Software Process Modeling and Execution Within Virtual Environments

John C. Doppke
doppke@cs.colorado.edu

CU-CS-805-96 July 1996

University of Colorado at Boulder

Technical Report CU-CS-805-96
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Software Process Modeling and Execution Within Virtual Environments

John C. Doppke
doppke@cs.colorado.edu

July 1996

Abstract

While multi-user virtual environments have been developed in the past as venues for entertainment and social interaction, recent research in virtual environments has focused on their utility in carrying out work in the real world. This recent research has identified the importance of a mapping between real and virtual that permits the representation of tasks in the virtual environment. In this paper 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 process within a MUD called `LambdaMOO`. The system resulting from this mapping, called PROMO, permits the modeling and execution of software processes.

Keywords: Software process, virtual environments, MUD.

Contents

The material contained herein is based upon work sponsored by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253.

List of Figures

Chapter 1
Introduction

> **connect guest**
** Connected ***
The LEGO Closet
It's dark in here, and there are little crunchy plastic things under your feet! Groping around, you discover what feels like a doorknob on one wall.
Obvious exits: out to The E&L Garden
> **out**
The E&L Garden
The E&L Garden is a happy jumble of little and big computers, papers, coffee cups, and stray pieces of LEGO.
Obvious exits: hallway to E&L Hallway, closet to The LEGO Closet, and sts to STS Centre Lounge
You see a newspaper, a Warhol print, a Sun SPARCstation IPC, Projects Chalkboard, and Research Directory here.
Amy is here.
> **say hi**
You say, "hi"
Amy says, "Hi Guest! Welcome!"

Figure 1: A transcript of a portion of a MediaMOO session.

Virtual environments have typically been developed in the past primarily for their entertainment value. Indeed, the earliest MUDs survived precisely because of their popularity as games.

Recently, however, a movement began towards the creation of MUD-like environments whose main purpose was not gaming but the building of virtual worlds. While the earliest systems with this purpose were still intended for entertainment value, the promise of an extensible virtual environment has intrigued researchers and has prompted the question of the viability of such systems in aiding work and collaboration in the real world.

In this paper, then, we examine the use of virtual environments—in particular, a MUD called `LambdaMOO`—in support of software process modeling and execution. We have chosen software process because it provides us with a "bird's-eye view" of tasks; that is, instead of attempting to model a specific task within the MOO, we are attempting model the process of completing tasks within the MOO.

Before investigating the possibility of integrating the two domains of virtual environments and software process, we shall examine each domain separately.

Chapter 2
Software process

A software process is typically defined as a set of *activities* carried out to build, deliver, and evolve a software product, from the inception of an idea to the delivery and retirement of a system [23]. However, this seemingly simple definition requires a great deal of elucidation in order to determine what pieces make up a process and how they interact.

Software process has become an area of active research. One major area of research concerns the representation of processes using a consistent (and often formal) *process model.* A second area of research concerns the development of software to support the development of software based on a given process; such products are referred to as *process-centered environments.* Such systems require that they be given some process model in order to support the execution of the process.

We shall discuss process in further detail, then, by first laying out definitions of the different entities involved in processes—that is, all the various pieces that are part of any consideration of process. Based on these definitions, we then list the types of support a process-centered environment should provide.

2.1 Definitions

2.1.1 Activity

The notion of an activity within a process requires some elucidation. Since the granularity of the process model is largely an issue left to the discretion of the modeler, different models for the same process may disagree about what the activities within the process are. Within this paper, we will use the word *activity* (and the word *task*) to denote some sequence of one or more *operations* (or *actions*). 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 or task because we have a semantic understanding of how the specific operations within the testing task (generation of test cases, running the program against them, etc.) fit together. We may also desire the definition of *subtasks* within larger tasks, thus forming a hierarchy of tasks.

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

2.1.1.1 Sequencing and history

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.

2.1.1.2 Actions

Actions within the process may or may not be understood to be atomic; for the sake of simplicity, we shall assume 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 also requires the availability of an appropriate *tool.*

2.1.2 Agents

Any discussion of operations and activities within a process raises the issue of *agents*—the human or machine 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 [15]; 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 [24] eschew this notion of roles because of its tendency to conflate issues that should remain separate: threads of control, unification of similar activities, and access control.

2.1.3 Artifacts

Software process also governs the definition of *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 form parts of the larger project, their interrelationships must also be defined.

2.1.4 Product

Finally, the process concerns the final *product* itself. We may also choose to partition the product into *subproducts* in much the same way as we partitioned tasks into subtasks. Note that although a product generally contains a set of artifacts, the product also corresponds to a larger view of the product; it can be thought of, in essence, as a project or reified process.

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 IDs 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.

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., graphical or textual), the term is intended here to designate a whole set of methods through which the user may query the system about the process. The user may wish to know information about the current state of the process—for example, what human agent is carrying out which task with respect to what artifact(s)—and the system must provide a means of answering these questions. Furthermore, it must provide this means within the framework of the representations chosen for the entities and their connections.

Chapter 3
MUDs and Virtual Environments

Multi-User Dimensions, or MUDs, were first created as simple text-adventure 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 a network, or via the telephone system) and that presents to these users a virtual world in which each user is represented as a *player.* A diagram of this architecture is given in Figure 2. MUDs began as text-only systems, and nearly all

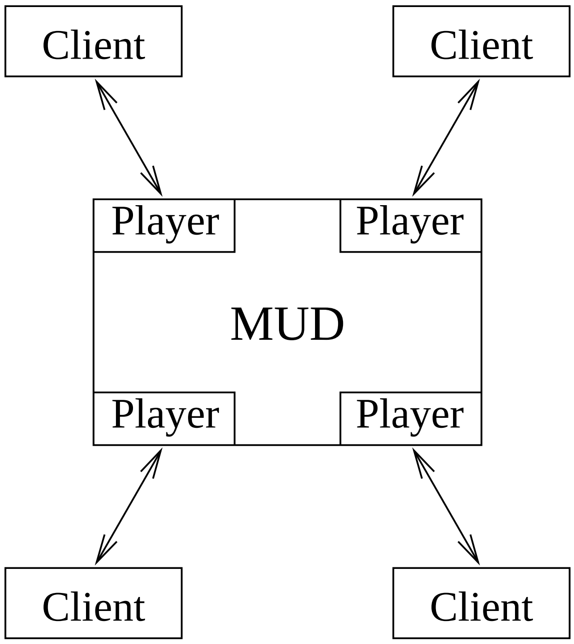

Figure 2: The client/server architecture of a typical MUD.

are still essentially text-based; however, many now are beginning to offer more sophisticated interfaces of various kinds [20, 26, 35, 6].

