SPECIFICATION-BASED COMPUTING ENVIRONMENTS

R. Balzer, D. Dyer, M. Fehling, S. Saunders

Information Sciences Institute

This paper considers the improvements that could result from basing future computing environments on specification languages rather than programming languages. Our goal is to identify those capabilities which will significantly enhance the user's ability to benefit from the computing environment.

We have identified five such capabilities: Search, Coordination, Automation, Evolution, and Inter-User Interactions. They will be directly supported by the computing environment. Hence, each represents a "freedom" that users will enjoy without having to program them (i.e. be concerned with the details of how they are achieved). They form both the conceptual and the practical basis for this computing environment.

SPECIFICATION-BASED COMPUTING ENVIRONMENTS

Traditionally, our computing environments have been based on operating systems. Each tool or facility existed as a separate subsystem and communication was via files. Unix allowed these files to be in-core ports and provided an interconnection language. These systems created rigid boundaries around each tool (subsystem) preventing integration, and provided only narrow low-bandwidth communication paths.

More recently, computing environments have been based on programming languages such as Lisp and Smalltalk. These languages provide a much wider channel between separate tools, and hence, foster tighter integration among those tools. Any object, or set of objects, definable in the language can form the interface between tools. Furthermore, the full set of control structures of the language can be used for initiating interaction. Finally, the fact that these tools exist in the language of the computing environment, allow them to be modified and extended through the programming language. This extensability, which enables user code (or another tool) to be strategically interspersed within an existing tool, contributes greatly to the success and popularity of these programming language based computing environments.

This research was supported by Defense Advanced Research Projects Agency (DARPA) contract MDA903-81-C-0335

In the hands of a wizard these computing environments are truly a thing of beauty. They are highly extensible, rapidly modified, and hence, adaptable to new situations. Unfortunately, these systems require wizards. They lowered the boundaries between tools, fostering interaction, but, as a byproduct, exposed huge amounts of detail. No mechanisms were provided for controlling this detail and keeping it consistent. Only through force of will could users (wizards) master this detail and maintain consistency.

This paper considers the improvements that could result from basing future computing environments on specification languages rather than programming languages. Our goal is to identify those capabilities which will significantly enhance the user's ability to benefit from the computing environment.

We have identified five such capabilities: Search, Coordination, Automation, Evolution, and Inter-User Interactions. They will be directly supported by the computing environment. Hence, each represents a "freedom" that users will enjoy without having to program them (i.e. be concerned with the details of how they are achieved). They form both the conceptual and the practical basis for this computing environment, for to the extent that we are successful in providing them as freedoms (specifications rather than algorithms), and hence lower the "wizard" level of users, we must provide corresponding automatic compilation techniques to keep this environment responsive, and hence, useable.

There are some obvious dependencies among these freedoms, and this decreases the number of mechanisms that will be needed to support them. These issues will be considered in the Implementation Basis section following consideration of the freedoms themselves.

Computing Environment Freedoms

Search

The main activity in a computing environment is building and manipulating various types of objects. Many of these objects are persistent - their lifetime exceeds, and is independent of, the programs that build and manipulate them.

For objects to be persistent, they must be stored somewhere so that they can be reaccessed later. Current storage and retrieval mechanisms are inadequate and require detailed programming. Files are neither appropriately sized nor adequately indexed to be used as containers for objects. External databases have strong limitations on the types of objects that can be stored [and on the manipulations that can be performed on stored objects]. Objects stored in a programming environment are idiosyncratically indexed and retrieved.

Consider instead an environment, based on the database viewpoint, which houses a universe of persistent objects within the environment itself and which provides descriptive access to those objects. That is, rather than using some predefined criteria, ANY combination of attributes, properties, and relations can be used to access an object (or set of objects if the request was not specific enough). Objects housed within the environment can be manipulated by the full power of that environment. Any modification causes them to be automatically reindexed for later descriptive reference.

This, of course, describes a fully associative entity-relationship database [Chen79] integrated with a programming language that creates and manipulates the objects in that database. All objects in the environment are represented in the database (a one-level virtual store) in terms of their relationships (including entity-class) with other objects. The only changes that can occur in this universe of objects are the database operations of creating and destroying object instances, and asserting or denying relationships between objects. By requiring all the objects of the environment to be housed in the database, by imposing a full associativity requirement on that database, and by expressing the services of the environment totally in terms of the object (i.e. database) manipulations they perform (that is, by integrating the processing with the database), users would be freed from having to predetermine how objects ought to be indexed so that they can be later retrieved, and from programming their retrieval from that predetermined structure. Much of the complexity and difficulty of using current environments arises from the care and feeding of such "access structures". In this proposed environment, any classification structure merely becomes additional properties of the object which can be used, like any others, as part of a descriptive reference to that object.

