Oz : A Decentralized Process Centered Environment

(Thesis Proposal)

Israel Z. Ben-Shaul

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
israel@cs.columbia.edu

TR CUCS-011-93

May 5, 1993

Abstract

This is a proposal for a model and an architecture for decentralized process centered environments, supporting collaboration and concerted efforts among geographically-dispersed teams – each team with its own autonomous process – with emphasis on flexible control over the degree of collaboration and autonomy provided. The focus is on decentralized process *modeling* and on decentralized process *enaction*.

1 Introduction and Motivation

Software Development Environments (SDEs) emerged in an attempt to address the problems associated with developing, maintaining and managing large-scale software projects. One of the main issues in SDE research is how to construct environments that are flexible and extensible, while at the same time integrated [44]. Early SDEs focused on integration, by providing frameworks in which tools and users could communicate and exchange information. However, most SDEs were either knowledge-less with respect to the actual process of development or maintenance, or imposed a hard-wired process.

Process Centered Environments (PCEs) emerged in the mid '80s as an attempt to address mainly the flexibility/extensibility aspect by means of "process modeling". The gist of process modeling is (1) to provide a special purpose programming language for the definition of the software processes (e.g., management, design, coding, testing) and (2) to take advantage of well-understood programming language implementation techniques to execute as much as possible of the software process. Thus, a PCE typically consists of a "process-translator" that compiles the specifications of the process and loads them into the instantiated environment, and a "virtual machine" (or process-engine) that "enacts" the defined process.

Enaction is a widely used term in the PCE community. The purpose of process enaction is to assist and carry out the development process as defined in the modeling language. Kinds of enaction include monitoring, guidance, automation, enforcement, and control, for parts or all of the enacted process. Process enaction differs from conventional program execution in that: (1)the *operators* of the language are tools (e.g., compilers, editors, etc.); (2) the *operands* of the language are product artifacts (e.g., code, documentation, etc.); and (3) human users are inherent part of the process (program), and may actively participate in its enaction (execution).

Process modeling has increasingly attracted attention in the software engineering community, as evidenced by the eighth International Software Process Workshop and the second International Conference on the Software Process. Various PCEs have been constructed as research prototypes and non-commercial systems, and some have been recently released as commercial products. Examples of relatively well-known academic and other research PCEs include Arcadia [26], Common Lisp Framework (CLF) [37], Melmac [14], Merlin [47], Adele [33], and Oikos [1]. Examples of commercial products include Process Weaver [10], and HP SynerVision.

1.1 Why Decentralization ?

Large-scale product development (be it software, or other engineering) typically requires participation of multiple users, often divided in multiple groups, each of which is concerned with a different aspect of the product. For example, a software product may require teams for requirements specifications, design, development, testing, and maintenance. Each team requires its own specific tools, and may possibly require its own private data, management policies, and a set of constraints and workflow, all parts of the process. At the same time the teams need to collaborate in order to develop the product, and as research in Software Engineering (SE) has shown, the interaction among team members accounts for a significant part of the total cost of the product being developed [19]. The degree of team *autonomy* and of the *collaboration* between teams depends on the nature of the product being developed and on the organizational policies (e.g., centralized vs. decentralized management). It is also often the case that multiple independent organizations are collaborating to produce a product, in which case autonomy (and privacy or security) are "hard" constraints that cannot be compromised. Finally, with the advent of high-speed networks and enhanced communication facilities, geographical dispersion is an additional system requirement, particularly among teams (as opposed to within a team). Thus, the underlying motivation for decentralization is to address *scale* and *heterogeneity*.

Decentralization is a very active research area in the database community (Heterogeneous Distributed Database Systems (HDDB)) and in engineering (Concurrent Engineering (CE)). Section 4 briefly discusses some of that work. However, it has been hardly explored in the area of PCEs, mainly because the field is relatively young and the state-of-the-art in PCE technology has not reached this stage yet. But it has been recently acknowledged [34] as one of the main future research directions, and seems to be a natural evolution of PCE technology.