The world that the MUD provides to the user consists of a set of rooms and, within those 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 3.

Noticeable differences of opinion exist on the "point" of a MUD. While MUDs were begun as multi-user games in the spirit of text-adventure games (such as ADVENT [32]), 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

>**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.

Figure 3: A sample transcript of a session with a 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 technical; many MUDs are devoted to a certain topic (e.g., biology [6] or media research [30]), 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 History

The original MUD, entitled "Multi-User Dungeon" (and originally abbreviated `MUD` but referred to as `MUD1` in the literature), was created by Roy Trubshaw at Essex University in 1978 [4, 11]. Trubshaw's interest in the game was mostly technical: `MUD1` was the first MUD to have been designed as a multi-user game, as opposed to the multi-user variants of Adventure-style games; and `MUD1` had a database design language whose development was one of Trubshaw's chief interests in the system [3]. Richard Bartle became involved in the effort shortly after it was created and was primary in designing the game-oriented aspects of `MUD1`.

Despite the programmer's technical intentions, `MUD1` became wildly popular as a game and is generally recognized as the ancestor of nearly every MUD in existence (a notable exception being Klietz's "Scepter of Goth" [11]). `MUD1` already contained most of the basic features associated with MUDs: the ability to manage of multiple players, both local and remote; a database of rooms, along with the ability to add new rooms during play; an environment centered around fantasy and role-playing, along with game-related features like scoring systems; and so on. `MUD1` thus became quite seminal in this field—in fact, as of 1990, a version of `MUD1` was still running on the online service CompuServe.

The popularity accorded to `MUD1` led to the creation of numerous MUDs derivative of `MUD1`; Bartle's 1990 survey [3] lists some 47 types of MUDs. Many of these MUDs were essentially similar, both in features and in spirit, to `MUD1`. However, a turning point occurred in 1989 with the release of two new independent systems, `LPMUD` and `TinyMUD`, that both featured greatly enhanced world-building facilities. In particular, both systems were among the first to make player-created objects persistent (whereas such player creations were not persistent in previous systems).

3.1.1 `TinyMUD` vs. `LPMUD`

One important difference between these new systems lay in the fact that whereas only the "wizards" (experienced players) on `LPMUD` were allowed to build objects and rooms, any player was entitled to do so within `TinyMUD`. While this distinction may seem minor, it caused a severe rift among the otherwise similar systems that persists to this day: namely, the focus of `TinyMUD` shifted to the building of the virtual world and to the social interaction within that world, whereas `LPMUD` remained more game-like in its focus. Fans of MUDs as games (such as the creators of game-like MUDs [3]) reviled `TinyMUD` for its lack of adventure and for its chaos.

`TinyMUD`, however, thrived as a new kind of MUD and gained popularity among a new set of people—as with `MUD1`, much more popularity than its creator, Jim Aspnes, expected. Furthermore, a whole collection of systems were created based on `TinyMUD`. One such derivative system, `TinyMUCK`, featured some limitations on who could build rooms and objects, and it provided its own language, `TinyMUF` ("Multi-User Forth"), as the building language. In May 1990, Stephen White created `TinyMOO` as a derivative of `TinyMUCK`, where MOO stood for "MUD, Object-Oriented." `TinyMOO` provided a C-like language along with some object-oriented features such as a class hierarchy and inheritance.

3.2 LambdaMOO

In October 1990 [11], Pavel Curtis of XEROX PARC brought up his new system, `LambdaMOO`, for the first time. Based on `TinyMOO`, `LambdaMOO` provided a fully object-oriented language tightly coupled with an object-oriented database. Since the original database—the so-called "core" database—was fairly small, the original opening was not publicly announced; however, by February 1991 the database had become sufficiently populated that Curtis opened it to the public. The term `LambdaMOO` thus refers both to the MOO system and to its original incarnation, though generally we shall use it to refer to the system.

3.2.1 Architecture

A `LambdaMOO` system consists of two parts: the server program [18] 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` [17]; the objects in this database, numbering about 100–150, comprise some basic MUD-related objects (player, room, exit, etc.) along with some utility functions (string and list manipulation, coding and network utilities, etc.). 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. Some server patches and modifications do exist, and recently new "core" databases have been offered by MOOs besides the `LambdaMOO` at XEROX PARC (for example, JHM [26]).

3.2.2 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 OO systems) [19]. 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 OO 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.

3.2.2.1 Common objects

The `LambdaMOO` object model is sufficiently rich to support the modeling of any types of objects the programmer may wish. 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:

- *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 which 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.
- *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 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—all without disrupting the people inside.

3.2.3 Containment

Two built-in properties on every `LambdaMOO` object are `.location` and `.contents`, and together these properties 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 tehse 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 main 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 that object; if `.location` is a player, we may say the player is holding the object; if `.location` is the Recycler object, then the object is a "dummy" object waiting to be recreated. 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.

3.3 Virtual environment research

Our interest in MUDs focuses on their place in the more general field of virtual environments. While some researchers [20] have used the term *virtual reality* to describe systems such as MUDs, this term typically refers to systems that attempt to represent the real world accurately (using three dimensions, etc.). In MUDs this is often *not* the case: the absence of a real-world–like spatial metaphor, the ability to perform tasks that are impossible in real life (e.g., teleporting objects), and the MUD-specific communicative and social forms [12] all point to a major difference between MUDs and reality. Accordingly, we shall use the more generic and less loaded term *virtual environment* to refer to any system that presents a (possibly unrealistic) world or space for users to visit and inhabit.

3.3.1 Constructionism

Bruckman [8, 9, 10] 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 [9] 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 [30], the paper illustrates the effectiveness of the constructive aspect of the virtual in encouraging interaction within a professional community.

3.3.2 Work in the virtual

Kaplan's work [22] on the use of virtual environments to accomplish work in the real world makes some useful distinctions. 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: that is, a locale is an abstraction of the virtual space in terms of the semantics of the real-world domain.

