Software Process Modeling and Execution within Virtual Environments

JOHN C. DOPPKE, DENNIS HEIMBIGNER, and ALEXANDER L. WOLF
University of Colorado

In the past, multiuser virtual environments have been developed as venues for entertainment and social interaction. Recent research focuses instead on their utility in carrying out work in the real world. This research has identified the importance of a mapping between the real and the virtual that permits the representation of real tasks in the virtual environment. We investigate the use of virtual environments—in particular, MUDs (Multi-User Dimensions)—in the domain of software process. In so doing, we define a mapping, or *metaphor*, that permits the representation of software processes within a MUD. The system resulting from this mapping, called *Promo*, permits the modeling and execution of software processes by geographically dispersed agents.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development; software maintenance*

General Terms: Management

Additional Key Words and Phrases: MOO, MUD, PROMO, tools, software process, virtual environments

1. INTRODUCTION

Virtual environments (VEs) have typically been developed in the past primarily for their entertainment value. Indeed, the earliest virtual environments, called Multi-User Dungeons, were built as games involving the exploration of rooms containing various obstacles and opponents. Recently,

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under contract number F30602-94-C-0253. This work was also supported by the Naval Research and Development Division (NRaD) and the Defense Advanced Research Projects Agency under contract number N66001-95-D-8656. The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

Authors' addresses: J. C. Doppke, New Decade Software, LLC., 5924 Gunbarrel Avenue, Suite E, Boulder, CO 80301; email: doppke@ibm.net; D. Heimbigner and A. L. Wolf, Software Engineering Research Laboratory, Department of Computer Science, University of Colorado, Boulder, CO 80309; email: {dennis; alw}@cs.colorado.edu.

however, a movement has begun to create more general virtual environments, called Multi-User Dimensions (MUDs), whose main purpose is not gaming but rather the building of virtual worlds involving more general (and some would say, more positive) social interactions. While the earliest systems with this purpose were still intended for entertainment, the promise of an extensible virtual environment has intrigued researchers of collaborative work systems and has prompted questions about the viability of such systems in aiding work and collaboration in the real world.

We are investigating the use of MUDs in the particular collaborative work domain concerned with software development, called *software process*. MUDs appear to be a suitable context for performing software processes for two main reasons. First, VEs help make software processes—which are sometimes viewed as formal and forbidding control mechanisms—more accessible to users through their appeal to a familiar metaphor of operation. In current software process support environments, the metaphor is usually drawn directly from computer science. Typically, either a person views the environment and the actions carried out within it as a series of database transactions or as the execution of some state machine such as a Petri net. Second, the VE serves as a model for a convenient mix of synchrony and asynchrony within a process. Activities that take place simultaneously may take advantage of their simultaneity within the VE, but the VE does not require such synchrony. Similarly, the VE serves as a model for both centralized and distributed interaction.

In this article we examine the use of virtual environments in support of software process modeling and execution. We begin in the next section by briefly reviewing the domain of software process in terms of its primitive concepts of activities, agents, artifacts, resources, and products. We then describe in Section 3 the central concepts found in MUDs, concentrating on the features present in one very popular MUD called LambdaMOO [Curtis 1997]. There are a variety of possible metaphors that can serve to capture software processes within MUDs. These are explored in Section 4. The implementation of one of the metaphors in a system called Promo is described in Section 5. Section 6 provides several interesting and important lower-level details of the Promo implementation. An example software process drawn from the literature is used in Section 7 to illustrate the modeling features of Promo; further details of the example are provided in Appendix A. We conclude with a summary and a look at future work.

2. SOFTWARE PROCESS

Software process has become a major area of research in software engineering [Finkelstein et al. 1994; Fuggetta and Wolf 1996]. One concern of that research is the representation of processes using a consistent, and often formal, *process model*. A second concern is the development of technologies to support the actual execution of processes. These technologies are often collected together into what are referred to as *process-centered environments*. In this section we review the basic concepts of software process in

order to provide a context for describing how software processes can be captured in MUDs.

2.1 Definitions

An *activity* (or *task*) denotes a sequence of one or more *actions* (or *operations*) executed by people or tools as part of a software process. Presumably, the definition of a particular activity is guided by an understanding of the semantics of the process. For example, we may consider "testing" an activity because we have a semantic understanding of how the specific actions within testing, such as generating test cases and running the program against test cases, fit together. The appropriate granularity of activities modeled within a particular process is an issue that must be left to the discretion of the process engineer. Adding to this flexibility would be an ability to define *subtasks* within larger tasks, thus forming a hierarchy of activities.

The definition of a particular activity need not be specific about what actions constitute the activity or in what order they occur. Moreover, the means of defining the activity depends greatly on the process-modeling language or languages being used. The definition of an activity may, for example, consist of a specification of the conditions that must hold before the activity is considered complete, rather than a prescriptive set of actions.

Actions within the process may or may not be understood to be atomic. For the sake of simplicity, we assume here that actions are indeed atomic. Actions may be grouped together into *transactions* to permit the atomicity of groups of actions. The execution of an action often requires the availability of an appropriate *tool*, although some actions (e.g., a decision made by a person) may not be associated with any tool.

The sequencing of activities is a key factor in how a process is executed. This sequencing is usually not linear. In fact, the sequencing of future activities usually depends on the state of the process that results from past activities.

Any discussion of actions and activities within a process raises the issue of *agents*: the humans or machines involved in carrying out an activity. When multiple people collaborate in carrying out a set of activities, the issue of who performs which activity is an extremely important question. Many accounts of software process include the notion of a *role*, a unit of functional responsibility [Curtis et al. 1992]. In such models, a role is assigned to each activity, and one or more of the human agents who are authorized to take on this role must do so in order to complete the activity. Other accounts of process [Heimbigner and Osterweil 1994] eschew this notion of roles because of its conflation of issues that should remain separate: threads of control, unification of similar activities, and access control.

Part of any process is the need to secure *resources*: those aspects of the process that are expended (e.g., time and money) or that may be otherwise

limited in some way. For example, carrying out a testing activity may require scheduled use of a testing laboratory. The activity may not proceed unless the laboratory is available. Given this definition, we may consider a tool to be an example of a resource, since it may be licensed and therefore subject to limitations on its use. A process description may delineate a set of policies dictating how resources may or may not be shared and whether resources are finite or infinite.

Software process governs the manipulation and definition of the *artifacts* within a system. That is, activities within a process must be carried out on pieces of the system being developed, and these pieces must be explicitly defined. Often the definition of artifacts entails the definition of a set of artifact types. Such a typing system may be used to guide the definition of the set of actions that may be taken on an artifact. Since artifacts often form parts of larger artifacts, their interrelationships must also be defined.

Finally, the process concerns the *product* itself. We may also choose to partition the product into *subproducts* in much the same way as we partition tasks into subtasks.

2.2 Process-Centered Environments

In order for a process-centered environment to support the modeling and execution of a process, it must first address all of the aspects of the process and model them in some convenient and systematic way. First, it must be able to represent all the entities listed above within the process. This representation need not be complicated. For example, a system may represent human agents by simply using user identifiers provided by the operating system. Typically, however, a representation of some process entity consists of an abstraction over the machine and operating system. For example, artifacts may be reified within the system in such a way as to obscure the actual files within the file system that contain the artifacts' data, as in Marvel [Kaiser et al. 1990]. This contrasts with process-centered environments such as Merlin [Junkermann et al. 1994] and SPADE [Bandinelli et al. 1994], which provide their own databases to hold the actual project artifacts.

Closely coupled with the representation of process entities is the manifestation of connections among these entities. For instance, a system must not only represent tools and artifacts, but also encapsulate the ability to invoke a tool on a specific artifact.

Finally, an environment must provide an interface through which the user may interact with the process. While the use of the word "interface" suggests a discussion of user interface (e.g., whether it is graphical or textual), the term is intended here to designate a whole set of methods through which the user may query and control the environment. The user may wish to know information about the current state of the process; for example, which human agent is carrying out which task with respect to which artifact(s); and the environment must provide a means of answering these questions. Furthermore, it must provide this means within the

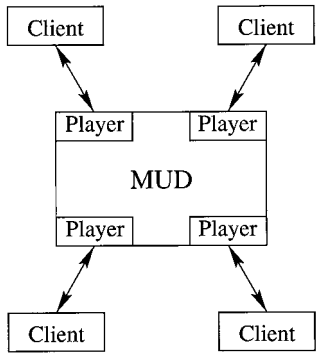

Fig. 1. Client-server architecture of a typical MUD.

framework of the representations chosen for the entities and their connections. Finally, part of this interface may—but need not—involve some proactivity on the part of the process, i.e., the environment may be designed to take actions on behalf of the user.

3. MUDs AND VIRTUAL ENVIRONMENTS

As mentioned above, MUDs were first created as simple games to be played by several users simultaneously. The term MUD generally refers to a system that permits multiple users to connect to it (via some network) and that presents a virtual world to these users in which each user is represented as a *player*. A diagram of this architecture is given in Figure 1. MUDs began as text-only systems, and nearly all are still essentially text based. However, many now are beginning to offer more sophisticated interfaces of various kinds [Curtis et al. 1995].

The world that the MUD provides to the user consists of a set of rooms and, within these rooms, myriad objects, including other players. The MUD's world represents space by means of spatial relationships among objects—for example, connections between rooms—but not specific distances or directions. This permits the system's description of the world and the user's traversal and manipulation of the world by simply textual means. MUDs thus differ from other virtual reality systems that wish to present an accurate three-dimensional (or even two-dimensional) view of the world. An example of interaction with a MUD is given in Figure 2.

Noticeable differences of opinion exist on the "point" of a MUD. While MUDs were begun as multiuser games in the spirit of adventure games (e.g., ADVENT), the concept of what constitutes a MUD has grown over time to encompass the general field of *virtual environments*. Accordingly, while many MUDs still exist primarily as games—and many MUD aficionados enjoy them for this reason—other systems have eliminated many of their game-oriented aspects and have focused instead on the creation and exploration of their virtual worlds. Systems of this latter kind are often as social in focus as they are technical. Many MUDs are devoted to a certain

```
>look
Corridor
The corridor from the west continues to the east here, but the way is blocked by
a purple-velvet rope stretched across the hall. There are doorways leading to
the north and south.
You see a sign hanging from the middle of the rope here.
>read sign
This point marks the end of the currently-occupied portion of the house. Guests
proceed beyond this point at their own risk.
--- The residents
>go east
You step disdainfully over the velvet rope and enter the dusty darkness of the
unused portion of the house.
```

Fig. 2. Sample transcript of a session with a MUD.

topic, and such MUDs serve as sites for virtual collocation and collaboration.

The difference of opinion on MUDs suggests a dual approach to our study of MUDs: first, in terms of details particular to MUDs, and second, in terms of virtual environments in general.

3.1 LambdaMOO

The original MUD, entitled "Multi-User Dungeon" (and originally abbreviated as *MUD*, but referred to as *MUD1* in the literature) was created in 1978. The immediate popularity accorded to MUD1 led to the creation of numerous MUDs derived from MUD1. Many of these MUDs were essentially similar, both in features and in spirit, to MUD1. However, one derivative system, *TinyMUD*, shifted the focus from gaming to the building of the virtual world and to the social interaction within that world. TinyMUD's constructive focus thus spawned a new breed of MUDs, including an object-oriented version called *TinyMOO*, where "MOO" stands for "MUD, Object Oriented."