Mapping the problem description to the PCE domain, there is a need to provide support for multiple groups – each with its own *process* and possibly geographically-dispersed – and support for control over process autonomy and collaboration.

What distinguishes research in DEcentralized PCEs (henceforth DEPCEs) from the other mentioned domains, is the fact that heterogeneity, autonomy and collaboration have to be addressed not only in the context of data (HDDB) and human interaction (CE), but also in the context of the *process* operating on the data on behalf of, and with the participation of, human users. Since process represents the variant component of a PCE, it is also impossible to fix the degree of autonomy or collaboration, and so a flexible control on a per-project basis is required. Finally, since processes typically involve rich semantics and encapsulate knowledge about the development process, inter-operation is harder to achieve.

1.2 Marvel 3.1 - The Predecessor Environment

It is important to give an overview of the existing Marvel PCE, both to introduce concepts and terms which will be used throughout the proposal, and to understand what is part of this proposal and what has already been done and is taken as "given" in Oz - the system that will realize the proposed model. (For a detailed description of Marvel and for a list of publications, see [32].)

Marvel's generic kernel is tailored by an *administrator* (or *process engineer*, these terms will be used interchangeably) who provides the *data model*, *process model*, *tool envelopes*, and *coordination model* for a specific organization or project. Users interact with the system through a client that provides user interface.

```
PROTECTED_ENTITY:: superclass ENTITY;
owner: user;
rule's perm_string: string = "rwad rwa*"; end

AFILE :: superclass ARCHIVABLE, RANDOMIZABLE, HISTORY, PROTECTED_ENTITY;
    machines : set_of MACHINE;
    config   : string = "MSL"; end

MINIPROJECT :: superclass BUILT, PROTECTED_ENTITY;
    config       : string;            # state
    options      : string;            # state
    log          : text               # file
    exec         : EXEFILE;           # single composite
    files        : set_of FILE;       # multi  composite
    glb_exe      : link EXEFILE;      # single link
    includes     : set_of link INC;   # multi  link
    afiles       : set_of link AFILE; # multi  link
end
```

Figure 1: Several Classes from C/Marvel

1.2.1 Data Model

The data model defines an object-oriented schema for the product data (the software system under development) and the process data (additional state information used to track the ongoing process). An object in Marvel has a unique identity and a state associated with it. However, it does not contain behavioral "methods". The "methods" are a set of rules, defined separately (see below). Class definition supports multiple-inheritance in the conventional manner, i.e., the class lattice is a directed acyclic graph, and subclasses denote specialization of their superclasses, and overriding of methods defined on those classes.

Marvel supports four types of attributes: state, file, composite, and reference. The first two attributes contain the contents of objects whereas the last two attributes are used to denote relationships to other objects. *State* attributes are used mainly for process data (although they can be used to hold product data as well), and can be formed from a set of primitive types such as integer, string, enumerated, etc. *File* attributes can be either text or binary, and are used to maintain product data which is held in files. File attributes are implemented as a file-system path pointing to the file in a "hidden" file system. Thus, users do not have direct access to the file system, which is abstracted in the objectbase. *Composite* attributes are used to denote an "is-part-of" relationship between objects, to form the *composition hierarchy*. Finally, *reference links* (or simply *links*) allow any arbitrary semantic relationship between two objects.

Figure 1 shows some representative classes from C/Marvel, the process for developing Marvel itself.

1.2.2 Process Modeling

The process is specified in a rule-based *process modeling language*, called the MARVEL Strategy Language (MSL). Each process step is encapsulated in a *rule* with a name and typed formal parameters. Each rule is composed of a *condition*, an *activity*, and a set of *effects*; each is an optional component.