Chapter 4

Implementing Software Process Within Virtual Environments

Given the potential utility of virtual environments as systems for accomplishing work in the real world, we examine the potential for using such environments with respect to software process. It behooves us, then, to explain what virtual environments can offer to software process and what forms such systems might take. We begin by discussing the motivation for such a system, continue by listing some different metaphors that might prove useful, and conclude by discussing some other issues related to the system.

4.1 Motivation: Metaphor

As pointed out above, the chief 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 2.2 we laid out some requirements for what a process-centered environment must provide to its users. Chief among these requirements was the need to represent the entities of software process and the connections among these entities within the system. In the case of a virtual environment, then, 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 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 undly restrict access to artifacts.
- *Collaboration:* One of the more appealing aspects of virtual environments is virtual collocation, or the ability to be in the same "space" with another person in the virtual world despite physical separation from that person in reality. A system that implements software process ought to use this collocation to its advantage and to allow maximal collaboration as a result.
- *Realism:* While a virtual world need not be identical to the real world, it should represent a limited extension of concepts in the real 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 many cases, for example, one may achieve greater realism at the cost of diminished correspondence. Furthermore, it may be acceptable or even preferable for a metaphor not to correspond exactly to software process; 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 of choice, 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 mapping, we presume that this is a part of any metaphor, and so we shall choose 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.

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 defined 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: for example, by reifying tools and using them to effect the action, or by using the MOO's verbs to invoke actions.

This metaphor corresponds well to more conventional notions of process activities, especially those that use state machine– or Petri net–based process modeling languages; it also maps well to a fairly natural human concept of motion as progress. One can imagine, for example, the MOO artifact objects as the manifestation of tokens in the Petri net and the player being the force that pushes them through transitions. Furthermore, the metaphor does support some collaboration, in that people working on the same task are in the same room together; however, those performing distinct tasks 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"—that is, they alternate among these tasks—the MOO using this metaphor needs to provide for the ability for a person to leave one project and go back to another at the same point where she or he left off. `LambdaMOO` does not provide this capability, and in PROMO we have had to add a facility to support this alternation of tasks. We shall refer to this problem as one of "multi-threading" to suggest the similarity between this alternation among tasks and the manner in which operating systems alternate among threads of control within a concurrent problem. The multi-threading 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 shall examine the metaphor in greater detail in chapter 5.

4.2.2 Agent-centered metaphor: Workspaces

The phrase "agent-centered metaphor" may seem a bit misleading; given the use of terminology for the other metaphors, this would seem to indicate that agents were being mapped onto rooms. Instead of performing this seemingly strange mapping, we map the notion of agent onto a MOO player but map the notion of person onto a MOO room. In this way, the MOO player performs only one task at a time, and the person's many "threads" are unified in that all the person's activities are carried out in her or his room. For this reason we refer to this metaphor as the *workspace* metaphor: the person's room acts as her or his personal workspace.

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; 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.

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 (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 process often involves manipulating and combining artifacts, this is a point where the metaphor severely fails to correspond to process. 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 [29] 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 Product-centered metaphor

A product-centered metaphor would map each separate product onto a different room. If sub-products 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.

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.

4.2.5 Hybrid metaphors

After spending a great deal of time defining semantics of different mappings, we should note that in the real world people do not require precise semantics. Rooms in the real world may be task-related rooms (meeting rooms or classrooms), workspaces (offices), artifact-centered or product-centered rooms , or tool-centered rooms (workshops); and we are not put off by the mixing of these distinct types of rooms.

However, even though the mere presence of different types of rooms within a MOO is not problematic, it may be useful for the software process side of a system to enforce specific semantics. One possible avenue of future research concerns 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. In such an environment, one could enforce one metaphor within one set of rooms and a different metaphor within another and thereby create a hybrid system.

4.3 Other issues

4.3.1 Interface

Once a means of representing a process within a virtual environment is defined, the environment must provide the user with an interface to processes—i.e., a means of interacting with processes. While the interface issue encompasses the narrower issue of user interface—i.e., textual, graphic, etc.—it also encompasses a whole 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:

- *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.

4.3.2 Proactivity

A second question raised by the fact of executing processes via software 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 [25] and events and triggers [1].

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.

Chapter 5
Design of PROMO

In an effort to illustrate the concepts discussed in the previous chapter, we have created PROMO, a prototype process-centered environment based on the LambdaMOO system. The metaphor used in PROMO is the task-centered metaphor described in section 4.2.1. PROMO is based on and extends the LambdaMOO system [18] and its core database, LambdaCore [17].

In this chapter we shall discuss the design decisions made in creating PROMO in light of the discussion of issues in the previous chapter. In doing so, we shall evaluate the specifics of the metaphor using the criteria set out in section 4.1. A discussion of the architecture of PROMO along with a discussion of more technical issues regarding its implementation are in chapter 6.

5.1 PROMO metaphor

PROMO represents software process within LambdaMOO using the task-centered metaphor described in section 4.2.1. Our description of PROMO must begin with a more in-depth examination of the PROMO metaphor. We state this metaphor as a mapping of process entity to PROMO representation, and in the process of detailing this mapping, we also indicate how these representations interact. As we shall see, the role of process modeler is, as in all process-centered environments, a key role; the decisions made by this person (or this set of people) regarding the granularity of modeling have a profound effect on the resulting PROMO system. Our discussion below therefore highlights the likely effects of differing 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 is 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 [5] that acts as a pointer to the actual artifact; beyond that, the object may contain as much or as little information about the actual artifact as is desired by the modeler (i.e., as much as is needed for the process).

The artifact-to-object mapping thus 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 synch becomes a difficult task. Any tool that modifies the artifact must propagate all information about the changes to the artifact back into PROMO. (Section 6.3 goes into 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.

PROMO can, of course, represent information about an artifact that may not be contained within the artifact's file(s). In particular, properties on the PROMO object can represent inter-object relationships and artifact state. For example, a source module may contain a "pointer" to (i.e., the object ID 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 module object, create a generic C source module object as its child, and then create specific C source module objects as children of the generic C source module. Indeed, one of the core Promo objects is a generic artifact object called `$artifact`, and all artifact objects are expected to be descendants (i.e., children, or children of children, etc.) of `$artifact`.