In October, 1990, Curtis designed a new system, called *LambdaMOO*, based on TinyMOO. LambdaMOO provided a fully object-oriented language tightly coupled with an object-oriented database. The term "LambdaMOO" refers both to the basic MOO system and to its original operation as a public virtual environment at Xerox PARC. Here we generally use the term to refer to the system.

3.1.1 *Architecture.* A LambdaMOO system consists of two parts: the server program and the database. The server provides the low-level functionality for the system. In particular, it provides the MOO code interpreter and database engine. However, much of the functionality actually used by the LambdaMOO is encapsulated within objects in the database. As a result, a LambdaMOO is seldom started from scratch. Usually, the MOO administrator ("archwizard") downloads a database of core objects as the starting point for the MOO.

The most common core database is the database distributed along with the LambdaMOO system itself, called *LambdaCore*. The objects in this database, totaling about 100 to 150, comprise some basic MUD-related

objects (e.g., player, room, and exit) along with some utility functions (e.g., string and list manipulation, coding and network utilities). The core database is built by the maintainer of the LambdaMOO at Xerox PARC, who periodically extracts the core objects from that LambdaMOO's database, builds LambdaCore so that it contains only those objects, and then offers this database for downloading.

This server/database architecture greatly aids the implementation of systems within LambdaMOO. Instead of modifying the server's code to add new features, the developer may simply add or modify database objects and program them in the MOO's object-oriented programming language.

3.1.2 *Object Structure.* Objects in LambdaMOO are identified by their object identifiers and are characterized by their *attributes*, *properties*, and *verbs* (akin to methods in more conventional object-oriented systems) [Curtis 1996]. The attributes on each object consist of a flag designating the object as a player, the object ID of the object's parent, and the object IDs of its children. Each object has exactly one parent and may have any number of children. This parent/child relationship forms a hierarchy that represents a combination of typical object-oriented notions of inheritance and instantiation. Each object has eight built-in properties that govern naming, location, and permissions. In addition, the set of verbs and properties on any object may be extended indefinitely.

The LambdaMOO object model is sufficiently rich to support the modeling of any type of object the process engineer may need. However, it is perhaps more useful to understand what types of objects are commonly encountered in the course of using the system. These common objects include the following.

—*Players:* Every user of the system is represented by a player.

—*Rooms and exits:* A room is a space designed to contain players, and an exit is a one-way connection between rooms. While these are very simple objects, they are extremely important in forming the basis of the spatial metaphor in LambdaMOO. On a technical level, these rooms and exits in LambdaMOO form a directed graph that the players navigate. More importantly, however, nearly all action that takes place in a LambdaMOO centers around some room, so navigating among the rooms is of paramount importance. Rooms and exits also provide some hooks for access control, in that a room or exit may refuse to accept an object.

—*Bots:* Since the LambdaMOO server is capable of performing actions independent of players, various kinds of automated agents—including robots, animals, and the like—are often found in LambdaMOO. Some bots, like the so-called "Housekeeper," perform useful actions within the MOO. Others are just for testing out ideas and for entertainment.

—*Utility objects:* Not usually seen by ordinary LambdaMOO users, utility objects generally encapsulate some set of functionality available to programmers. For example, the Code utilities object contains a number of

functions that help the system and the programmers maintain MOO code.

Note that the semantics of these objects resides primarily in convention; often LambdaMOO does not enforce the meaning of a given object very strictly. For example, one could pick up a room and carry it to remote parts of the world without disrupting the people inside!

3.1.3 *Containment.* Two built-in properties on every LambdaMOO object are **.location** and **.contents**, which together form a containment hierarchy of all objects within the MOO. That is, each object has another object as its location and zero or more objects as its contents. LambdaMOO maintains these properties carefully across all objects to ensure that they remain consistent and acyclic: i.e., an object's location must contain the object, and no object may contain itself, either directly or indirectly.

It is interesting to note that LambdaMOO does not necessarily assign one specific meaning to the **.location** and **.contents** properties. If an object's **.location** is a room, then we may say that the room contains the object; if **.location** is a player, we may say that the player is holding the object; if **.location** is the so-called "recycler" object, then the object is a "dummy" object waiting to be re-created. The fact that containment, possession, and object status differ from one another semantically does not cause any difficulty with respect to the use of these properties.

More details on objects and their properties and verbs may be found elsewhere [Curtis 1996]. In this article, we supply only those details about objects that are important to our discussion. Examples of software process-related objects are given in Section 5.

3.2 Virtual Environment Research

Our interest in MUDs focuses on their place in the more general field of virtual environments. While some researchers [Curtis et al. 1995] have used the term *virtual reality* to describe systems such as MUDs, this term typically refers to systems that attempt to represent the physical world accurately (e.g., by depicting three dimensions). In MUDs this is often not the case: the absence of a realistic spatial metaphor, the ability to perform tasks that are impossible in the physical world (e.g., teleporting objects), and the MUD-specific communicative and social forms [Carlstrom 1992] all point to a major difference between MUDs and reality. Accordingly, we use the more generic term *virtual environment* to refer to any system that presents a (possibly unrealistic) world or space for users to visit and inhabit.

Bruckman [Bruckman 1994; Bruckman and Resnick 1995] has contributed a great deal to the field of virtual environment research, particularly in the use of MOOs for educational purposes. In particular, Bruckman and Resnick [1995] argue that the construction and reconstruction of the virtual world leads to a heightened effectiveness in collaborative learning and interaction. Based on their experience in running MediaMOO, they

believe in the effectiveness of the constructive aspect of the virtual in encouraging interaction within a professional community.

Kaplan's work on the use of virtual environments to accomplish work in the real world makes some useful distinctions [Fitzpatrick et al. 1996]. He points out the existence of a mapping between the virtual world and the real—in his terms, between the "sites and means" (spaces and methods) of the virtual and the "social world" (real-world domain). He defines the term *locale* to designate some portion of the social world that corresponds to a given element of the virtual, i.e., a locale is an abstraction of the virtual space in terms of the semantics of the real-world domain.

One commercial system that uses a virtual environment metaphor to facilitate real-world work is Taligent's Places for Project Teams. The product uses spatial concepts—in particular, the concept of a room—to model cooperative work on a common project. The room is used as a gathering space for all those working on a project, and it is used to store documents common to the project. This virtual collocation permits what Taligent terms "serendipitous communication": the ability to communicate with those who are in the same room (and are therefore working on the same project).

4. CAPTURING SOFTWARE PROCESSES WITHIN VIRTUAL ENVIRONMENTS

Given the potential utility of virtual environments as systems for accomplishing work in the real world, this section explores how such environments might capture software processes. We begin by discussing the motivation for such an approach, continue by describing several metaphors for process execution within a virtual environment, and conclude by discussing some other key issues in capturing the software process in virtual environments.

4.1 Motivation: Finding a Metaphor for Process

As pointed out above, the primary feature of a virtual environment that makes it attractive for accomplishing work is its mapping of the real-world domain into the virtual. Any system that exploits such an environment, then, must define this mapping with respect to the domain of interest. We refer to this mapping as the *metaphor* of the environment.

In Section 2 we describe some requirements for what a process-centered environment must provide to its users. Chief among these requirements is the need to represent within the environment the entities of the software process and the connections among those entities. In the case of a virtual environment, this representation must be defined in terms of the environment's metaphor.

We must be extremely careful, however, in defining this metaphor. If the virtual environment is to provide a useful abstraction of the software process, then it needs to satisfy certain requirements:

—*Correspondence:* The environment should represent each software process entity in a manner that preserves the properties of that entity. For example, the environment's metaphor should not unduly restrict access to artifacts.

—*Collaboration:* The environment ought to provide facilities for collaboration among the people executing the process. Since one of the more appealing aspects of virtual environments is virtual collocation—the ability to be in the same "space" with another person in the virtual world despite physical separation from that person in reality—a VE-based process execution system should use virtual collocation to facilitate collaboration.

—*Realism:* While a virtual world need not be identical to the real (i.e., physical) world, it should represent a limited extension of concepts in the physical world. For example, while moving objects from room to room is generally quite natural within environments, picking up rooms and moving them through other rooms would tend to disrupt the metaphor (and makes the users of the environment a little confused).

There may be times when these criteria represent a trade-off. In some cases, for example, one may achieve greater realism at the cost of diminished correspondence, such as by not allowing a room to be picked up and moved to another room even though this might otherwise correspond to a reasonable activity in the process. Furthermore, it may be acceptable or even preferable for a metaphor not to correspond exactly to software processes as we understand them. Such a metaphor may lead to new insights into process because of its unique perspective.

4.2 Metaphors

While the set of potential mappings between the virtual and the real worlds is nearly limitless, we wish to choose one that respects the features of both domains as much as possible. We proceed, therefore, by selecting the most prominent features of both domains and attempting to map them to one another.

Since we have chosen LambdaMOO as our virtual environment framework, we have also chosen the concepts of rooms, exits, and players as the most prominent aspects of our environment. Since mapping human beings onto players is such an obvious approach, we presume that this is a part of any metaphor, so we examine different metaphors by mapping the other process entities onto the room structure. We then attempt to evaluate each metaphor with respect to the criteria mentioned above.

In performing our evaluation of metaphors, we examine how the metaphor might represent a simplistic testing activity within the MOO. The testing process entails the following two main activities:

(1) The tester (a person) obtains sole use of a testing laboratory.

(2) The tester performs specified tests on a program using the facilities of the testing laboratory.

To be sure, in order for a metaphor to facilitate the execution of this process, it must be able to represent all the entities of the process: the tester (an agent), the testing laboratory (a resource), the test specification and results (a complex of artifacts), and the program under test (a product). Furthermore, the metaphor should represent the activities themselves so as to make process execution possible.

4.2.1 *Task-Centered Metaphor.* A first metaphor to consider is that in which each task (i.e., activity) corresponds to a room within the MOO. Since we define a process as a sequence of activities, a mapping of activity onto room suggests a mapping of activity sequences onto the room layout using exits. Such a system could represent actions in one of several ways, e.g., by reifying tools and using them to effect the action or by using the MOO's verbs to invoke actions.

Using the task-centered metaphor for our testing activity could be extremely simple: a single room would correspond to the testing activity itself; the executable, test input, and desired output files would be objects within the room; the human performing the testing would enter the room to engage in the activity; and either the person or the executable would not be permitted to leave the room until the testing had been performed (and perhaps until a report had been generated). To be sure, a process modeler could be more specific than this in describing the activities; instead of a single room, the player could, for example, carry the artifacts through rooms representing the testing and report-generating subtasks.

This metaphor corresponds well to more conventional notions of process activities, especially those that use process modeling languages based on state machines or Petri nets. The state of a process is modeled by the state of the artifacts together with the rooms within which those artifacts currently reside. The metaphor also maps well to a fairly natural human concept of physical motion as progress. One can imagine, for example, the MOO artifact objects as the manifestation of tokens in a Petri net and the player being the force that pushes them through transitions. In terms of collaboration, the metaphor supports people working on the same task occupying the same room together. Those performing distinct tasks, however, do not enjoy the collocation that the MOO offers.