Coordination (Consistency)

Given the ability to create and manipulate persistent objects and to access them descriptively, the next most important capability is to coodinate sets of such objects -- that is, keep them consistent with one another. Whenever one object in such a coordinated set changes, the others must be appropriately updated. Currently, we attempt to realize such coordination through procedural embedding. That is, into each service that modifies such an object we insert code to update the others. Since the consistency criteria are not explicit, this currently is necessarily a manual task and is error prone, both in the placement and form of the required update. Such manual procedural embeddings are a key reason current systems are complex. This problem is exacerbated by the fact that the services, and the relationships among objects effected by these services, are evolving independently.

Consider instead making the coordination rules explicit so that coordinated objects are defined in terms of each other. Each definition is expressed in terms of a mapping (called a *perspective*) which generates a dependent object (called a *view*) from one or more objects with which it is coordinated. Whenever a coordinated object changes, the view can be updated automatically. Views are first-class objects: they can be accessed descriptively, and, if the back mapping is defined, they can be modified, causing the appropriate changes in the "defining" objects. (Some of these back-mappings can be inferred automatically, others are underdetermined and must be explicitly defined.)

Such coordination represents a major departure from existing systems. Coordinated objects are tightly coupled, so that changes in one are automatically reflected in the others. With such a mechanism, once the coordination criteria (mappings) are stated, the system could assume full responsibility for maintaining consistentcy among coordinated objects. Changes to existing services or addition of new ones could be accommodated automatically. Furthermore, the system could then employ lazy evaluation [Friedman76] to delay updating views until those updates were actually required.

The reason that the terms, *perspective* and *view*, were chosen ,respectively, for the mapping and the object produced is that, in addition to its intended use as the mechanism to keep objects coordinated, perspectives will also be used as the mechanism by which a user displays and manipulates objects. Displays are just particular views (which like other views must be kept coordinated with the object being viewed) for which the system knows how to create a picture on the user display screen and how user gestures (whether by entering text, making selections, and/or graphical motion) change the display (and hence, both the picture on the screen and, via a back mapping, the object being viewed).

Coordination is thus an extremely powerful mechanism. It not only provides an explicit mechanism for maintaining consistency between objects, but also provides the mechanism by which manipulatable filtered (i.e. partial) views could be constructed for both internal and external (display) use.

The user interface to this environment would therefore be a set of perspectives (mappings) used for display. Through them the user could observe objects, watch them change, invoke tools and services to manipulate them, or change them himself. This user interface would be fully programmable and extensible (see Evolution below).

As an example of the power of the coordination mechanism, justified text is just a view of text, and object code is just a view of source code. By defining justification and compilation as the perspectives which produce those views, these processes will be automatically invoked as needed. The maintenance task (coordinating the objects) will shift from the user to the system.

Automation

In interacting with a computing environment, many repetitive sequences are employed. Programming language based environments provide the ability to bundle such repetitive sequences as macros and/or procedures. But such macros and/or procedures still have to be invoked explicitly. The user is required to remain in the loop having to perform the pattern recognition function and determine when and upon which objects to invoke the macro and/or procedures.

By adding demons to the computing environment, users could be freed from being in-the-loop through automating the way that their environment reacts to specified situations. Those situations would become the firing pattern of the demons, and the responses become their bodies. This would allow users to define active "agents" operating on their behalf which autonomously monitor the computing environment for those situations for which a response has been defined. This freedom allows users to focus their attention on the more idiosyncratic aspects of the computing while their agents handle the more regularized ones. In particular, these agents could operate in the absence of the user, responding to interactions initiated from other user's environments (see Inter-User Interactions below).

This automation mechanism not only frees users from repetitive tasks, but also changes their perception of their environment. First, it emphasizes the data base orientation of the environment by basing responses on situations (the state of some set of objects) rather than on the processes (code) that produced those situations. As we will see in the next section, this data base orientation greatly facilitates evolution of the tools and services in the environment. Second, these responses convert the previously passive environment into an active one.

As an example of automation, consider an agent which responds to the arrival of a message by presorting it for the user into some predefined catagory on the basis of the sender, the topic, and/or the content of the message, and then decides whether to inform the user of its arrival based on the user's current activity.