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 3.2.3 raises an interesting questions 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 real world does not contain pointers in this sense.

While 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 real world, satisfy the technical constraints of implementing software process, and still maintain the consistency of the metaphor. (A possible implementation for this concept is suggested by Smith's solution to an unrelated problem in [34].)

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 conflates too many issues that should be separated [24].

Instead, we identify each of the person's threads as a *persona*: that is, 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; if the person wishes to change threads, she or he may simply switch 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.

5.1.3 Action: Verb

A given action (or operation) within a process is modeled within Promo as a verb. For example, in order for a player to edit an artifact, she or he issues the command

```
edit <object-name>
```

where `<object-name>` is the name of the object corresponding to that artifact.

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, the action may take an arbitrarily long time to complete. Similarly, within the MOO, a single verb call is also intended to take place immediately, 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 very distant from the actual MOO, 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.

An alternate metaphor for 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 metaphor within PROMO, it is conceivable that this would be desirable (as in [29]). 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 we defined a task as a semantically meaningful set of actions, we must specify how to define tasks. 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.

5.1.4.1 Sub-buildings

In support of the concept of sub-tasks, PROMO provides the ability to create a "sub-building"—that is, a building within a room. The sub-building may have as many rooms within it as are desired, and it may have as many doors as are desired. The sub-building thus enables passage through a sequence of rooms, with their own exits and constraints, while staying within a larger room.

The sub-building model provides two main advantages. First, it permits the representation of sub-tasks while maintaining the containment of objects. That is, if an object is contained within a room corresponding to a sub-task, 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 the enclosing *Implement* task. Second, sub-buildings are a means of implementing transactions: for example, the first room in a sub-building could have an exit constraint that a transaction have been begun, and the sub-building's exit door could require the successful termination of that transaction. The main disadvantage of the sub-building model is that it does not correspond to a real phenomenon; enclosed spaces within larger rooms are not commonly found in reality.

5.2 Interface

While the semantics of the process-to-MOO mapping are important, the interface by which the user interacts with PROMO is equally important. Below we discuss how PROMO addresses the interface issues previously mentioned. Admittedly, many interface issues within PROMO remain in the realm of future work.

5.2.1 Process state

Because of the task-centered model that PROMO uses, the state of a given artifact may 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 sub-tasks, 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 between artifacts. Currently, a player may walk through the set of rooms corresponding to the process and examine the objects found therein to determine process state.

5.2.2 Process training

Trying to understand an unfamiliar process can be a cold and forbidding task, particularly in a non-proactive system (like PROMO) wherein finding out what actions are not permitted is easier than finding out what actions are possible. Since MOOs are text-based and typically foster a great facility with textual description [31], PROMO places the onus of describing the process on the shoulders of the process modeler.

However, the structure of rooms and exits in PROMO does provide a convenient structure on 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 `look`ing at a room lists all its exits, which may also have documentation strings attached to them. Hence, simply by `look`ing at a room, the player may find out what the room is intended for as well as what conditions are necessary to exit in any of several ways.

A further facility for documenting processes is provided in the constraint creation facility. A 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 perhaps indicate some action or set of actions that are needed to cause the successful evaluation of the constraint.

5.2.3 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 what 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 [27].

5.2.4 User Interface

One common criticism of MUDs is their basis in textual communication. While this textual foundation often leads to forms of communication that are of interest to sociologists and linguists [13, 14, 12, 33, 16], the interface is considered by some to be adequate at best in terms of usability [21, 27].

One of the main moves in MOO interface work has involved a push for WWW interfaces; several MOOs [6, 35, 26] now offer such 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, 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 [6] may provide an exception to this rule; the interface combines a traditional textual connection with a WWW interface.

PROMO currently uses the traditional textual interface, although the addition of a WWW interface (in addition to, not instead of, the usual interface) is another possible future direction.

5.2.5 Tool interface

PROMO's interaction with tools occurs solely through the action invocation system described in sections 5.1.3 and 6.3. 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.

Chapter 6
Implementation of PROMO

This chapter discusses the implementation of the important features of PROMO. While this discussion is by no means intended to be a complete account of the implementation, it touches on the important issues arising from the design of PROMO discussed in the previous chapter.

6.1 Artifacts

As previously mentioned, 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; it merely acts as the manifestation of the artifact within PROMO. Instead, the artifact's data is 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 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.

Currently PROMO supports only URLs corresponding to local files. However, the use of URLs provides a mechanism for future support for remote artifact retrieval and manipulation.

6.2 Constraints

The exit constraints used in PROMO represent a fairly powerful and complete method of guaranteeing task completion on exits. Each exit has a property, `.constraint`, that stores the constraint that must be satisfied in order for that exit to permit an object through it; 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 that are built by calling functions to create an internal representation. Since certain types of constraints take other constraints as arguments, the constraint can built in 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 building constraints may be avoided.

6.2.1 Constraint types

The types of constraints available include:

- *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`

6.2.2 Constraint rationales

As previously mentioned, every constraint may have attached to it a string that the user will see should the constraint fail. Called the *rationale,* this string makes the failure of a transition constraints much easier for a user to understand, and liberal use of rationales is highly recommended to the process modeler.

6.3 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.

A diagram of the action invocation system in PROMO is shown in Figure 4 and is described in more detail below.

6.3.1 PROMO to client

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.

A number of clients exist for MUDs 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 client supported by PROMO is TinyFugue [28].

Among the information sent to the client are the following:

- a unique ID identifying this action;
- the object ID and name of the object on which the action is being carried out;

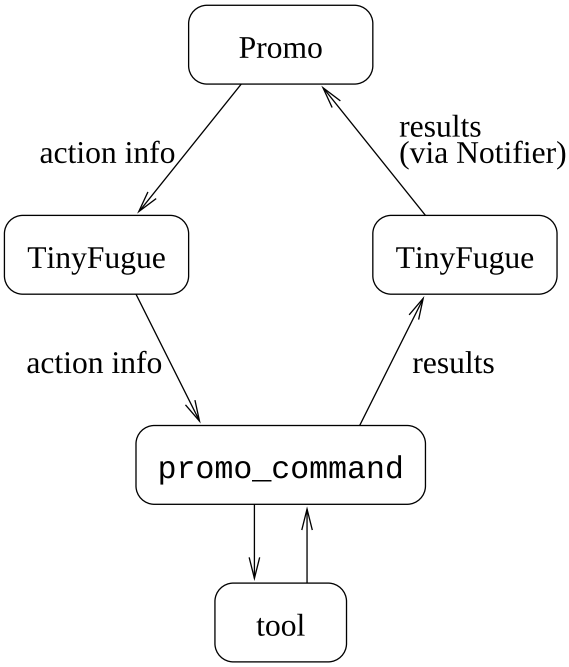

Figure 4: Architecture of the Promo action invocation system.