One respect in which the task-centered metaphor, as stated thus far, fails to correspond to the real world is, interestingly enough, in the mapping of person to player. Since people often work on several different tasks "simultaneously"—i.e., they alternate among these tasks—the MOO using this metaphor needs to provide a means for a person to leave one activity and return to another activity, restoring the original activity's context as appropriate. LambdaMOO itself does not provide this capability. Therefore, in Promo, we have had to add a facility to support the interleaving of activities by players. We refer to this problem as one of "multithreading" to suggest the similarity between the interleaving of process activities and the manner in which operating systems alternate among threads of control within a concurrent program. The multithreading difficulty is not unique to

the task-centered metaphor; in fact it is shared by nearly all of the other metaphors.

Since the task-centered metaphor forms the basis for Promo, we examine this metaphor in greater detail in Section 5.

4.2.2 *Agent-Centered Metaphor: Workspaces.* In the agent-centered metaphor we map a person's *workspace* to a room. A workspace acts as a synthesizer of multiple activities to be carried out by the same human agent. The MOO player representing a person performs only one task at a time, but the person's many "threads" are unified in that all the person's activities are carried out in one room. In our testing example, the artifacts are again objects in the MOO. In this case, the player brings them into their personal workspace in order to carry out the testing activity.

One clear advantage to this metaphor is its realism: the notion of workspace is quite familiar to anyone who has ever had an office. The correspondence of this metaphor with notions of process, however, is unclear because no mapping between the remaining process entities and the MOO is obvious. Furthermore, such a metaphor would have to elucidate how collaboration would occur; while a player could certainly visit someone else's workspace, the process system ought to specify more closely how players—and, perhaps more importantly, artifacts—should travel between workspaces. Finally, how such a metaphor should represent an activity is not clear, and without an explicit representation of activities the execution of a process is much more difficult.

4.2.3 *Artifact-Centered Metaphor.* In an artifact-centered metaphor, the artifacts are mapped onto rooms and the agents onto players. While this could be a viable metaphor—for example, exits could be used to represent relationships among artifacts (i.e., rooms)—it immediately seems a bit cumbersome in that the artifacts would then be stationary. The ability to combine artifacts, at least spatially, would be lost, and since processes often involve manipulating and combining artifacts, this is a point where the metaphor severely fails to correspond to reality. The same criticism applies to a tool-centered metaphor, since we may easily consider a tool to be an artifact of a process.

Work done by Masinter and Ostrom [1993] in integrating Gopher into a MOO seems to bear out our hypothesis about stationary artifacts. Their first attempt at providing access to Gopher within a MOO was a "Gopher room" that acted much like a traditional Gopher client. However, they discovered that the need to travel to the room both for the tool and for the data made the tool very cumbersome, and as a result they opted for metaphors that provided for greater portability of tools and data (e.g., a portable "Gopher slate").

4.2.4 *Resource-Centered Metaphor.* In a resource-centered metaphor, the MOO's rooms would correspond to resource instances within the process. Being in a resource's room would then indicate possession of an instance of that resource. For example, a player's presence in a "testing

laboratory" room would represent that the player had the use of that laboratory, and a player's presence in a room corresponding to a tool (particularly a tool with finitely many licenses) would represent the player's possession of one license. Furthermore, the room (and its exits) would enforce the appropriate sharing policy for that resource. For example, a meeting room could be used for only one meeting at a time, whereas a room for a given tool might permit several occupants (users of the tool) at the same time.

In the testing example, a room could correspond to the laboratory used to test the product. Various restrictions could be placed on this room and, thus, on the laboratory. For example, only one person (or group of people) may use the room for one purpose at a time.

In a sense, the resource-centered metaphor corresponds to one common use of rooms in the physical world. Since a real room consists of a finite amount of space, the room itself is a resource, as in the case of a meeting room or office. Furthermore, a real room serves as a convenient means of organizing other resources, such as electricity, network wiring, climate control, and the like.

However, the mapping of MOO room to resource fails to preserve the resource-oriented aspects of real rooms. First, the physical limitations imposed by the real world are not present in the MOO; a room in a MOO may be virtually infinite in size, and rooms may be created and destroyed quite readily. Consequently, associating real rooms with MOO rooms becomes unnecessary, since these constraints of the physical world do not apply within the MOO.

Second, and perhaps more importantly, assigning a MOO room to each resource fails to scale when multiple resources must be secured. In particular, a severe scaling problem arises whenever the MOO must ensure exclusive access to more than one finite resource. For example, suppose that our testing activity requires not only sole use of one of m laboratories, but also sole use of a testing tool with n licenses. It would be convenient to allow the player to be in a laboratory room *and* a tool room in order to represent the separate resource instances being held simultaneously. However, multilocation is not possible within a MOO—the player may be in only one room at a time—so the room must represent combinations of resources held by the player. Hence, in our example, the MOO must use mn rooms to represent the many laboratory-license combinations. This multiplication of rooms for additional resources becomes unacceptable when several resources must be secured.

4.2.5 *Product-Centered Metaphor.* A product-centered metaphor would map each separate product onto a different room. If subproducts were defined, then they could be represented as separate rooms connected to the parent product room either by exits or by containment. As before, each agent would be represented as a player and each artifact as an object.

In this metaphor, the testing example would be represented by means of a room corresponding to the executable along with objects representing the

test input and output. A player would execute the process by going to the room corresponding to the executable and invoking the appropriate actions there.

This metaphor would certainly facilitate a great deal of collaboration among people working on the same project, although those working on corresponding tasks in separate projects would not be collocated and would not have such collaboration facilitated. While the metaphor supports a view of software process based on product structure, most process systems model processes based on activities and not on products; hence the metaphor does not fare well with respect to correspondence. In addition, this metaphor has the same problem as the task-centered metaphor in its need to support a person's ability to switch among threads.

4.3 Hybrid Metaphors

Having defined the semantics of several different mappings of process onto virtual environments, we note that in the real world people do not require strict semantics in order to operate within some metaphor. Spaces (buildings) in the real world may be combinations of distinct types of rooms, including task-related rooms (meeting rooms), workspaces (offices), and artifact-centered rooms (workshops).

Nevertheless, the task of combining different aspects of these metaphors into a single metaphor is far from simple. Indeed, many systems that support cooperative work allow users to define the metaphor during execution by designating the meanings of rooms, spaces, etc., as they desire. However, while people may not have difficulty understanding a composite metaphor, our focus on the execution of processes makes the well-defined metaphor a crucial piece of our system. A system cannot use its virtual environment to support process execution without a well-defined metaphor. For example, constraint checking cannot be integrated into the virtual environment if the virtual environment cannot represent such concepts as task completion, artifact state, and the constraints themselves in a consistent fashion. An example of this shortfall can be found in Taligent's Places for Project Teams, mentioned in Section 3.2. In this product, the meaning of a room is defined by the user, and as a result, the room-based metaphor does not support automated process execution at a system level.

One possible avenue of further research concerns the unification of metaphors, i.e., the ability to provide different metaphors for the same process and the same database, in much the same way that DBMSs provide multiple views on the same data. Such a system would use the metaphor as a conceptual "filter" to allow users to understand the interactions among tools, artifacts, etc., within the virtual environment, so metaphors could be considered different views on the same organization of data. As with the hybridization of metaphors, however, we must tread carefully on this ground; the metaphors are far from identical. Given the primacy of rooms within virtual environments, our choice of process entity to be mapped to a room indicates our belief about the importance of that entity within the

process. For example, using a task-centered environment implies that tasks are central to our understanding of process. Furthermore, a particular mapping defined by a metaphor implicitly attaches the properties of MOO entities to process entities. For example, the task-centered metaphor suggests that one person may be working on only one task at a time because a player may be in only one room at a time.

4.4 Other Issues

Once a means of representing a process within a virtual environment is defined, the environment must provide the user with an interface that permits the user to interact with processes. While the interface issue encompasses the narrower issue of user interface, it also encompasses a range of issues regarding how the user/process interaction should take place within the metaphor of the virtual environment.

Some features the interface should provide include the following:

—*Process state:* The environment should be able to represent and answer queries about the state of a process.

—*Process training:* The environment should convey information about how to execute a process to users, particularly to users unfamiliar with the process.

—*Process history:* The environment should permit the querying of previous actions within the current process.

—*User interface:* The user should be able to interact with the environment in a convenient way.

—*Tool interface:* The environment should provide simple access to any tools needed to execute the process.

A second question raised by the fact of supporting process execution concerns the degree of automation of such a system. While many actions within a process must be carried out by a human agent, others may often be carried out in automatic fashion. A system that supports this degree of automation is considered *proactive*. Various techniques exist for implementing this automation, including rule bases [Heineman et al. 1992] and events and triggers [Bandinelli et al. 1994].

Systems that provide little or no proactivity may afford the user a great deal of flexibility in executing the process, but it is interesting to note that this flexibility may be a liability in many cases. A user whose actions are not guided by the system may be at a loss as to how to make progress within the process.

A virtual environment could be used as the basis of either a proactive or a nonproactive system. In a nonproactive system, the virtual environment would simply assume that the user was capable of carrying out the appropriate activities within the process, perhaps aided by an operation that explained to the user what was to be done in the room. On the other hand, a proactive VE-based system could represent its proactivity in a very concrete way by incorporating this proactivity into its metaphor. For

example, robots could be used to represent that part of the system concerned with taking actions on behalf of the user; the robot could then invoke actions, carry artifacts from room to room, and so forth.

5. IMPLEMENTING THE TASK-CENTERED METAPHOR

In an effort to illustrate the concepts discussed in the previous section, we have developed Promo, a prototype proscriptive process-centered environment based on the LambdaMOO system. The metaphor used in Promo is the task-centered metaphor described in Section 4.2.1. This metaphor was chosen simply because it is consistent with the structure underlying most existing process-centered environments, and therefore provides a good first-example application of virtual environment technology. It is conceivable to develop prototypes based on the other metaphors, which would be a fruitful area of future work.

This section discusses the design decisions made in creating Promo in light of the metaphor discussions of the previous section. In doing so, we evaluate the specifics of the metaphor using the criteria set out in Section 4.1. A discussion of several lower-level implementation issues of Promo appear in Section 6.

5.1 Capturing the Metaphor

Promo captures the task-centered metaphor in LambdaMOO through a particular mapping of process entities to MOO representations. As the discussion reveals, the role of process modeler is, as in all process-centered environments, a key role. Our discussion below highlights the likely effects of different modeling choices.

5.1.1 *Artifact: Object.* Each artifact within the process is represented by a single object within Promo. That is, while the data of the artifact are assumed to exist outside the MOO (presumably in the file system), every artifact to be manipulated within the process is represented by a corresponding object within Promo. The Promo object contains a URL [Berners-Lee et al. 1994] that acts as a pointer to the actual artifact. Like Marvel's use of the file system as its database [Kaiser et al. 1988], the use of URLs within Promo permits the easy storage and manipulation of potentially complex data. By contrast with Marvel, using URLs rather than the local file system permits the distribution of artifacts in a manner that makes sense for the process.[1] While the data of the artifact reside in the network resources specified by the URL, some of this information may also be replicated in the Promo object; a similar approach is also taken in Marvel.