Evolution (Perspecuity)

One of the key problems with traditional computing environments is the inability to modify the tools and services of those environments. Programming language based environments improve this situation by coding the tools and services in the language of the environment (with which the user is necessarily familiar) and by making the source code available to the user. To the extent that the user can understand the tools and services, he can modify them.

Once the commitment has been made to provide accessible source code, evolvability is almost completely an understandability issue. This is another way that adopting a specification-based approach has a big payoff. Besides alleviating implementation concerns, each of the specification freedoms improves understandability by allowing the code to more closely describe intent rather than implementation.

As a prime example, consider the use of the "automation" demons, described in the previous section, to provide situation-based extensions. Rather than procedurally embedding the extension at each appropriate place in the existing tools or services, a single demon is created that specifies when, in terms of the objects in the environment (i.e. a situation), the extension is appropriate. By localizing the extension and specifying the situation to which it is to be applied, the understandability of the resulting service is greatly enhanced. This paradigm has already been successfully employed for many years within Artificial Intelligence and production-rule languages. We believe it has much wider applicability.

But tool and service understandability need not be based solely on the readability of the source code. These tools and services manipulate objects in the environment. That is, they have behavior, and that behavior provides a strong basis for understandability. By making the behavior explicit in the form of a recorded history (as an object in the environment) the full power and extensibility of the viewing (coordination) mechanism could be used to understand the recorded behavior.

The recorded history would include attribution so that the old debugging problem of determining how an object reached its current state and who was responsible for it will finally be resolved.

Recording history is a major design commitment of our computing environment which provides the basis for its behavior based understandability. To the extent that we are successful in providing an evolvable, integrated, and automated computing environment, the need for such behavior based understanding will correspondingly increase.

The recorded history also provides the basis for an important habitability feature - the ability to undo operations [Teitleman72]. There are three reasons why such a capability is crucial. First, we are faliable - from lack of forthought or just plain carelessness. Second, no matter how consistent and well integrated the environment is, we will occasionally be unpleasantly surprised at the effect of an operation, or the situation in which it was invoked. Finally users need a convenient way to experiment to learn about unfamiliar services, to debug their own additions to the environment, and simply just to see the effects of some course of action. For all these reasons, an undo mechanism which can be invoked after the operation(s) to be undone, is a crucial habitability feature (as shown by its popularity and use in the Interlisp [Teitleman 78] environment). Such a facility can be easily constructed from the recorded history.

Inter-user Interaction

Our specification based environment has so far proposed the freedoms of search, coordination, automation, and evolution. These four freedoms resolve the major difficulties encountered *within* a computing environment. But our future computing environments cannot be self-contained. They must interact with the environments of other users and with various shared services.

As was the case when we considered persistent objects, files are an inappropriate mechanism (though they are the basis for existing inter-user interactions). Inter-user interactions require no less powerful nor rich a set of capabilities than those needed within a single environment. Objects need to be accessed, coordinated, and manipulated across environment boundaries. The boundary between environments has to be suppressed so that the full power of the computing environment can be applied to inter-user interactions.

One remaining issue must be addressed. Our rights and privileges are very different within someone else's environment from those within our own. Within our own environment, we can do as we please - accessing any object, manipulating it, and defining the rules of consistency which it must obey. Within someone else's environment, they have all the rights and privileges. We must ask their permission for anything within their environment.

We do this by dividing the notion of an active object [Kay74, Hewitt77] into an active intermediary (programmed agent) and a (passive) object owned by that intermediary. If we are manipulating (including accessing) an object that we own, then the manipulation is performed directly. However, an attempt to manipulate someone else's object is treated as (i.e. translated to) a request to the owner of that object which can be either honored or refused. This specification freedom enables object owners to define external access and manipulation rights that allow others to manipulate objects without respect to environment boundaries, as long as they don't exceed those rights. Privacy and/or access can be programmed on a local object-by-object basis and can be both state and requestor dependent.