The condition has two main parts: local *bindings* which are used to gather related objects (e.g., included ".h" files when compiling a ".c" file) by *querying* the objectbase, and a *property-list* that must evaluate to true (in which case the condition as a whole is said to be *satisfied*) prior to invocation of the activity. In addition to a quantified variable and its class, each rule binding may optionally include a complex clause (nested conjunctions and disjunctions) of *predicates*. *Structural* predicates navigate the objectbase to obtain ancestors or descendants of specified types, containers or members of aggregate attributes, and objects linked to or from other objects. *Associative* predicates query the objectbase to obtain those objects satisfying a relation (equality, inequality, less than, etc.) specified between attributes of two objects, or between an object and a literal. The binding phase completes with a set of variables (henceforth *derived parameters*) which are bound to sets of objects that match the query. A property list is a complex logical clause of associative predicates over the actual and derived parameters.

An *activity* involves invocation of an external tool to operate on the product data encapsulated within the bound objects. In MARVEL, tools are encapsulated via an *envelope* mechanism written in a Shell Extended Language (SEL) [20]. If there is no activity, then by definition there can be only one effect. If there is an activity, then in general the invoked tool may have several possible results corresponding one-to-one with the given effects. A non-empty rule activity specifies an envelope and its input and output arguments, which may be literals, status attributes and/or (sets of) file attributes. In addition to output arguments, each envelope returns a code that uniquely selects one of the specified effects.

Finally, a rule's *effects* are mutually exclusive in the sense that only one effect can be asserted at any rule invocation, as determined by the return code from the activity. Each effect consists of a conjunction of predicates. An effect predicate assigns the specified value to an attribute, or applies any of MARVEL's built-in add, delete, move, copy, rename, link and unlink operations.

A representative rule is shown in Figure 2. This `archive` rule accepts one parameter of class `MODULE`. It has six composite binding expressions, a property-list expression, an activity that takes three arguments (each of which can be possibly bound to a set of objects), and two effects.

Rules are interrelated by means of matchings between an effect of one rule and a condition of another rule. Thus, operations between steps in a process can be implicitly formed by matching predicates in the condition of one rule and an effect of another rule, and the enaction engine enforces and/or automates the sequencing. However, the process is not in any sense limited to a deterministic sequence of steps. (See [28] for discussion of the specification of alternatives, iteration and synchronization through the conditions and effects of rules.)

```
archive [?m:MODULE]:

###  bindings
(and (forall CFILE ?c suchthat (and no_chain (member [?m.cfiles ?c])
                                    (or (?c.config = ?m.config)
                                        (?c.possible_config = ""))))
     (forall YFILE ?y suchthat (and no_chain (member [?m.yfiles ?y])
                                    (or (?y.config = ?m.config)
                                        (?y.possible_config = ""))))
     (forall LFILE ?x suchthat (and no_chain (member [?m.lfiles ?x])
                                    (or (?x.config = ?m.config)
                                        (?x.possible_config = ""))))
     (forall MODULE ?child suchthat no_chain (member [?m.modules ?child]))
     (exists AFILE ?a suchthat (and no_chain (linkto [?m.afiless ?a])
                                             (?a.config = ?m.config)))
     (forall MACHINE ?mc suchthat no_chain (member [?a.machines ?mc])))
    :

### property-list

    (and no_chain (?m.archive_status = NotArchived)
         no_forward (?m.compile_status = Compiled)
         no_forward (?c.archive_status = Archived)
         no_forward (?y.archive_status = Archived)
         no_forward (?x.archive_status = Archived)
         no_forward (?child.archive_status = Archived))

### activity

    { ARCHIVER mass_update ?m ?.afile ?a.history }

### effects

# effect 0
    (and (?m.archive_status = Archived)
         no_chain (?mc.time_stamp = CurrentTime)
         [?a.archive_status = Archived]);
# effect 1
    no_chain (?m.archive_status = NotArchived);
```

Figure 2: Example Rule from C/Marvel

1.2.3 Process Enaction

Enaction is provided in MARVEL by *chaining*. Forward and backward chaining over the rules enforces consistency in the objectbase and automates tool invocations. Enforcement and automation are the two forms of enaction supported in MARVEL.