The artifact-to-object mapping provides us immediately with an instance of the modeling granularity issue. On one hand, if the Promo object contains a great deal of information about the actual artifact, keeping the artifact and its corresponding object in sync becomes a difficult task. Any tool that modifies the artifact must propagate all information about the

[1] Of course, Marvel predated the concept of URLs.

```
Object ID: #162
Name:     C source module
Parent:   Source module (#117)
Properties:
     description: "The generic C source code module."
     url: ""
     mime_major: "text"
     mime_minor: "c-source"
     modreq:
     object: #-1
```

Fig. 3. Definition of a generic C language source-code object.

changes to the artifact back into Promo. (Section 6 provides greater detail about the mechanism by which this propagation is accomplished.) On the other hand, if an object contains too little information about the artifact, the state of the artifact—and thus the state of the process—becomes difficult or impossible to track within Promo. This is an issue faced by any process-centered environment that represents actual artifacts by objects [Feiler and Kaiser 1987].

Promo can, of course, represent information about an artifact that might not be contained within the artifact's file(s). In particular, properties of the Promo object can represent interobject relationships and artifact state. For example, a source module may contain a "pointer" to (i.e., the unique identifier of) its related object module, and an object module may contain pointers to the executables that use it. In addition, an object may contain a property indicating its completion status as the result of a manager's approval.

The "type" of an artifact is generally represented within Promo by exploiting the parent/child relationships inherent in the LambdaMOO database. Every object in LambdaMOO has exactly one parent and zero or more children. This parent/child relationship represents a combination of the ideas of inheritance and instantiation. For example, the modeler may create a generic source-code module object, create a generic C language source-code module object as its child, and then create specific source-code module objects as children of the generic C language source-code module object. Indeed, one of the core Promo objects is a generic artifact object called **$artifact**, and all artifact objects are expected to be descendants of **$artifact**.

Figure 3 shows the LambdaMOO definition of a generic C language source-code module object. The **.url** property holds the URL of the artifact, whose value would be set only within the instances of this generic object. The type of the artifact, expressed here in MIME-like notation [Borenstein and Freed 1992], is given by the properties **.mime_major** and **.mime_minor**. Property **.modreq** contains a list of the modification requests that refer to the source-code object. Finally, **.object** contains the object identifier of the Promo object that corresponds to the compiled code for the module (i.e., the "**.o**" file for this "**.c**" file).

While this mapping of artifact to object is a fairly natural one, it does pose some problems with respect to our criteria for evaluating a metaphor. The single-containment property mentioned in Section 3 raises an interest-

ing question about artifacts: how does the MOO represent the concept of having access to an artifact? The requirement that a player pick up an artifact's object before using the artifact does not correspond to the typical process assumption that many people may read an artifact simultaneously. On the other hand, a multiple-access paradigm, while preserving correspondence with typical process concepts, would be difficult to model within the MOO without causing surprising actions-at-a-distance and thereby violating our realism criterion. Essentially, this problem introduces the concept of pointers not only into Promo's implementation but also into the user's view of Promo, whereas the physical world does not contain pointers in this sense.

Although this problem is as yet unsolved in Promo, one possible solution would entail introducing a concept of "virtual presence," i.e., a player would be able to have (and pick up, hand to other players, etc.) an ethereal copy of an object without having the actual object. Such a solution would preserve the feel of the MOO as a conceivable (if slightly implausible) extension of the physical world, satisfy the technical constraints of implementing software process, and still maintain the consistency of the metaphor.

5.1.2 *Human Agent: Player.* As suggested by the description of artifacts above, the human agents in the process are represented as players in Promo. As mentioned in Section 4.2.1, this poses a problem in that it fails to account for the multiple "threads" of activity that a person carries out. While some systems use the notion of roles to solve this issue, we eschew this solution, since the notion of role implied by such an approach is a conflation of too many issues [Heimbigner and Osterweil 1994].

Instead, we identify each of the person's threads as a *persona*: a persona corresponds to a person within a certain process at a certain stage (i.e., working on a certain task) with respect to one or more artifacts. In this way, we may continue to identify a person with a LambdaMOO player; the person may change threads by simply switching personae. The fact that we define a persona as being at a certain stage *with respect to artifacts* is crucial in that it permits us to define the process in terms of the artifacts' state and not in terms of the human agents. In moving away from the notion of roles, we have effectively freed the users to act independently in various contexts (i.e., in various processes) without constraint.

The player object in LambdaMOO was augmented within Promo to yield a new generic object, called the generic developer object, as shown in Figure 4. Every user of Promo is intended to be a child of this generic developer object. The properties and verbs on this object all govern the creation, selection, and deletion of personae. Property **.persona** holds the name of the current persona, and **.personae** holds the information corresponding to each persona (indexed by name); the default values for these properties are indicated as null. The verb **:@addpersona** creates a new persona corresponding to the current activity, and **:@rmpersona** deletes a persona. Finally, **:@persona** permits switching between personae (or displays the name of the current persona).

```
Object ID: #130
Name:     Generic Developer
Parent:   generic programmer (#57)
Verb definitions:
      @addpersona
      @persona
      @rmpersona
Property definitions:
      personae
      persona
Properties:
      personae: {}
      persona: ""
```

Fig. 4. Definition of a generic developer object.

5.1.3 *Action: Verb.* A given action (or operation) within a process is modeled within Promo as a verb. For example, a player may edit an artifact by issuing the command

edit ⟨**object-name**⟩

where ⟨**object-name**⟩ is the name of the object corresponding to that artifact. These verbs are designed so that they may be invoked either by a player or by Promo itself (i.e., by a robot).

The similarity between the semantics of process actions and those of MOO verb calls suggests this mapping. A process action is intended to be a single, atomic operation on the artifact, and a request for the action should result in the immediate invocation of the action. However, an action may take an arbitrarily long time to complete. Similarly, within the MOO, a single verb call is also intended to take place instantaneously, but its long-term effects may not be felt until some time later, particularly if the verb involves interaction with another player or with a robot.

Since the player actually corresponds to a networked user who may be geographically distant from the MOO server, the invocation of an action must be carefully coordinated with the user's client. We have chosen an architecture for action invocation that relies on a "smart client" on the user's side. When an action is invoked, the MOO issues to the client a command to invoke the action along with a set of information about the action. In this way, we push the decision about which tool to execute onto the client, ensuring that the appropriate platform-specific tool will be invoked for the action. The details of action invocation are discussed further in Section 6.

An alternate approach to action invocation would entail the reification of the tool itself. That is, instead of issuing a verb to perform the action, the player could "hit" the artifact (or otherwise operate on it) with the tool, and this would invoke the specified action. While we have not implemented this approach within Promo, it is conceivable that it would be desirable, as shown by experience with GopherMOO [Masinter and Ostrom 1993]. In particular, tool reification could provide the ability to subclass tools. The process modeler could design, for example, a generic editor tool, and then subclass that with platform- and artifact-type-specific editors.

5.1.4 *Task: Room.* The centerpiece of the metaphor used in Promo is the identification of tasks with rooms. Since a task is a semantically meaningful set of actions, we must specify how to define tasks and how to manage task instances, such as when several people are simultaneously executing a process. Within Promo, a task is defined by a *constraint*, a set of conditions that must hold before the task is to be considered complete. The transitions between tasks are represented within the metaphor as exits. Each exit has associated with it a constraint that, if violated with respect to some object, will not allow the passage of that object through the exit.

The constraint language developed for use with Promo is quite generic and allows for many different types of constraints to be written. However, constraints are intended to apply not to the players but to the objects being moved through the process. The freedom of movement that results from this use of constraints is important so that players may switch among multiple personae and walk through rooms to query the state of a process.

Since the state of the process—the set of locations of artifacts pertinent to the given process—is controlled only by the exit constraints, Promo is a *proscriptive* environment for process execution [Heimbigner 1990]. That is, Promo controls the process only by ensuring that invalid process states do not occur, rather than by guiding the process in a specific direction.

In support of the concept of subtasks, Promo provides the ability to create a "subbuilding," i.e., a building within a room. The subbuilding may have as many rooms within it as are desired, and it may have as many doors as are desired. It thus enables passage through a sequence of rooms, with their own exits and constraints, while staying within a larger room.

The subbuilding model provides two main advantages. First, it permits the representation of subtasks while maintaining the containment of objects. That is, if an object is contained within a room corresponding to a subtask, it is indirectly contained within the room corresponding to the parent task. Hence we may ask questions about a running process or about the state of an artifact simply by phrasing the question as one of containment. For example, we could say that an artifact was in the *Update* subtask of an enclosing *Implement* task. Second, subbuildings are a means of implementing transactions. For example, the first room in a subbuilding could have an exit constraint that a transaction has begun, and the subbuilding's exit door could require the successful termination of that transaction. The main disadvantage of the subbuilding model is that it does not correspond to a physical phenomenon; enclosed spaces within larger rooms are not commonly found in reality.

As discussed in Sections 4.2.1 and 5.1.2, Promo supports a notion of "threading" that is concretely represented by the concept of personae. Personae contribute to a mechanism for representing a process instance. The movement of the persona indicates the progress of a given process. Personae can also handle highly automated processes, since they can be attached to robots as well as people. Additionally, some parts of the process instance are mapped to objects. Thus, rather than creating a new persona

for each test case, for example, test cases are represented as objects, and their room location and associated attributes provide information about the object's status vis-a-vis the process.

5.1.5 *Resources.* The metaphor used in Promo does not explicitly represent resources themselves. However, Promo can support standard operations on resources (e.g., acquisition and release) by representing these operations as activities within a process. For example, a room corresponding to a testing activity may be preceded by a room that corresponds to the acquisition of a test laboratory, and the exit constraints may be designed so that only one testing activity may be carried out in the testing room.

5.2 Atomicity and Transaction Semantics

Promo, since it is based on LambdaMOO, only allows an object to be in one place at a time and to be picked up by one player at a time. Thus, Promo naturally supports a restricted form of atomic action: only one player can be performing an action on an object at any given time. But there are many relationships among objects, and ideally one would like to include many objects within the boundaries of an atomic transaction. As mentioned above, subbuildings could be used to designate transaction boundaries, so one can achieve the desired effect by bringing all relevant objects inside such a subbuilding.

But Promo's ability to support a general transaction semantics is limited. In particular, the actual artifacts are stored outside of Promo and thus may be modified independently of Promo's actions, without regard to any notion of transaction that might be present inside Promo. On the other hand, if the objects referenced by Promo were actually stored within a database system supporting transactions, then Promo could invoke operations on that database to implement the required transactions. Thus, traversing an exit into a subbuilding might invoke a begin-transaction operation, while exiting that subbuilding might invoke an end-transaction operation.

Handling transaction abort is a bit more difficult. The simple abort action can be handled by providing a special abort exit. But abort also implies rollback, so there must be some mechanism for Promo to modify its state (including contained objects) to be consistent with the aborted state in the database. At the very least, Promo could track all objects modified within the subbuilding. On abort, it would then rebuild the state of each modified object from the database state.

5.3 Interface

While the semantics of the process-to-MOO mapping are important, the interface by which users and tools interact with Promo is equally important. Below we discuss how Promo addresses the interface issues previously mentioned.