Beyond Freedoms: General Support

In addition to the specification freedoms described above, two other capabilities must be available within the computing environment to simplify service creation and improve the habitabilty of the environment. First is a comprehensive set of general object manipulations. Since the main activity in any computing environment is building and manipulating objects, such a set of widely applicable object manipulations is essential. These manipulations include object definition (since the class of object types is not fixed), instantiation (since the set of objects of each type is not fixed), examination (often called browsing in interactive systems), modification, and destruction. To the extent that traditional services have employed idiosyncratic versions of these capabilities, providing a comprehensive set of widely applicable object manipulations will reduce service implementation effort while improving the consistency and coherency (and hence habitability) of the environment. As an example of such a reduction consider an electronic mail service. The only portions of this service which must be specially built are the definition of the object *message* and the mail service specific operations of sending a completed message (transferring a copy to each of its addressee attributes) and answering a message (partially constructing a message with the addressees and the beginning of the body ("In reply to your message of...") filled in). All of the other capabilities normally associated with a mail service such as comparing messages, examining them, editing them, filing them, retrieving them, deleting them, etc., are provided through the general object manipulation capabilities of the environment. Clearly, such reductions in the scope of service implementation greatly facilitate the creation of new services.

The second additional capability required within the computing environment is a suitable user interface. As previously discussed under the coordination freedom, the user interface will be a set of perspectives (mappings) used to display and manipulate objects. By defining a "service invocation" as an object, it can be instantiated, displayed, and manipulated by this interface, and by defining a service on such objects which invokes the named service on the specified objects (parameters), then this interface can be used as a "command interpreter" to specify the parameters needed for some service and to invoke it. In addition, since a wide variety of views will already be needed for user browsing, these same views can be used to display the effects of services. In fact, since all the effects of a service invocation are recorded in the history, a much more sophisticated display mechanism can be created, external to the services, which examines the effects and determines what to display based not only on these effects, but also the current user context including what is currently displayed on the screen and on various user declarations of personal preference. By removing both input (service invocation) and output (how to display effects) from service definitions, their scope will be reduced to a kernel consisting of only the functional object manipulation effects of the service. This will greatly simplify service creation while simultaneously providing a more powerful comprehensive user interface.

Implementation Basis

This computing environment has not been implemented. It is still in the conceptual design phase. We consider here the basis for our eventual implementation.

We have proposed quite an ambitious set of freedoms to resolve the difficulties that have made current environments so complex and hard to use. Each of these freedoms (and their combinations)must be compiled into efficient mechanisms to keep the environment responsive (and hence useable). Yet the set of mechanisms and compilation techniques is relatively small and, we think, manageable.

First and foremost, there is the issue of associative access. Objects can participate in, and be accessed via, arbitrary relations. Clearly, some set of internal inversion indices must be selected and maintained for rapid access. This means that object updates must also update the appropriate indices. Modularity concerns imply all such updating be encapsulated in a data base interface responsible for all object manipulation. Since much of the environment exists as code, it can be analyzed to determine which indices are (most) needed. Yet, since the environment must be responsive to direct user interaction, and evolution will occur in the existing code, it must also be adaptive. Thus, it requires combining traditional data base technology handling large statically-indexed data bases with programming language and Artificial Intelligence techniques for smaller dynamically indexed data bases.

Such a technology would directly support both the descriptive references realizing the search freedom, and the demon firing patterns realizing the automation freedom. The (data base) interface it imposes makes the addition of automatic history recording (required for the evolution freedom) and the active intermediary that owns remote objects (required for the inter-user interaction freedom) straightforward. Each of these latter two facilities has separate efficiency requirements, but they are relatively minor and the existing technology [Balzer69, Teitelman78, Kay74, Rashid80, Nelson81] seems adequate.

Only one freedom, coordination, remains unaddressed. Since this freedom is realized in terms of explicit mappings which define the coordination, it is trivial to apply them to obtain updated views. Unfortunately, much more than simply applying the mappings is required, and the needed technology does not yet exist.

This needed technology poses the second major implementation problem. It must address four issues. First, there is the question of determining when an existing view is obsolete. It is relatively simple to syntactically determine which changes could possibly affect a view. It is much more difficult (and in general undecidable) to determine which actually do affect the view. To the extent that this difference is undetected, views are needlessly obsoleted.

Second, rather than recomputing a view as soon as it is obsoleted, this computation can be delayed until the view is actually needed (Lazy evaluation [Friedman76]). To the extent that obsoleted views are never referenced, or not referenced before further obsolescence, needless compution is avoided. Since many of these computing environments are planned for personal machines, this lazy evaluation should become *opportunistic* evaluation, utilizing any otherwise wasted wait time to recalculate not yet required obsoleted views. Clearly some sort of priority mechanism either a-prior or adaptive would be required.

Third, while, in general, the back-mappings are underdetermined, many of them are not, and given suitable restrictions in the mapping language, they could be automatically generated.