MARVEL's process enaction is user-driven. When a user enters a command, the environment selects the rule with the same name and "closest" signature to the provided actual parameters considering multiple-inheritance [4]. If the condition of the selected rule is not satisfied, backward chaining is attempted. If the condition is already satisfied or becomes satisfied during backward chaining, the activity is initiated. After the activity has completed, the appropriate effect is asserted. This triggers forward chaining to any rules whose conditions become satisfied by this assertion. The asserted effects of these rules may in turn satisfy the conditions of other rules, and so on. Eventually, no further conditions become satisfied and forward chaining terminates. MARVEL then waits for the next user command. Due to the event-driven nature of the enaction model, the actual parameter-selection for rules invoked through chaining is done by an algorithm that "inverts" the logic of the bindings of the chained-to rules [24].

Predicates in effects of rules are each annotated as either *consistency* or *automation*. By definition, all forward chaining from a consistency predicate in an asserted effect to rules with satisfied conditions and empty activities is mandatory. In contrast, forward chaining from an automation predicate or into any rule with a non-empty activity is optional, and can be "turned off" wholesale by a user or explicitly restricted through `no_forward`, `no_backward` or `no_chain` directives on individual automation predicates. Backward chaining is meaningful only into (unrestricted) automation predicates, since any consistency predicates that could satisfy would already have done so during consistency forward chaining. It is important to understand that only automation *chaining* is optional; users are still obliged to follow some legal process step sequence implied by the conditions and effects of rules, whether through manual selection of commands or automation chaining. For more on the consistency model in MARVEL see [3].

1.2.4 Synchronization and Coordination Modeling

In addition to process modeling, MARVEL provides capabilities to model coordination among team members, by means of a Coordination Rule Language (CRL) [2] that defines how to resolve concurrency conflicts in accessing data. In addition to the programmable interface for conflict resolution, the actual conflict detection scheme is modifiable by project-specific external tables which are loaded at environment start up.

1.2.5 MARVEL 3.1 Architecture

MARVEL's architecture [7] is illustrated in figure 3.

The architecture follows the client-server model, where the server is centralized and manages