5.3.1 *Process State.* Because of the task-centered model that Promo uses, the state of a given artifact can always be determined by simply

examining the artifact object's location. Although this examination may sometimes require traversing the containment tree in the case of subtasks, it provides a convenient way of tracking artifact state.

Since no object exists within Promo that corresponds to an entire process, however, querying the state of an entire process is not easily supported within Promo. Such a facility would be fairly straightforward to add by simply exploiting the location of artifacts and the relationships among artifacts. Currently, a player can "walk" through the set of rooms corresponding to the process and examine the objects found therein to determine process state. This could be automated by having a robot perform the walk instead.

5.3.2 *Process Training.* Trying to understand an unfamiliar process can be a formidable task, particularly in a proscriptive environment wherein finding out which actions are not permitted is easier than finding out which actions are possible. Since MOOs are text based and typically foster a great facility with textual description, Promo places the onus of describing the process on the shoulders of the process modeler.

Nevertheless, the structure of rooms and exits in Promo does provide a convenient framework in which to place process documentation. Since each room corresponds to a task that presumably has some meaning, the **.description** property for the room may be set to a string describing that meaning. Furthermore, rooms within Promo are designed so that examining a room (using the LambdaMOO verb **look**) lists all its exits, which may also have documentation strings attached to them. Hence, simply by examining a room, the player can find out what the room is intended for as well as which conditions are necessary to exit in any of several ways.

A further facility for documenting processes is provided in the constraint creation facility. A parameterized string may be attached to any constraint that the user will see should the constraint fail. Hence the constraints may be tailored in such a way that they tell the user why the constraint failed—in terms of the real-world domain and referring to specific objects, personae, and rooms—and perhaps indicate some action or set of actions that are needed to cause the successful evaluation of the constraint.

5.3.3 *Process History.* The facility that Promo provides for querying process history, like that for querying process state, resides with each artifact. That is, one can list which actions have been taken on a given artifact. While this may not provide enough information to the user about the history of the collective process, the problem of providing information about process history is one found in most process systems [Kaplan et al. 1997].

5.3.4 *Tool Interface.* Promo's interaction with tools occurs through the "smart client" remote action invocation. While this is a convenient model for tool invocation, it presumes that all the desired tools will be invoked from within Promo. However, this need not be the case. For example, an

artifact may be edited or executed from outside the system, and Promo cannot currently track such outside actions.

5.3.5 *User Interface.* One common criticism of MUDs is their basis in textual, as opposed to graphical, communication. While text often leads to forms of communication that are of interest to sociologists and linguists [Carlstrom 1992; Cherny 1995a; 1995b; Curtis 1992; Reid 1994], this interface is considered by some to be only adequate at best in terms of usability [Curtis and Nichols 1993; Kaplan et al. 1997].

One of the new directions in MOO interface work has involved a move toward World Wide Web (WWW) interfaces. However, it is not clear that these interfaces typically offer major improvements over the older interfaces, since the data in MOOs are still largely textual, and hence the WWW interface provides hypertext links but few graphics. Furthermore, WWW technology does not permit these interfaces to be as interactive as their textual counterparts, and this has a severe negative impact on the collaborative character of MUDs. An interface recently added to BioMOO may prove an exception to this rule.[2] BioMOO combines a traditional textual interface with a WWW interface.

Promo currently uses the traditional textual interface, in large part because of its basis in LambdaMOO. However, while most MUDs rely almost entirely on the room designers to provide adequate descriptions of rooms, Promo automatically generates a fair amount of (textual) information to make navigation of the rooms easy for the user. For example, when the user enters a room, a description of the room is displayed, along with a list of exits. This exit list contains each exit's name, destination room, and any additional information provided by the process designer.

A more graphical interface would enhance interactions with the environment in at least the following ways:

—*Navigation:* Users could benefit from a map of the current room and of those rooms reachable from the current room, i.e., a picture of the process from the perspective of the current task. A processwide map would also be helpful for the user, but given the potential number of connections among rooms, it may not be possible to draw a simple (or even planar) graph of all the rooms in a process. A similar problem is encountered in graph-based process representations, such as Petri nets.

—*Action invocation:* A graphical interface could provide a straightforward means of invoking actions on an object. This would not only prevent the user from having to type a potentially complicated command, but could also ensure that the action requested by the user was a valid one (by providing a list of valid actions in a pop-up menu or list box, for example).

[2] http://bioinfo.weizmann.ac.il/BioMOO.

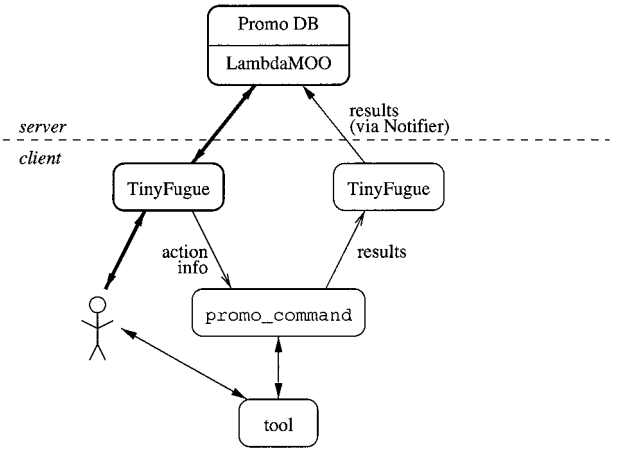

Fig. 5. Promo system architecture.

—*Administration:* Setting up rooms and exits in a MUD can be fairly tedious using a text-based interface. A graphical interface could greatly enhance the ease of creating and maintaining processes.

Without a graphical interface, interaction in Promo is much like that depicted in Figure 2, but specialized by the structure and constraints inherent in the process being modeled. An example of such a process is described in Section 7.

6. THE PROMO SYSTEM

This section discusses aspects of Promo as a system, including a description of the system architecture, discussions of some technical issues encountered in implementing Promo, and the methods by which those issues were resolved. While this discussion is by no means intended to be a complete account of the implementation, it does touch on the important issues that arose as part of the implementation effort.

6.1 System Architecture

Promo is designed to be a centralized environment for managing a distributed process. In our model, process modeling and execution must occur in a single location, namely within the LambdaMOO on which Promo runs. However, the process itself may be distributed, in that the location of artifacts and of action invocations that make up the process may occur on other machines. Promo's high-level design takes account of and supports this distribution of artifacts and actions. A diagram of the system architecture is given in Figure 5. The figure illustrates two important aspects of Promo's architecture.

First, the Promo server is simply the unmodified LambdaMOO server using a special database designed for Promo. Users interact with Promo much as they would interact with any LambdaMOO. The only difference

between Promo and other LambdaMOOs is the existence of special objects (and verbs on those objects) in the Promo database.

Second, while Promo tracks the process as it proceeds, it does not directly govern the actions that make up the process. Instead, each action that is invoked occurs on the client site. In the absence of action invocations, the user simply interacts with the textual client TinyFugue, which communicates with the server via the Telnet protocol [Postel and Reynolds 1983]. These elements of the architecture are shown using bold rectangles and edges. When an action is invoked, the server sends information regarding the action to be invoked to the client, which invokes the action and communicates the results of the action back to the server. Since these elements of the architecture are instantiated on-the-fly and are therefore more transitory, they are shown in the figure using thinner rectangles and edges. The process of invoking actions is described in detail in Section 6.4.

6.2 Artifacts

As mentioned above, each process artifact is represented within Promo by an object that has **$artifact** as an ancestor. However, this Promo object is not intended to store the artifact's data. Rather, it merely acts as the manifestation of the artifact within Promo. The artifact's data are stored in a file specifiable by a URL. Storing and referring to artifacts in this way serves two purposes.

First, it frees Promo and the tools it uses from the need to represent the artifact's data within LambdaMOO and to transfer the artifact's data in and out of the database. Since LambdaMOO is not designed to handle 8-bit ASCII, both the representation and the transfer of complex data would be extremely difficult. By keeping the data in a location specified by a URL, we rely on mechanisms designed to handle such data to perform any necessary transfers.

Second, it permits the system administrator to place artifacts at strategic points of the network in a manner that makes sense given the network configuration. Since Promo may be used in a wide-area context, the local availability of artifacts may be an important issue. Admittedly, Promo does not solve the problem of ensuring continued access to artifacts over a possibly unreliable network. It simply provides a mechanism for artifacts to be strategically placed over a network by the system designer.

6.3 Constraints

The exit constraints used in Promo represent a fairly powerful and complete method of guaranteeing task completion on exits. The constraint system is built as an extension of the existing LambdaMOO facilities for access control on exits. Whenever a player or object attempts to pass through an exit, the exit's verb **accept_object** is called with the player or object as an argument. Only if this verb returns a true value may the object pass through the exit. Within Promo, we have modified this access system to check exit constraints.

Each exit has a property, **.constraint**, that stores the constraint that must be satisfied in order for that exit to permit an object to pass through. The constraint is evaluated with respect to every object that attempts to pass through the exit and is recursively evaluated on all the contents of those objects. A false value of the constraint on any contained object will cause the main object's attempt through the exit to fail.

Constraints are typed expressions built from the following primitives:

—*Relational operators:* $<$, $\leq$, $>$, $\geq$, $=$, $\neq$
—*Arithmetic operators:* $+$, $-$, *, /, unary negation operator
—*Boolean operators:* and, or, xor, implies
—*Property retrieval:* Obtains the value of a property on an object.
—*Let expressions:* Permit binding of variables and subsequent reference to these variables.
—*Type checking:* isa-style ancestry checking.
—*Quantifiers:* forall, exists

In the current version of Promo, constraints are specified through calls to special functions that create an internal representation of those constraints. Since certain types of constraints take other constraints as arguments, the constraints can be built in a bottom-up fashion by starting with atoms (string, list, and integer literals), supplying these atoms to functions that build constraints, supplying the results of these functions to other functions, and so on. Future work on this aspect of Promo concerns the design of a constraint language along with a parser for this language so that this arcane method of specifying constraints can be avoided.

LambdaMOO supports the dynamic addition of constraints, although we do not currently make use of that facility. Instead, we have found that specifying class-specific and object-specific constraints has been adequate to capture the constraints of our examples.

Every constraint may have attached to it a string that the user will see should the constraint fail. Called the *rationale*, this parameterized string makes the failure of a transition constraint much easier for a user to understand, and liberal use of rationales is highly recommended to the process modeler. (Examples of constraints can be found in the appendix.)

6.4 Actions

Our decision about locating artifacts' data outside of Promo was largely motivated by considerations of the potential wide-area use of Promo and the difficulty of representing complex data within LambdaMOO. Similarly, we assume that the tools necessary to carry out actions within Promo will reside outside the system. Furthermore, information about available and appropriate tools for a given task may depend on the user's platform, which may not be known to Promo. Hence we must address the issue of how to invoke actions and obtain results from them. While invoking an action within the MOO is a fairly simple task—as easy as issuing any verb—it

begins a complex set of steps that actually cause the action to be carried out outside the MOO.

