference model are applicable to other application domains, and it would be relatively easy to build service-based models for other application domains. Beginning with other service masks (RMM and RMOM), the design process is general across many design areas.

ACKNOWLEDGMENTS

Research support for this work was partially provided by grant 5-123 from National Aeronautics and Space Administration/Goddard Space Flight Center to the University of Maryland. The author would like to acknowledge the many discussions with Alan Brown, David Carney, and Tricia Oberndorf, all of the Carnegie Mellon University Software Engineering Institute, that led to some of the ideas expressed here.

REFERENCES

Brown, A. W., Earl, A. N., and McDermid, J., *Software Engineering Environments*, McGraw-Hill International, 1992.

Brown, A., Carney, D., Oberndorf, P., and Zelkowitz, M., eds., The Project Support Reference Model, Version 2.0, National Institute of Standards and Technology, Special Publication SP 500-213, 1993. (Also CMU SEI TR 93-TR-23, November, 1993.)

Brown, A., Carney, D., and Oberndorf, P., Practical evaluation of software engineering environment technology, Software Technology Conference, Salt Lake City, UT, 1994.

Buxton, J., Requirements for Ada Programming Support Environments "Stoneman," U.S. Department of Defense, 1980.

ECMA, Portable Common Tool Environment, Edition 2, ECMA 149, Geneva, Switzerland, 1993.

IEEE, *IEEE Standard Portable Operating System Interface for Computer Environments—POSIX*, Standard 1003.1, IEEE, New York, 1988.

NIST, Reference Model for Frameworks of Software Engineering Environments Special Publication 500-211, National Institute of Standards and Technology, 1993. (Also ECMA TR 55, Edition 3, 1993.)

Perry, D., and Kaiser, G., Models of Software Development Environments, *IEEE Trans. Software Eng.* 17, 283-295 (1991).

SIGSOFT, *Proceedings of ACM SIGSOFT Symposium on (Practical) Software Development Environments*, 1984, 1986, 1988, 1990, 1992.

Thomas, I., and Nejmeh, B., Definitions of Tool Integrations for Environments, *IEEE Software* 9, 29-35 (1992).

Wasserman, A. I., Tool integration in software engineering environments, in *Software Engineering Environments* (F. Long, ed.) (*Lecture Notes in Computer Science* 467), Springer-Verlag, Berlin, 1989, pp. 137-149.

Zelkowitz, M. V., Use of an environment classification model, in *ACM/IEEE 15th International Conference on Software Engineering*, Baltimore, MD, 1993, pp. 348-357.

APPENDIX: ENVIRONMENT REFERENCE MODELS

A.1 PSE Reference Model End-User Services

Each of the end-user service categories (technical engineering, technical management, project management, and support services) of the PSE reference model (Brown et al., 1993) is further subdivided by engineering domain, user role, or life cycle phase.

Technical engineering services focus on the technical aspects of project development. These services support activities related to the specification, design, implementation, and maintenance of systems. They are subdivided into system engineering and software engineering services. System engineering services includes services such as requirements engineering, system design and allocation, simulation and modeling, static analysis, testing, integration, reengineering, host-target connection, target monitoring, and traceability. Software engineering services include services for requirements engineering, design, simulation and modeling, verification, generation, compilation, static analysis, debugging, testing, build, reverse engineering, reengineering, and traceability. In addition, there are life cycle, engineering services for managing the process model of the development environment.

Technical management services include the following services: configuration management, change management, information management, reuse management, and metrics.

Project management services include management functions such as planning, estimating, risk analysis, and tracking.

Support services include those facilities needed by all users of an environment, such as text processing, numeric processing, figure processing, audio and video processing, calendar and reminder, annotation, publishing, mail, bulletin board, conferencing, and administration services.

A.2 Framework Reference Model Services

Framework services of the NIST/ECMA Frameworks Reference Model (NIST, 1993) comprise the

infrastructure of an environment. They include those services that jointly provide support for the end-user services given in the previous section. The following is a brief overview of the 66 framework services, as organized by service groupings:

Operating system. These services provide the primitive control of the underlying operating system by providing facilities for creating low-level processes, low-level I/O, and low level synchronization among the components of the environment.

Object management. These services concern the definition, storage, maintenance, management, and access of object entities and the relationships among them. Object management includes facilities for creating a database of objects, establishing relationships among different objects, information transfer through common metadata, as well as maintenance services such as archiving, backup and versioning.

Process management. These infrastructure services support the end-user life cycle management services by defining processes and mechanisms for controlling the execution and library management of such processes.

Policy enforcement. The reference model uses the term *policy enforcement* to cover the similar functionality of security enforcement, integrity monitoring, and various object management functions such as access control. It includes both integrity and access control attributes.

User interface. User interface services includes the connections between the user and the environment. Although emphasizing terminal and window display mechanisms, they include additional services that address multimedia issues such as mouse input, sound, video, and handwritten text.

Communication. Communication services need to provide two-way communication among the components of an SEE. This may include sharing of data as a means of communication as well transmission mechanisms such as the remote procedure call and messaging system.

Framework administration. This involves management of the framework to monitor users, tools, and resources of the framework.