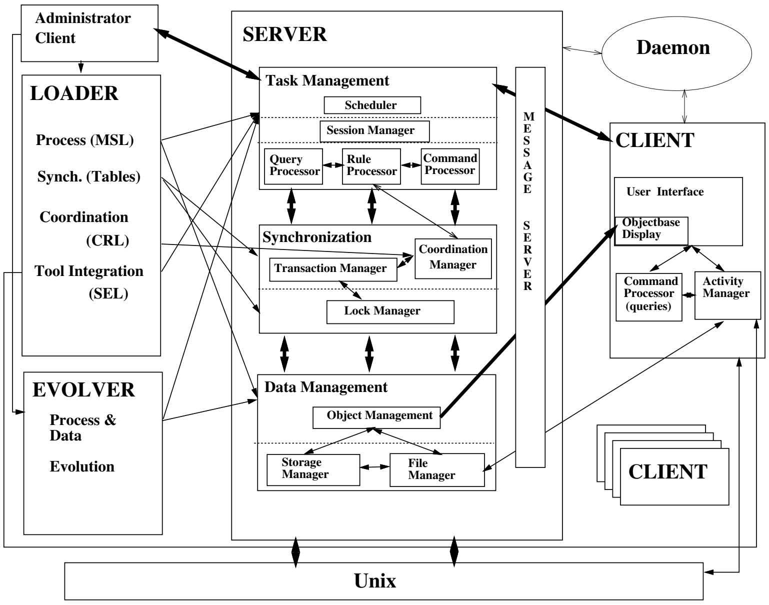

Figure 3: Marvel 3.1 Architecture

the data, process, and synchronization, and clients manage the user interface (including objectbase browsing) and activity invocation - by forking operating system processes to execute external tools using the envelope wrapping mechanism. Each client can support multiple threads of control [1], and clients can be distributed across machines, but the server and all of its clients must reside in the same site and share the file system. Every user command (besides a small number of commands that are handled solely at the client) is transferred to the server, which validates the request and possibly backward-chains to other rules, manages the access to objects (including concurrency control), sends the activity to be carried out to the client, and switches context to service new requests. (The actual scheduling algorithm is a simple FIFO queue.) Upon completion of an activity, the server attempts to forward chain to other rules, which in turn may lead to more interactions with the client to execute more activities, and so forth. Synchronization management is highly flexible in Marvel, mainly due to the separation made between conflict detection and conflict resolution, and the architecture supports a wide variety of synchronization policies. Finally, additional components of the system include: the Loader which translates the process specifications and loads them into the server; the Evolver, a process and data evolution tool; and a daemon responsible for activating a server on a given environment upon client request when the environment is inactive (the server normally shuts down when no clients are logged-in to it). The rationale behind the Marvel architecture can be found in [7].

This architecture is adequate for a small to medium number of people interacting with the server through clients in the same local-area network (we have experienced using Marvel successfully with up to 20 concurrent clients and 8 users). But as the number of (simultaneous) clients grow, the server might become a bottleneck. Additionally, the architecture

[1] this feature is available only in the XView interface, the most advanced of three kinds of clients supported in Marvel.

dictates that all users must follow essentially the same process, or at best allow some deviations but still from a single process[2]. And finally, MARVEL requires all entities to reside in the same site. Thus, MARVEL lacks the necessary architectural support for scale and heterogeneity.

1.2.6 Relationship of MARVEL to OZ

Version 3.1, the latest and final version of MARVEL, was released for licensing in March 1993, concluding approximately six years of research and involving various students and staff which contributed to the design and implementation (about 150,000 lines of code) of the system. OZ is conceived as the "next-generation" PCE, which extends MARVEL's functionality significantly in several directions. However, OZ is not planned to be a "designed-from-scratch" system, and some of the fundamental aspects (and code) of MARVEL will be carried over to the OZ project. Wherever necessary, the "inherited" aspects will be denoted in the proposal.

1.3 Terminology

The following terms will be frequently used throughout the thesis.

Site - A set of one or more computers (hosts) that physically, as well as logically, comprise a computing unit. This means that they are connected by a local area network and share a file system (e.g., NFS) under which the project is developed. In addition, all hosts on the site can be reached from remote hosts through a designated address that receives incoming requests and forwards them as necessary. An example of such site is the Computer Science Department at CU. A site is functionally equivalent to a *domain* in Internet terminology.

Environment - This term may be context sensitive. Unless otherwise mentioned, this term refers to an instance of a process with a populated objectbase. A *global* environment is a collection of *local* or *sub*-environments (henceforth SubEnv), each of which resides on a host in the same or in different sites. A SubEnv is an autonomous entity with a single process and a single schema, that can possibly share its data and its process with other SubEnvs in the same global environment.

Similarly, a global objectbase refers to the collection of all sub-objectbases of an environment, one per sub-environment.

An *active* SubEnv is a SubEnv on which DEPCE programs are currently running. A SubEnv can be in an *inactive* state, i.e., it is not required nor assumed that it is continuously running. However, it should be able to be activated upon a (local or remote) request for service, which requires an automatic activation mechanism.

[2] The actual implementation does not support deviations at all. See [5] for a design of such a mechanism for MARVEL

2 The Problems

The goals of this research are to devise a model and an architecture for decentralized PCEs that support collaboration and concerted efforts among teams, while retaining their autonomy.

This thesis addresses three major problems:

1. Decentralized, yet collaborative process *modeling*.
2. Decentralized process *enaction*.
3. Decentralization over geographically dispersed computational and storage resources.

The first two problems are complementary to each other, while the third is an extension of the first two. That is, a distinction is made between *logical* and *physical* decentralization. The former refers to multiple autonomous processes that differ from each other, and are enacted separately, but are physically closely connected (e.g., they reside on the same local area network), whereas the latter adds the dimension of physical separation of processes with arbitrary distance (bandwidth) between them. Physical decentralization obviously bears implications on the architecture, but is likely to also affect the general model. Nevertheless, the problem of having different autonomous yet cooperative processes can be examined independently of the additional constraints and problems associated with having those processes enacted in arbitrary physical separation. Thus, the thesis breaks down the general problem into two phases, starting with logical decentralization that supports some, but not all of the requirements for physical distribution, followed by a general model and architecture for geographical decentralization.