A number of MUD client programs exist that provide facilities for triggering actions based on strings sent by the MUD. By presuming that the client is "smart" in this way, Promo can simply print out a block of information about the desired action to the terminal. Currently, the only such client supported by Promo is TinyFugue, a line-based client designed for connecting to MUD servers.

Among the information sent to the client are the following:

—a unique identifier identifying this action;
—the object identifier and name of the object on which the action is being carried out;
—the object identifier and name of the player carrying out the action;
—the object identifier and name of the room in which the action was invoked;
—the type of the artifact, in MIME-like notation [Borenstein and Freed 1992]; and
—the name of the action as a string (e.g., **edit**).

Promo also supports arbitrary extensions to this block of information. In particular, a verb exists in the generic artifact object that can be overridden by artifact subtypes to allow the addition of arbitrary name-value pairs to this block.

Once the client has recognized that Promo is sending it a block of information for an action, it captures this entire block and gives it to **promo_command**, a script responsible for invoking the tool. The action's name and type, along with a mapping file, is used by **promo-command** to determine which tool should be invoked.

The script **promo_command** first parses the block of information received from the client and then consults its mapping file to find the Tcl procedure that encapsulates the desired tool. Creating these encapsulations is fairly straightforward. We have currently encapsulated the editors **vi** and **emacs**, the C compiler **gcc**, and a script used to run programs against input files. This method is quite similar to the Marvel concept of "envelopes" [Gisi and Kaiser 1991].

The encapsulation must provide information to **promo_command** to return to the MOO. In particular, it should return the following:

—A list of *events* that the action generated. The events are strings that should be understood by the artifacts within the MOO.
—A *comment*, for the user, describing the result of the action.
—Optionally, a set of *property-value pairs*. The properties of the object on which the action was invoked will be assigned the corresponding values from this list.

Once the action is complete, **promo_command** sends the results back to Promo. In fact, the mechanism for communicating these results is the same

client used by the user, TinyFugue. In particular, **promo_command** uses the client to log into the MOO as a pseudoplayer called the Notifier, issue special commands available only to this pseudoplayer that update the appropriate artifacts, and then disconnect.

The information reported back to Promo is then given to the artifacts. Each event is matched against the list of property specifications belonging to the artifact, and each specification permits the automatic setting of properties to certain values should an event occur. For example, an event **modify** could set the **checked** property to a false value, since the previous checking of the artifact may have been invalidated by the modifications made. Also, if an artifact has a verb **:⟨event⟩_trigger** for some event type ⟨**event**⟩, then this verb is called when the event is found. For example, a verb **:modify_trigger** on a source module could be used to notify the corresponding object module that it is out-of-date.

This method of communicating with the outside environment is limited in various ways. First, the method in which Promo communicates is fairly primitive, since it does not permit the passing of complex data into or out of the MOO. Second, and perhaps more importantly, Promo assumes that it has complete control over the artifacts and that no additional events will take place that were not begun within Promo. Both of these omissions make the modeling and execution of processes more difficult within the system, so Promo ought to provide better facilities for tool integration and communication. One possible model for these facilities is provided by the Oz process-centered environment [Valetto and Kaiser 1996], which permits interaction with more sophisticated tools, such as those that are continuously running.

7. EXAMPLE PROCESS

For an illustration of the use of Promo in modeling software processes, we have taken the anomaly report process of an industrial organization and implemented it using Promo. This process is described by Bandinelli et al. [1995], who modeled the process using the SLANG process modeling language. Following Bandinelli et al., we present only representative portions of the process.

Because we are implementing only portions of the process here, we do not intend this to be a full evaluation of Promo, either on its own or with respect to other process-centered environments. Rather, the example is intended to demonstrate that a reasonably complex process, previously captured in a rather sophisticated environment, can indeed be modeled in Promo.

7.1 Statement of the Process

The process in question concerns the generation and handling of anomaly reports (ARs) for a software system. The steps in this process are as follows:

(1) The configuration management group (SGMR),[3] which is in charge of the AR, checks the report for correctness of form (e.g., all the required fields must be filled in).
(2) The SGMR then summons the Change Control Board (CCB), which reviews the AR to determine whether it is a valid report. If the CCB decides that the report is not valid, it is rejected. If it is found to be valid, the CCB accepts it and generates modification requests (RIMOs[4]), i.e., requests for changes to be made to the product in order to fix the problem. The CCB may also suspend its investigation of an anomaly report.
(3) Development and qualification groups implement the requested modifications. They also perform the tests indicated by the CCB and issue a test launch report (TLR). The TLR and changed artifacts are placed under configuration control.
(4) A review team performs verification and validation (V&V) activities both on the artifacts modified and on the TLR. They issue a verification and validation report (VVR) based on their findings.
(5) The SGMR receives the VVR and determines the new status of the modification requests and anomaly report based on the VVR.

The SLANG process modeling language used by Bandinelli et al. results in specifications that consist of finite-state machines and ER (Environment/Relationship) nets [Ghezzi et al. 1991], which are extensions to Petri nets in which the tokens represent objects in an object-oriented database. ER nets also permit the abstraction of nets by providing an "interface" to a given set of places and transitions.

7.2 Modeling the Process

The three main portions of the process modeled by Bandinelli et al. are (1) the top-level items in the process, modeled as a finite-state machine; (2) the activity entitled *ImplementRIMO*, modeled as an ER net; and (3) the activity entitled *RequalificationOfUpdates*, also modeled as an ER net. Accordingly, we illustrate Promo by showing our implementation of these three process fragments.

Although the Promo metaphor determines much of the way in which the process is captured, we must still decide the types of artifacts modeled. Furthermore, we must determine the relationships among these artifact types. The artifact types we use are as follows:

—*Anomaly report*
—*Modification request:* Many-to-one relationship with *anomaly report*.
—*Executable:* One-to-many relationship with *anomaly report*. The executable essentially represents the product being modified, and it is possible that there should be a separate object, *Product*, that represents the

[3] SGMR is the Italian acronym for *Sottogruppo Gestione Modifiche e Rilasci*.
[4] RIMO is the Italian acronym for *Richiesta di Modifica*.

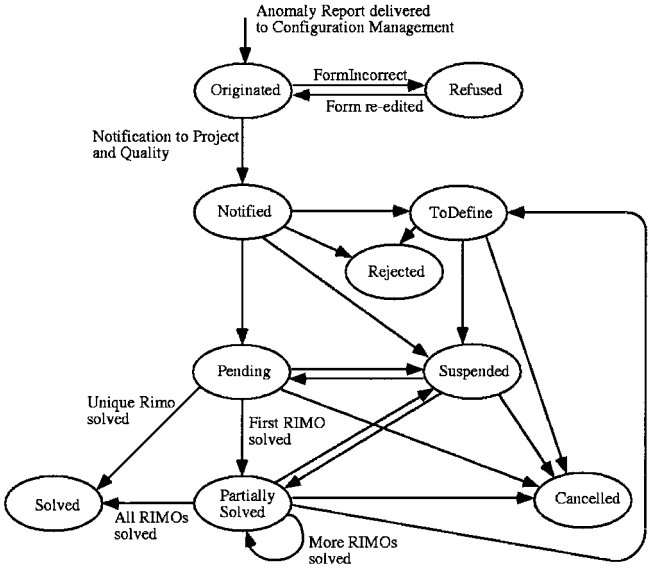

Fig. 6. SLANG state-machine model of the top-level anomaly report process [Bandinelli et al. 1995].

product. However, for the sake of simplicity we have assumed that a product consists of a single executable, so the identification of product with executable is not a difficulty.

—*Source module:* Many-to-many relationship with *modification request*.

—*Object module:* One-to-one relationship with *source module*, and many-to-many relationship with *executable*.

—*Test set:* One-to-one relationship with *modification request*.

—*Text file*

—*Input file:* Child of *text file*, and many-to-many relationship with *modification request*.

—*Output file:* Child of *text file*, and many-to-many relationship with *modification request*.

—*Test report:* One-to-one relationship with *test set*.

While additional artifact types may exist beyond these—for example, language-specific source and object code modules and editor-specific documents—the above constitute the basic types and their relationships.

7.3 Top-Level State Machine

The state machine used in the SLANG model of the process is shown in Figure 6. Since the room structure in Promo closely resembles a state machine, we begin our modeling of this top-level process by attempting to recast the state machine in terms of the Promo metaphor, i.e., in terms of tasks. In doing so, we must try to determine what task is actually being

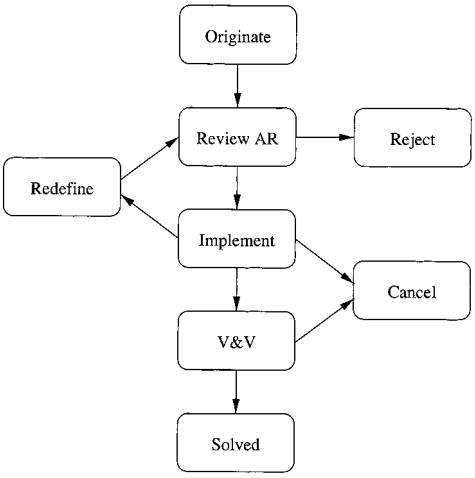

Fig. 7. Promo model of the top-level anomaly report process.

performed while an anomaly report is in a given state. If we can identify such a task, then we may consider the state and the task to be in correspondence. If not, or if the state corresponds to a task already modeled, then we must consider deleting the state from the new model.

Figure 7 shows the set of rooms designed to correspond to the top-level state machine. The two models do correspond, particularly in the initial stages when the anomaly report is the main artifact being manipulated. For example, the *Originated* state corresponds to the *Review AR* room, since the task performed on an AR in the *Originated* state is the task of reviewing the AR.

However, the state machine was intended to track the state of the anomaly report. Many states of the AR correspond to no task or to several tasks, and these states mark points where the two models differ. For example, the *Pending* state in the state machine corresponds to a great deal of activity with respect to the modification requests and ancillary reports, so this state had to be expanded into two separate tasks, each of which has subtasks. The *Suspended* state did not correspond to any task, so it was removed. The intention is that an incomplete anomaly report may just be left within the room corresponding to the task not yet complete.

The *Partially Solved* state is an interesting case. According to the state machine, it is the state entered when at least one modification request has been solved. However, the same tasks—implementing and testing modification requests—are carried out in both *Pending* and *Partially Solved*, suggesting that the two states should be unified into a single task. Furthermore, since the edges incident to *Partially Solved* are, with only one exception, identical to those incident to *Pending*, the unification of these two states is trivial. The room resulting from this unification is the *Implement* room.

Note that while the edges in Figure 7 are directed, this does not mean that each edge corresponds to a single one-way exit within Promo. Since we

wish the users to be able to walk freely among the rooms corresponding to the process, we create for each edge two exits—one forward and one backward—and simply constrain the backward edge if necessary so that no objects may pass through it.

The constraints on each edge are described below in natural language. The code for them is presented in the appendix.

—*Originate → Review AR:* The AR must have been checked for correctness of form. A successful checking sets the property **.checked** on the anomaly report to a true value, so the constraint simply checks this value.

—*Review AR → Reject:* No constraint on entering the *Reject* room, but no anomaly report may return to the *Review AR* room once it has been rejected.