- the object ID and name of the player carrying out the action;
- the object ID and name of the room in which the action was invoked;
- the MIME type[1] of the artifact; and
- the name of the action as a string (e.g., `edit`).

Promo also supports arbitrary extensions to this block of information; a verb exists in the generic artifact object that can be overridden by artifact subtypes that allows the addition of arbitrary tag-value pairs to this block.

6.3.2 Client to `promo_command`

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`, which is responsible for invoking the tool. `promo_command` uses the action's name and the artifact's MIME type, along with a mapping file, in order to determine which tool should be invoked.

6.3.3 `promo_command` to specific tools

`promo_command` is written in Tcl and relies on the ability of the system administrator to encapsulate every tool within Tcl. `promo_command` first parses the block of information received

[1] We use a MIME-like notation—type and subtype—to inform the client of the type of an artifact's type. The types used, however, are not to be confused with officially registered MIME types [7].

from the client; it 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.

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

- A list of *events* that the action generated. The events are strings that should be understood by the artifacts within the MOO; making use of these events will be described below.
- A *comment* describing the result of the action (for human consumption).
- 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.

6.3.4 `promo_command` to Promo

`promo_command` then sends the results back to Promo. In fact, the mechanism for communicating these results is the same client used by the user, TinyFugue; `promo_command` accomplishes this communication by defining triggers and then running TinyFugue in batch mode. In particular, `promo_command` uses the client to log into the MOO as a pseudo-player called the Notifier, issue special commands available only to this pseudo-player 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 a list of property specifications belonging to the artifact; 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.

Chapter 7
Example Process

As an illustration of the use of PROMO, we have taken a problem report process from the literature [2] and implemented it using PROMO. As the paper from which the process was taken presented only selected illustrative portions of the process, so we shall present those portions of the process that correspond to those found in the paper.

7.1 Statement of the process

The process in question concerns the generation and handling of problem reports (PR), also called anomaly reports, for a piece of software. The steps in this process are as follows:

- The configuration management group (SGMR), which is in charge of the problem report, checks the report for correctness of form (all the proper fields must be filled in, etc.).
- The SGMR then summons the Change Control Board (CCB), which reviews the problem report to determine whether it is a valid report. If the CCB decides that it is not valid, the report is rejected. If it is found to be valid, the CCB accepts it and generates modification requests (MRs, called RIMOs in the paper)—i.e., requests for changes to made to the product in order to fix the problem. The CCB may also suspend its investigation of an problem report.
- Development and Qualification groups implement the requested modifications, and 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.
- A review team performs V&V activities both on the artifacts modified and on the TLR; they issue a verification & validation report (VVR) based on their findings.
- The SGMR receives the VVR and determines the new status of the modification requests and problem report based on the report.

After describing the above process, the paper gives a model of parts of this process using the process modeling language SLANG. A SLANG specification consists of finite state machines and ER nets, 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 in the Bandinelli paper are: the top-level items in the process, modeled as a finite state machine; the activity entitled *ImplementRIMO*, modeled as an ER net; and the activity entitled *RequalificationOfUpdates*, also modeled as an ER net. Accordingly, we shall illustrate PROMO by showing our implementation of these three process fragments.

7.2.1 Process artifacts

Although the PROMO metaphor determines much of the way in which we map this process into the system, we must decide what types of artifacts we shall create; furthermore, we have to determine the relationships among these artifact types. The artifact types we have used are:

- *Problem report.*
- *Modification request:* Many-to-one relationship with *Problem report*
- *Executable:* One-to-many relationship with *Problem report.* The executable essentially represents the product being modified, and it is possible that there should be a separate object, *Product*, that represents the product. However, for the sake of simplicity we have assumed that a product consists of a single executable, and 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*; many-to-many relationship with *Executable.*
- *Test set:* One-to-one relationship with *Modification request*
- *Text file.*
- *Input file:* Child of *Text file*; many-to-many relationship with *Modification request.*
- *Output file:* Child of *Text file*; 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 modules or editor-specific documents—the above constitue the basic types and their relationships.

7.3 Top-level state machine

The state machine used in modeling this process in SPADE is shown in Figure 5.

7.3.1 Rooms

Since the room structure in PROMO closely resembles a state machine, we begin our modeling of this top-level process by attempting to re-cast 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 performed while an problem 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 6 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*

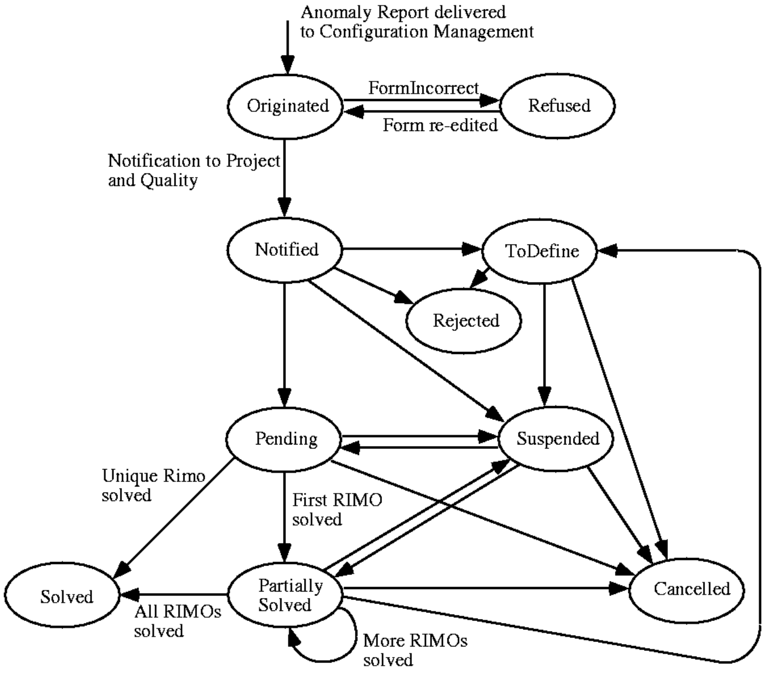

Figure 5: SPADE state-machine model of the top-level problem report process.