Finally, and most importantly, even handling obsolescence and lazy/opportunistic evaluation appropriately still leaves a major efficiency problem. Minimizing the frequency of view recalculation does not mitigate the cost of each such view calculation. Once a view has been calculated, most updates should be dealt with incrementally, rather than recalculated from scratch. Since views will be quite pervasive, an incremental update facility will have a major effect on the responsiveness of the environment. Unfortunately, such a facility requires sophisticated analysis of the mapping language. Again, suitable restrictions on that language can have large effects on the feasibility of such analysis. As the amount of such technology is crucially dependent upon the detailed conceptual design, our general strategy is to suitably restrict the design to minimize the need, and gradually relax these restrictions as the technology comes into existence.

Conclusion

We have examined current computing environments and tried to understand the causes for their limitations, particulatly in the areas of integration and habitability. Operating system based computing environments must be integrated at the subsystem level. The narrow communication channel imposed via files (whether real or in-core) appear to fundamentally preclude tight integration.

The situation is very different for programming language based computing environments. They appear structurally ideal for tight integration. Arbitrary objects can be defined and shared. The full range of control structures in the programming language can be used to tie tools and services together. While this programming-language basis is adequate for integration it causes habitability problems. The mechanisms are simply too low level (detailed) for the computing environment task. Rather than describing what to do, user must program how to do it, precisely because they are dealing with a *programming* language.

The obvious solution is to augment the computing environment language with higher level *specification* constructs. Each such construct represents a *freedom* that users can enjoy (because they no longer have to program the construct) and a responsibility the system must accept to provide an efficient implementation of the construct to keep the environment responsive.

1. Search - the ability to locate objects via descriptive reference.
2. Coordination - the ability to state the consistency criteria among objects and to have it maintained as any of them are changhed.
3. Automation - the ability to define the autonomous response to specified situations so that the user need not remain in the loop for repetitive operations.
4. Evolution - the ability to modify and extend existing services through increased perspecuity of those services and their behavior.
5. Inter-User Interaction - the ability to determine how others will be allowed to access your objects, as they determine.

We have no doubt that such freedoms, together with a comprehensive set of general object manipulations and user interface capabilities, will greatly facilitate service creation and markedly improve the habitability of future computing environments. These freedoms must be supported with efficient mechanisms. Two mechanisms seem most crucial. The first is an adaptive associative entity-relationship database. This will require integration of techniques being developed in the database, programming language and artificial intelligence fields. The second is view maintenance. It requires the integration of techniques for obsolescence detection, Lazy (and opportunistic) evaluation, generation of back-mappings, and most importantly for incremental update.

The open question is how long it will take to provide this underlying support technology. We hope that others will reach similar conclusions - that the path to progress in computing environments lies in identifying appropriate freedoms - and join us in this project.

Acknowledgement:

The ideas presented here have evolved from the efforts and philosophy of the SAFE group at ISI, particularly the development of the formal specification language of GIST and the ability to map it via transformations into efficient implementations.

References

[Balzer 69] - "Exdams - Extensible Debugging and Monitoring Systems", Robert Balzer, Spring Joint Computer Conference, 1969, pp. 567-580

[Chen 79] - Proceedings of the International Conference on Entity-Relationship Approach to Systems Analysis and Design, Dec. 1979, Los Angeles, Peter P. Chen, editor

[Friedman 76] - "CONS should not evaluate its arguments", Friedman, D.P. and Wise, D.S., Automator, Languages, and Programming, Michaelson and Milner, eds., Edinburgh University Press, 1976, pp. 257-284

[Hewitt 77]- "Laws for Communicating Parallel Processes", C. E. Hewitt and H. Baker, Proceedings of IFIP-77, Toronto, Aug. 1977

[Kay 74] - "SMALLTALK, a communication Median for children of all ages", A. Kay, Xerox Palo Alto Research Center, Palo Alto, Calif. 1974

[Nelson 81] - "Remote Procedure Call", Bruce Nelson, Xerox Palo Alto Research Laboratory, CSL-81-9, 1981

[Rashid 80] - "An Interprocess Communication Facility for UNIX", Richard Rashid, Carnegie Mellon University, Dept. of Computer Science, March 1980

[Teitelman 72] - "Automated Programming - The Programmer's Assistant", Warren Teitelman, Proceedings of the Fall Joint Computer Conference, Dec. 1972

[Teitelman 78] - Interlisp Reference Manual, Warren Teitelman, Xerox Palo Alto Research Center, Oct. 1978