—*Review AR → Implement:* The AR must have at least one RIMO associated with it. That is, the AR's **.modreq** property must be a list containing at least one descendant of the generic RIMO object.

—*Implement → Cancel* and *V&V → Cancel:* As with *Reject*, no constraints are placed on entering *Cancel*, but no AR may return via the return exit.

—*Implement → Redefine:* No RIMOs may pass through this exit, but an AR may pass through in order to allow additional RIMOs to be attached to it. The edge returning to *Implement* is unconstrained.

—*Implement → V&V:* Any RIMO passing through the exit must be complete (i.e., its **.complete** property must have a true value). Generally, completion status is set by a combination of error-free compilation of the product and adequate testing results. Furthermore, any RIMO with a test object attached must also have a test launch report attached to the test object.

—*V&V → Solved:* Any AR passing through the exit must have only completed and verified RIMOs and must have been approved (by the SGMR).

7.4 Implementation Task

The *ImplementRIMO* task in SLANG consists of an ER net that is used to implement a modification request as shown in Figure 8. Since the model of the task within SLANG is given as an encapsulated ER net, we model this in Promo by means of a subbuilding as shown in Figure 9.

The interaction with the configuration management system represented in the SLANG model can easily be represented as simple actions within Promo, so we do not need to represent these as tasks. We are left then with a simple transition between two tasks.

The only exit in the Promo model of the *Implement* task lies between *Update* and *Requalify*. The constraint on this exit requires simply that the product associated with the RIMO be up-to-date with respect to its sources. There is, of course, no way for any such system to determine whether the RIMO had actually been implemented. We use this constraint, then, simply

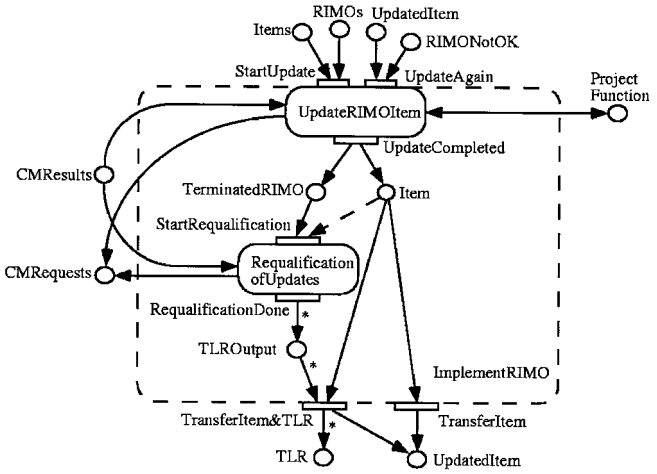

Fig. 8. SLANG model of the *ImplementRIMO* task [Bandinelli et al. 1995].

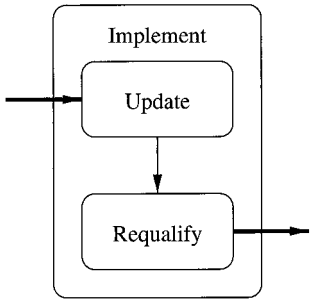

Fig. 9. Promo model of the *Implement* task.

to ensure that the product has been successfully built, so that at least the product is in a stable state.

7.5 Requalification Task

While the SLANG model of the *RequalificationOfUpdates* is fairly intricate, as shown in Figure 10, most of it centers around retrieving the tests and items. In fact, the only real action or subtask within *RequalificationOfUpdates* is the actual running of tests. In Promo, therefore, the *Requalify* task consists of only one room.

7.6 Summary

The anomaly report process described here has been implemented in Promo. External tools, such as editors and a script to run tests, can be used to manipulate the artifacts managed by the process. Engineers engaged in the process can connect to the MOO from remote sites and execute the process.

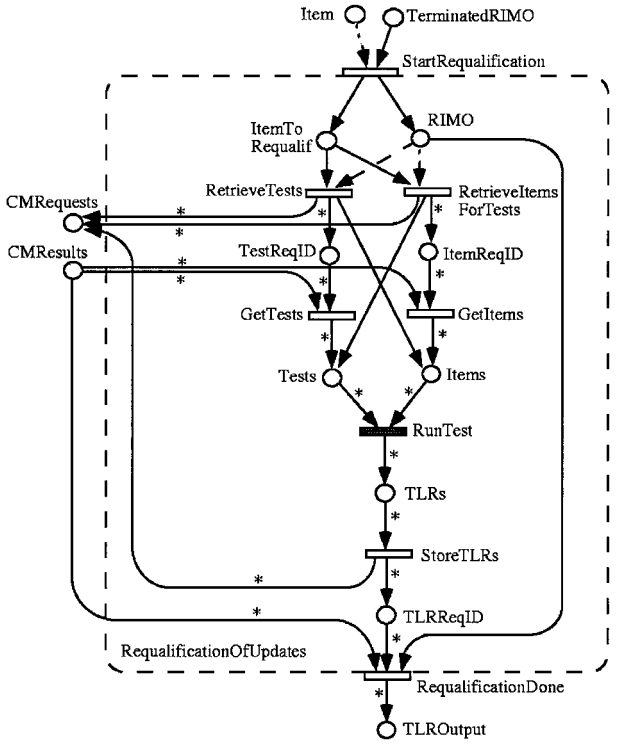

Fig. 10. SLANG model of the *RequalificationOfUpdates* task [Bandinelli et al. 1995].

One observation that may be made about the Promo version of the anomaly report process is that, despite the omission of small system-level tasks from the process, it matches the original SLANG definition quite closely. This confirms our hypothesis in Section 4.2.1 that the task-centered metaphor corresponds well to conventional (e.g., Petri net-based) notions of process. The omission of lower-level tasks in the Promo version has both advantages and disadvantages. On the one hand, the model is greatly simplified and culled down to its essential tasks. On the other hand, the omission of these details may be confusing to the uninitiated user, putting more of an onus on the process modeler to provide detailed explanations of what the user must do to travel from activity to activity.

8. CONCLUSION

Research in software process has matured to the point where it is now time to investigate suitable metaphors for interactions with process-centered environments. In the past we have offered to the user a rather direct mapping to the underlying execution model. For example, do we really want the user to think of process execution as a traversal of a Petri net?

Given the growing popularity and success of virtual environments, we have tried to understand how they might be applied in the domain of

software process. We have described a number of metaphors for process execution that fits within this context. Through a prototype, Promo, we have explored one of those metaphors, the task-centered metaphor, to some depth. Promo demonstrates the promise of virtual environments in modeling and executing software processes.

Although we chose the task-centered metaphor in designing Promo, other metaphors also show promise in their ability to model software process. One avenue of future work, then, concerns the exploration of these other metaphors and the comparison of those metaphors with the task-centered metaphor.

Such a comparison would also lead to another avenue of research: the attempt to unify these metaphors within one system. Just as a DBMS allows users to view data in any of several different ways, a virtual environment might permit different portions of the environment to represent software process using distinct metaphors. Ideally, the ability to combine metaphors would mitigate the liabilities that any one metaphor demonstrates.

A further area of promising research is to consider how the metaphors offered by virtual environments can be integrated with existing process-centered environments. For expediency, we built our prototype, Promo, completely within the virtual environment offered by LambdaMOO. Thus, while it serves to demonstrate the task-centered metaphor, Promo is necessarily limited by the capabilities of LambdaMOO to execute processes. The many years of research into process-centered environments have resulted in what are likely to be more powerful platforms.

An example of such a platform is Oz [Ben-Shaul and Kaiser 1995]. Oz is an inherently distributed process execution environment that offers the concept of multiple, interconnected workspaces. Recent extensions to Oz have made it operable within the context of the Internet [Kaiser et al. 1997]. Rules can be specified to regulate the actions that can be performed within and across workspaces. These rules provide both automation of actions and enforcement of their application. Workspaces are obvious candidates for implementing the rooms of a virtual environment. Rules provide a well-developed constraint language. Appropriately structured, they can even simulate the "bots" found in MOOs. Oz also provides a database that is akin to the databases found in MOOs. Finally, Oz supports flexible concurrency control policies.

APPENDIX

CONSTRAINTS IN PROMO

This appendix contains selected examples illustrating the constraint facility in Promo. As described in Section 6, the process modeler builds constraints by calling functions in order to create an internal constraint representation. These functions exist as verbs on a utility object called **$constraint_utils** (or **$cu** for short).

The representation built by these functions is intended to be assigned to the property **.constraint** on an exit; once this is done, the exit will automatically check this constraint with respect to every object that attempts to pass through it, and it will permit the passage of only those objects for which the constraint is true. Note that the constraint evaluation system examines not only the top-level objects that pass through it, but also the contents of any object that passes through it (and the contents of those objects). Hence, an object may be refused passage because the constraint is false for an object contained by the main object.

Ideally, constraints should be expressible in some constraint language, and the system would create the internal representation from expressions in this language rather than requiring the user to call the functions directly. Promo does not currently have such a constraint language, so we describe the use of constraint-building functions below.

Building Constraints

The verbs on the constraint utility object **$cu** that govern the building of constraints are named **make_***type*, where *type* designates the type of constraint to be built. For example, **make_relop** makes a relational operator; **make_arithop** makes an arithmetic operator, and so on. The arguments that each function takes depend on what information is required for that type of constraint, as listed below:

—**make_relop(expr1, op, expr2)** creates a relational operator; its value will be true (1) if the requested comparison is true and false (0) otherwise. Valid values of **op** are **$cu.less**, **$cu.lesseq**, **$cu.greater**, **$cu.greatereq**, **$cu.equal**, and **$cu.nequal**.

—**make_arithop(expr1, op, expr2)** creates an arithmetic expression; its value will be the result of the requested mathematical operation. Valid values of **op** are **$cu.plus**, **$cu.minus**, **$cu.mult**, and **$cu.div**.

—**make_negop(expr)** creates a logical unary negation operator; its value will be the logical negative of **expr**.

—**make_boolop(expr1, op, expr2)** creates a boolean operator (*and*, *or*, etc.); its value will be the result of the logical operation requested. Valid values of **op** are **$cu.and**, **$cu.or**, **$cu.xor**, and **$cu.implies**.

—**make_prop(obj, propname)** creates a property reference operator; its value will be the value of property **propname** on object **obj**.

—**make_let(var, expr, iconstr)** creates a Lisp-like “let” operator; its value will be the value of **iconstr** with variable **var** bound to the value of **expr**.

—**make_var(var)** creates a variable reference operator; its value will be the value of variable **var** as bound by a **let** expression. Two variables are automatically bound: **this**, which refers to the object trying to pass through the exit, and **exit**, which refers to the exit. **make_this()** and **make_exit()** are shorthand forms of **make_var(“this”)** and **make_var(“exit”)**, respectively.

—**make_isa(obj, type)** creates a type-checking operator; its value will be true if **obj** is a descendant of object **type** or false otherwise; **type** may be either an object identifier or an object name.

—**make_allop(var, set, iconstr)** creates a universal quantifier operator; its value will be true if **iconstr** is true for every **var** in **set** or false otherwise.

—**make_existop(var, set, iconstr)** creates an existential quantifier operator; its value will be true if **iconstr** is true for at least one **var** in **set**.