PR room, since the task performed on a PR in the *Originated* state is the task of reviewing the PR.

However, the state machine was intended to track the state of the anomaly report. Many states of the anomaly report 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, and 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, and so it was removed; the intention is that an incomplete problem 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 task—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 was trivial. The room resulting from this unification is the *Implement* room.

7.3.2 Exits

Note that while the edges in Figure 6 are directed, this does not mean that each edge corresponds to a single one-way exit within PROMO. Since we wish the players to be able to walk freely among the rooms corresponding to the process, we create for each edge two exits—

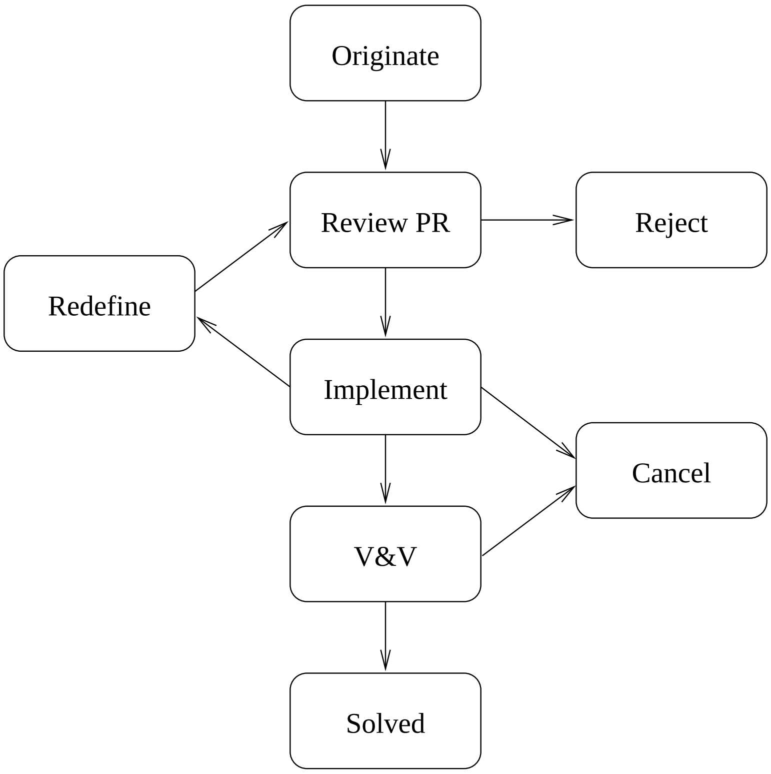

Figure 6: Promo **model of the top-level process.**

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 as follows:

- *Originate* → *Review PR*: The PR must have been checked for correctness of form; a successful checking sets the property `.checked` on the problem report to a true value, so the constraint simply checks this value.
- *Review PR* → *Reject*: No constraint on entering the *Reject* room, but no problem report may return to the *Review PR* room once it has been rejected.
- *Review PR* → *Implement*: The PR must have at least one MR associated with it. That is, the PR's `.modreq` property must be a list containing at least one descendant of the generic MR object.
- *Implement* → *Cancel* and *V&V* → *Cancel*: As with *Reject*, no constraints are placed on entering *Cancel*, but no problem report may return via the return exit.
- *Implement* → *Redefine*: No MRs may pass through this exit, but a PR may pass through in order to allow additional MRs to be attached to it. The edge returning to *Implement* is unconstrained.
- *Implement* → *V&V*: Any MR 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 MR with a test object attached must also have a TLR attached to the test object.
- *V&V* → *Solved*: Any PR passing through the exit must have only completed and verified MRs and must have been approved (by the SGMR).

7.4 Implementation task

The *ImplementRIMO* task consists of an ER net that is used to implement a modification request as shown in Figure 7.

7.4.1 Rooms

Since the model of the task within SPADE is given as an encapsulated ER net, we model this in PROMO by means of a sub-building as shown in Figure 8.

The interaction with the configuration management represented in the SPADE 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. One notable difference between the two models is that while the SPADE model permits the player to bypass the *RequalificationOfUpdates* task if no tests need to be performed, the PROMO model has only one branch to the process flow and so requires all objects to pass through *Requalify*.

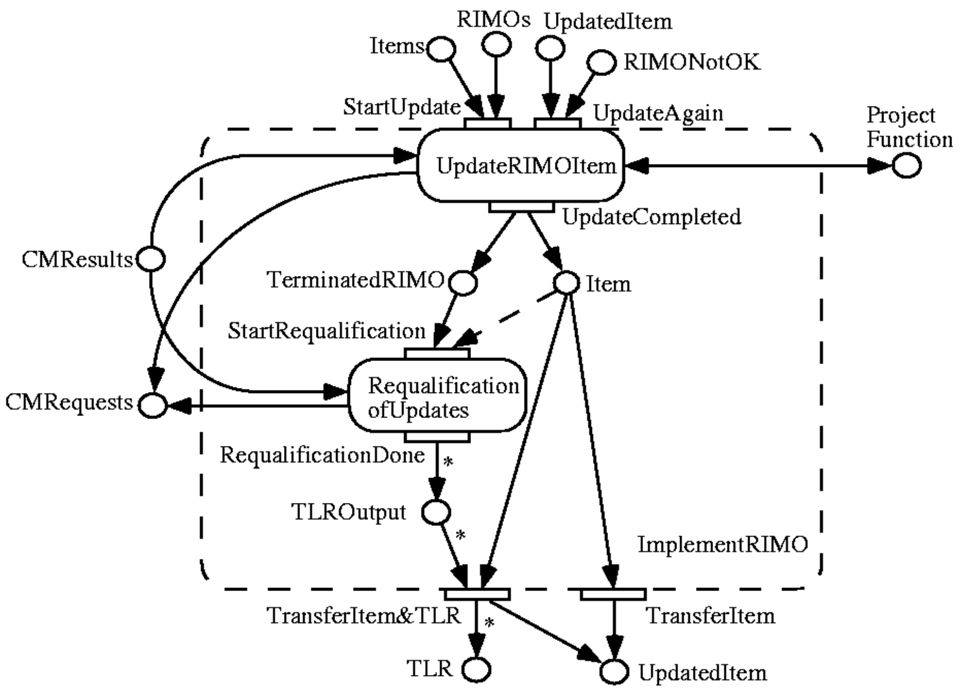

Figure 7: SPADE model of the *ImplementRIMO* task.