An additional argument may be added to any of these functions. This argument should be a string or list of strings to serve as the rationale for the constraint, as described in Section 6. If the value of the constraint is false, this string is printed out to the user as the reason why the constraint failed. The string may contain **printf**-style formatting sequences to refer to various values relevant to the constraint's failure.

Examples: Constraints in the Anomaly Report Process

The following are all of the constraints used in the modeling of the anomaly report process. They have been included here to serve as illustrative examples of the process of building constraints in Promo.

Originate → Review AR: If the object being checked is an anomaly report, then its **.checked** property must be true (i.e., nonzero).

```
$cu:make_boolop( $cu:make_isa( $cu:make_this(), $probrpt ),
  $cu.implies,
  $cu:make_prop( $cu:make_this(), "checked" ),
  "Anomaly report %n0 has not yet been (successfully) checked." )
```

Reject → Review AR: The object being checked may not be an anomaly report.

```
$cu:make_negop( $cu:make_isa( $cu:make_this(), "anomaly report" ),
  { "Sorry, rejected ARs can't be brought back from the dead--",
   " you'll have to drop %o0 before you can leave."} )
```

Review AR → Implement: If the object being checked is an anomaly report, its **.modreq** property must not be the empty list ({}).

```
$cu:make_boolop( $cu:make_isa( $cu:make_this(), $probrpt ),
  $cu.implies,
  $cu:make_relop( $cu:make_prop( $cu:make_this(), "modreq" ),
    $cu.nequal,
    $cu:make_quote( {} ) ),
  {"You cannot begin implementing AR %n0 without adding at least
    one RIMO to it.",
   "* Use '@setrv modreq ⟨RIMO⟩ on %n0' to add an RIMO to it."} )
```

Implement → *Cancel*: The object being checked may not be an anomaly report.

```
$cu:make_negop( $cu:make_isa( $cu:make_this(), "anomaly report" ),
    {"Sorry, canceled ARs can't be brought back from the dead--",
     "you'll have to drop %o0 before you can leave."} )
```

V&V → *Cancel*: The object being checked may not be an anomaly report.

```
$cu:make_negop( $cu:make_isa( $cu:make_this(), "anomaly report" ),
{"Sorry, canceled ARs can't be brought back from the dead--",
 "you'll have to drop %o0 before you can leave."} )
```

Implement → *Redefine*: The object being checked may not be a modification request.

```
$cu:make_negop( $cu:make_isa( $cu:make_this(), "modification request" ),
  {"Sorry, RIMOs are not allowed in the Redefine activity.",
   "* Please drop them here before continuing."} )
```

Implement → *V&V*: If the object being checked is a modification request, it must be complete, and all its tests (if such tests exist) must have compared successfully with the desired outputs.

```
$cu:make_boolop( $cu:make_isa( $cu:make_this(), "modification request" ),
  $cu.implies,
  $cu:make_boolop( $cu:make_prop( $cu:make_this(), "complete" ),
    $cu.and,
    $cu:make_let( "ts", $cu:make_prop( $cu:make_this(), "test_set" ),
      $cu:make_boolop( $cu:make_relop( $cu:make_var( "ts" ),
          $cu.equal, #-1 ),
        $cu.or,
        $cu:make_allop( "x",
          $cu:make_prop( $cu:make_var( "ts" ), "tests" ),
          $cu:make_relop( $cu:make_prop( $cu:make_var( "x" ),
                                "compare" ),
            $cu.equal, 1 ),
          "Test %o2 did not accurately compare with its output." )
      ))))
```

V&V → *Solved*: If the object being checked is an anomaly report, its **.complete** property must be true, and each of its modification requests must have a true-valued **.complete** property as well.

```
$cu:make_boolop( $cu:make_isa( $cu:make_this(), $probrpt ),
  $cu.implies,
  $cu:make_boolop( $cu:make_relop( $cu:make_prop( $cu:make_this,
                                                  "complete" ),
      $cu.nequal,
      0,
```

```
    "Anomaly report %o0 has not been flagged as complete" ),
  $cu.and,
  $cu:make_allop( "ar",
    $cu:make_prop( $cu:make_this(), "modreq" ),
    $cu:make_prop( $cu:make_var( "ar" ), "complete" ),
    "Anomaly report not yet finished: RIMO %o2 is not com-
    plete." ) ) )
```

ACKNOWLEDGMENTS

This work benefited greatly from discussions with Alfonso Fuggetta and Elisabetta di Nitto of the Politecnico di Milano. A valuable resource for information on MOOs and MUDs can be found at http://www.moo.mud.org.

REFERENCES

BANDINELLI, S., FUGGETTA, A., GHEZZI, C., AND LAVAZZA, L. 1994. SPADE: An environment for software process analysis, design, and enactment. In *Software Process Modeling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Wiley, London, U.K., 223–248.

BANDINELLI, S., FUGGETTA, A., LAVAZZA, L., LOI, M., AND PICCO, G. P. 1995. Modeling and improving an industrial software process. *IEEE Trans. Softw. Eng. 21,* 5 (May), 440–454.

BEN-SHAUL, I. S. AND KAISER, G. E. 1995. *A Paradigm for Decentralized Process Modeling.* Kluwer Academic, Boston, Mass.

BERNERS-LEE, T., MASINTER, L., AND MCCAHILL, M. 1994. Uniform resource locators (URL). RFC 1738, Internet Engineering Task Force.

BORENSTEIN, N. AND FREED, N. 1992. MIME (multipurpose Internet mail extensions): Mechanisms for specifying and describing the format of Internet message bodies. RFC 1341, Internet Engineering Task Force.

BRUCKMAN, A. 1994. Programming for fun: MUDs as a context for collaborative learning. In *Proceedings of the National Educational Computing Conference* (Boston, Mass., June). National Education Computing Assoc.

BRUCKMAN, A. AND RESNICK, M. 1995. The MediaMOO project: Constructionism and professional community. *Convergence 1,* 1 (Spring).

CARLSTROM, E.-L. 1992. Better living through language. The communicative implications of a text-only virtual environment, or, Welcome to LambdaMOO! Gronnell College.

CHERNY, L. 1995a. The modal complexity of speech events in a social MUD. *Elec. J. Commun. 5,* 4 (Nov.).

CHERNY, L. 1995b. The situated behavior of MUD back channels. In *Proceedings of the AAAI Spring Symposium*. AAAI, Menlo Park, Calif.

CURTIS, B., KELLNER, M. I., AND OVER, J. 1992. Process modeling. *Commun. ACM 35,* 9 (Sept.), 75–90.

CURTIS, P. 1992. Mudding: Social phenomena in text-based virtual realities. Xerox PARC, Palo Alto, Calif.

CURTIS, P. 1996. LambdaMOO programmer's manual. Xerox PARC, Palo Alto, Calif. For LambdaMOO version 1.80p5.

CURTIS, P. 1997. Not just a game: How LambdaMoo came to exist and what it did to get me back. In *Highwired*: *On the Design, Use and Theory of Educational MOOs*, C. Haynes and J. R. Holmevik, Eds. University of Michigan Press.

CURTIS, P. AND NICHOLS, D. 1993. MUDs grow up: Social virtual reality in the real world. In *Proceedings of the 3rd International Conference on Cyberspace*. Xerox PARC, Palo Alto, Calif.

CURTIS, P., DIXON, M., FREDERICK, R., AND NICHOLS, D. A. 1995. The Jupiter audio/video architecture: Secure multimedia in network places. In *Proceedings of the 3rd Annual ACM*

International Multimedia Conference and Exposition (San Francisco, Calif., Nov. 5–9). ACM, New York.

FEILER, P. H. AND KAISER, G. E. 1987. Granularity issues in a knowledge-based programming environment. *Inf. Softw. Tech. 29,* 10 (Dec.), 531–539.

FINKELSTEIN, A., KRAMER, J., AND NUSEIBEH, B., Eds. 1994. *Software Process Modeling and Technology*. Wiley, London, U.K.

FITZPATRICK, G., KAPLAN, S. M., AND MANSFIELD, T. 1996. Physical spaces, virtual places, and social worlds: A study of work in the virtual. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*. ACM, New York, 334–343.

FUGGETTA, A. AND WOLF, A. L., Eds. 1996. *Trends in Software*. Vol. 4, *Software Process*. Wiley, London, U.K.

GHEZZI, C., MANDRIOLI, D., MORASCA, S., AND PEZZÈ, M. 1991. A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. Softw. Eng. 17,* 2 (Feb.), 160–172.

GISI, M. A. AND KAISER, G. E. 1991. Extending a tool integration language. In *Proceedings of the 1st International Conference on the Software Process*. IEEE Computer Society Press, Los Alamitos, Calif., 218–227.

HEIMBIGNER, D. 1990. Proscription versus prescription in process centered environments. In *Proceedings of the 6th International Software Process Workshop*. 99–102.

HEIMBIGNER, D. AND OSTERWEIL, L. J. 1994. An argument for the elimination of roles. In *Proceedings of the 9th International Software Process Workshop*. IEEE Computer Society Press, Los Alamitos, Calif., 122–123.

HEINEMAN, G. T., KAISER, G. E., BARGHOUTI, N. S., AND BEN-SHAUL, I. Z. 1992. Rule chaining in Marvel: Dynamic binding of parameters. *IEEE Expert 7,* 6 (Dec.), 23–32.

JUNKERMANN, G., PEUSCHEL, B., SCHÄFER, W., AND WOLF, S. 1994. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In *Software Process Modeling and Technology*, A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds. Wiley, London, U.K., 103–129.

KAISER, G. E., BARGHOUTI, N. S., FEILER, P. H., AND SCHWANKE, R. W. 1988. Database support for knowledge-based engineering environments. *IEEE Expert 3,* 2 (Summer), 18–32.

KAISER, G. E., BARGHOUTI, N. S., AND SOKOLSKY, M. H. 1990. Preliminary experiences with process modeling in the Marvel software development environment kernel. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*. Vol. 2. IEEE, New York, 131–140.

KAISER, G. E., DOSSICK, S. E., JIANG, W., AND YANG, J. J. 1997. An architecture for WWW-based hypercode environments. In *Proceedings of the 1997 International Conference on Software Engineering*. ACM, New York, 3–13.

KAPLAN, S. M., FITZPATRICK, G., MANSFIELD, T., AND TOLONE, W. J. 1997. MUDdling through. In *Proceedings of the 30th Annual Hawaii International Conference on System Sciences*. Vol. 2. IEEE Computer Society, Washington, D.C., 539–548.

MASINTER, L. AND OSTROM, E. 1993. Collaborative information retrieval: Gopher from MOO. In *Proceedings of INET '93*. The Internet Soc.

POSTEL, J. AND REYNOLDS, J. 1983. Telnet protocol specification. RFC 854. Internet Engineering Task Force.

RAYMOND, E. S., Ed. 1996. The Jargon file. Available via http://www.fwi.uva.nl/%7Emes/jargon.

REID, E. 1994. Cultural formations in text-based virtual realities. Master's thesis, Univ. of Melbourne, Melbourne, Australia. Jan.

VALETTO, G. AND KAISER, G. E. 1996. Enveloping sophisticated tools into process-centered environments. *J. Automated Softw. Eng. 3,* 309–345.

Received November 1996; revised February and July 1997; accepted August 1997