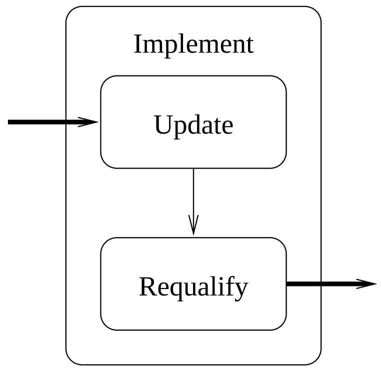

Figure 8: PROMO model of the *Implement* task.

7.4.2 Exits

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 MR be up-to-date with respect to its sources. While this constraint hardly guarantees that the MR has actually been implemented, there is in fact no way of ensuring this; this simply ensures that the product has been successfully built, so that at least the product is in a stable state.

7.5 Requalification task

While the SPADE model of the *RequalificationOfUpdates* is fairly intricate as shown in Figure 9, most of it centers around retrieving the tests and items. In fact, the only real action

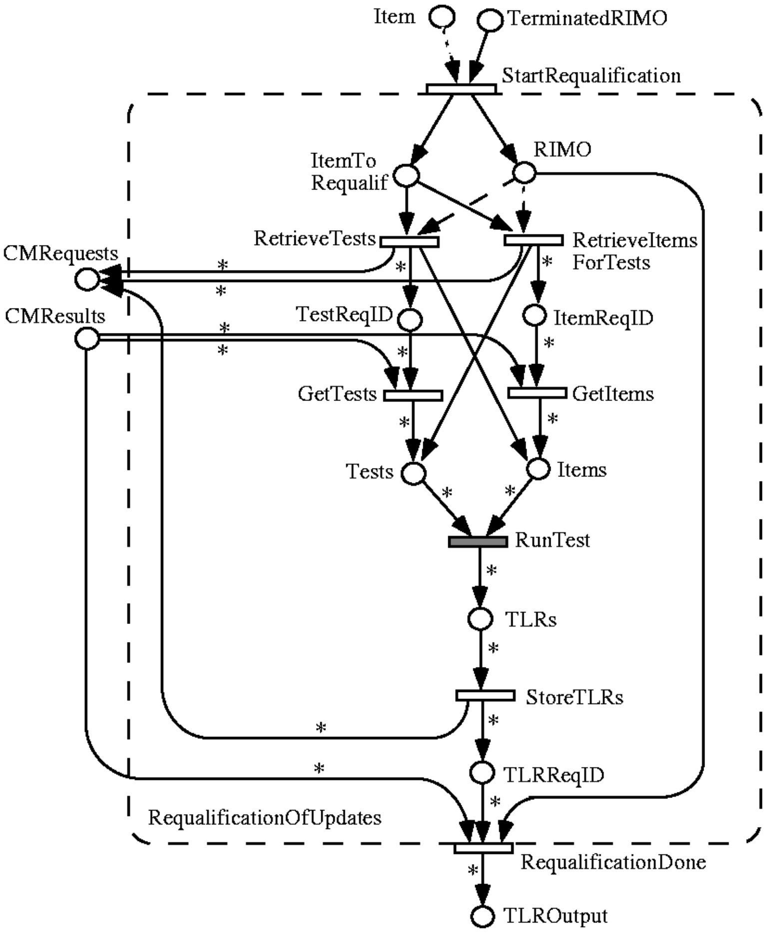

Figure 9: SPADE model of the task *RequalificationOfUpdates*.

or subtask within *RequalificationOfUpdates* is the actual running of tests. In PROMO, therefore, the *Requalify* task consists of only one room.

Chapter 8
Conclusion

PROMO demonstrates the utility of the metaphors provided by virtual environments in modeling and executing software processes. Furthermore, we believe that the use of metaphors in permits a type of collaboration and synergy not seen in many process-centered environments.

8.1 Future work

8.1.1 Comparisons with other metaphors

Although we chose the task-centered metaphor in designing PROMO, many of the other metaphors show promise in their ability to model software process. One avenue of future work, then, concerns the elucidation 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.

8.1.2 Proactivity

PROMO is currently a "passive" system; the player is expected to carry out all necessary actions, and only when attempting to complete a task (exit a room) does the player receive feedback on her or his progress through the process. Accordingly, one potential avenue of research would investigate the possibility of incorporating a greater degree of automation within the process. Other systems [25, 1] do provide this facility, and so using their approaches—or perhaps integrating PROMO at an architectural level with one of these systems—would be an interesting experiment in proactivity within a metaphor.

8.1.3 Integration of PROMO with the environment

As described in chapter 6, PROMO does communicate with the environment outside it in order to invoke actions. However, the communication is limited in various ways. First, the method in which PROMO communicates is fairly primitive; 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, and so PROMO ought to provide better facilities for tool integration and communication.

8.1.4 Constraint language

As mentioned in the discussion of constraints in section 6.2, the manner in which the process modeler is expected to describe exit constraints is rather arcane. Accordingly, the creation of a constraint language to describe these constraints, rather than forcing the modeler to call several functions to create the constraints, would be quite useful.

8.1.5 Additional examples

We have claimed that PROMO is a useful tool for modeling and executing any software process. However, our current work has led to the modeling of only one actual process. Further use of PROMO for additional processes is needed to justify claims of its general utility. Alternately, it is possible that PROMO will be found to be more useful for certain types of processes than for others, and this too is an important result to discover.

REFERENCES

[1] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The architecture of the SPADE-1 process-centered SEE. In *Proc. 3rd European Workshop on Software Process Technology*, Grenoble, France, February 1994.

[2] S. Bandinelli et al. Modeling and Improving an Industrial Software Process. *IEEE Transactions on Software Engineering*, 21(5):440–454, 1995.

[3] Dr. Richard Bartle. Interactive multi-user computer games. Research report commissioned by British Telecom plc., December 1990. `ftp://parcftp.xerox.com/pub/MOO/papers/mudreport.ps.Z`.

[4] Richard Bartle. `Early MUD history`. Article posted to USENET group `rec.games.mud`, 15 November 1990. `http://www.utopia.com/talent/lpb/muddex/bartle.txt`.

[5] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). Proposed as Network Working Group Request for Comments 1738, December 1994. `http://www.w3.org/pub/WWW/Addressing/rfc1738.txt`.

[6] BioMOO. `bioinfo.weizmann.ac.il`, port 8888. The biologists' virtual meeting place. `http://bioinfo.weizmann.ac.il/BioMOO`.

[7] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions): Mechanisms for specifying and describing the format of internet message bodies. Network Working Group Request for Comments 1341, June 1992.

[8] Amy Bruckman. Programming for fun: MUDs as a context for collaborative learning. In *Proceedings of the National Educational Computing Conference*, Boston, MA, June 1994. National Education Computing Association.

[9] Amy Bruckman and Mitchel Resnick. The MediaMOO project: Constructionism and professional community. *Convergence*, 1(1), Spring 1995.

[10] Amy Susan Bruckman. Moose crossing: Creating a learning culture. PhD thesis proposal, December 1994. `ftp://ftp.media.mit.edu/pub/asb/papers/moose-crossing-proposal.ps`.

[11] Lauren P. Burka. The MUDline. Published on the WWW, 1995. `http://www.utopia.com/talent/lpb/muddex/mudline.html`.

[12] Eva-Lise Carlstrom. The communicative implications of a text-only virtual environment, or, Welcome to LambdaMOO! Published on the WWW, 15 May 1992. `ftp://parcftp.xerox.com/pub/MOO/papers/communicative.txt`.

[13] Lynn Cherny. The modal compleixty of speech events in a social MUD. *Electronic Journal of Communication*, 5(4), November 1995.

[14] Lynn Cherny. The situated behavior of MUD back channels. In *Proceedings of the AAAI Spring Symposium*, March 1995.

[15] B. Curtis, M. Kellner, and J. Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[16] Pavel Curtis. Mudding: Social phenomena in text-based virtual realities. Published on the WWW, 3 March 1992. `ftp://parcftp.xerox.com/pub/MOO/papers/DIAC92.ps.Z`.

[17] Pavel Curtis. LambdaCore, 25 December 1995. The "core" database distributed along with LambdaMOO. `ftp://parcftp.xerox.com/pub/MOO/LambdaCore-25Dec95.db.Z`.

[18] Pavel Curtis. LambdaMOO, May 1996. The server code for the multi-user game LambdaMOO, version 1.8.0p5. `ftp://parcftp.xerox.com/pub/MOO/LambdaMOO-1.8.0p5`.

[19] Pavel Curtis. *LambdaMOO Programmer's Manual*. XEROX Palo Alto Research Center, May 1996. For LambdaMOO version 1.8.0p5. `ftp://parcftp.xerox.com/pub/MOO/LambdaMOO-1.8.0p5`.

[20] Pavel Curtis, Michael Dixon, Ron Frederick, and David A. Nichols. The Jupiter audio/video architecture: Secure multimedia in network places. In *Proceedings of the Third Annual ACM International Multimedia Conference and Exposition*, San Francisco, CA, 5–9 November 1995. Association for Computing Machinery.

[21] Pavel Curtis and David Nichols. MUDs grow up: Social virtual reality in the real world. In *Proceedings of the Third International Conference on Cyberspace*. XEROX Palo Alto Research Center, May 1993.

[22] Geraldine Fitzpatrick, Simon Kaplan, and Tim Mansfield. Physical spaces, virtual places and social worlds: A study of work in the virtual. Submitted to the 1996 ACM Conference on Computer Supported Cooperative Work (CSCW '96), 1996. `http://acsl.cs.uiuc.edu/kaplan/Papers/cscw-96-study.ps`.

[23] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[24] Dennis Heimbigner and Leon Osterweil. An argument for the elimination of roles. Position paper for the 9th Int Software Process Workshop, 5 March 1994.

[25] George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in Marvel: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[26] Jay's House MOO (JHM). `jhm.ccs.neu.edu`, port 1709. Research-oriented MOO. `http://jhm.moo.mud.org:7043/`.

[27] Simon M. Kaplan, Geraldine Fitzpatrick, Tim Mansfield, and William J. Tolone. MUDdling through. Submitted to the 1996 ACM Conference on Computer Supported Cooperative Work (CSCW '96), 19 March 1996. `http://acsl.cs.uiuc.edu/kaplan/Papers/cscw-96-mud.ps`.

[28] Ken Keys. TinyFugue, 28 May 1996. Software package used as a "smart client" to MUDs, version 3.5alpha19. `http://glia.biostr.washington.edu/%7Ehawkeye/tf/`.

[29] Larry Masinter and Erik Ostrom. Collaborative information retrieval: Gopher from MOO. In *Proceedings of INET'93*. The Internet Society, June 1993.

[30] MediaMOO. `mediamoo.media.mit.edu`, port 8888, maintained by Amy Bruckman. Private MOO restricted to people doing media research. `http://lcs.www.media.mit.edu/people/asb/MediaMOO/`.

[31] Pueblo. A text-based virtual learning community. `http://pc2.pc.maricopa.edu/`.

[32] Eric S. Raymond. The Jargon File. Published on the WWW, 25 January 1996. `http://www.fwi.uva.nl/%7Emes/jargon/t/Top.html`.

[33] Elizabeth Reid. Cultural formations in text-based virtual realities. Master's thesis, University of Melbourne, January 1994.

[34] Graeme Smith. `hiding .location`. Message sent to mailing list MOO-Cows, message-ID `<Pine.A32.3.91.960707010709.42700B-100000@fn1.freenet.edmonton.ab.ca>`, 7 July 1996.

[35] WWW5 Discussion Forum MOO. `spsyc.ac.nott.uk`, port 5555. MOO used to facilitate two ongoing online workshops: "Virtual Environments and the WWW" and "Artificial Intelligence-based tools to help W3 users.".