The problems described above are addressed at two levels of abstraction: first, a general decentralized *model* is developed, followed by an *architecture* that supports proper realization of the model. The distinction between the "model" and the "architecture" will be made clear in subsequent chapters. For now it suffices to say that the model provides a conceptual framework for a DEPCE, and the architecture provides the necessary mechanisms to realize the model in the best way in terms of performance, as well as flexibility. The focus of this proposal is on the model, with preliminary discussions on the architecture. The implementation of the architecture is intended to be part of the thesis. Figure 4 summarizes the approach taken here for decomposing the general problem, where the degree of shading indicates the amount of work that is going to be made in each of the subjects represented by the squares.

2.1 Requirements

This section formulates the problems into a set of requirements from a DEPCE, and presents more specific problems associated with achieving those requirements.

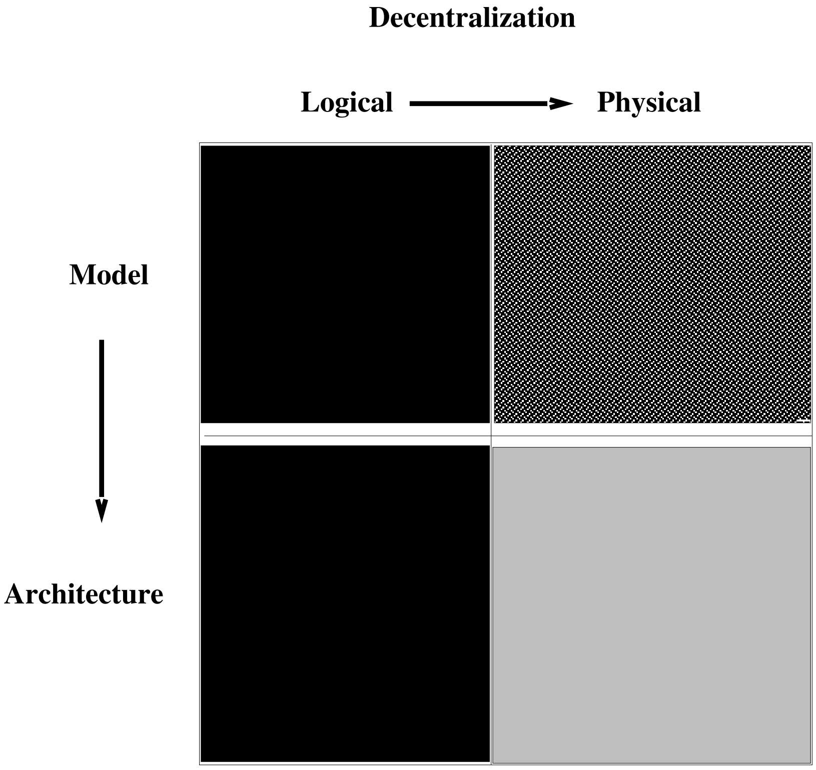

Figure 4: Problem Decomposition

A central underlying requirement is to preserve the core requirements of PCE in general, namely *flexibility*, *extensibility*, and *integration*. Flexibility refers to the ability to select different policies and tailor the behavior of the system to the requirements of a specific project. Extensibility refers to the possibility to augment the system with additional functionality. Integration refers to the ability to integrate existing tools and methodologies into a uniform, coherent environment. The intent is to extend those principles from a PCE to a DEPCE. In particular, decentralized process modeling, communication protocols, and environment configuration management should all be part of the process(es) and not hard-wired – thereby making them tailorable and flexible – while at the same time keep the environment integrated.

Another general requirement is that as far as local work is concerned, a DEPCE should provide the same capabilities and same support as a PCE does. This section addresses mainly the additional requirements from a DEPCE that are not PCE-specific, and extensions to existing PCE requirements. ([7] discusses general requirements for a PCE).

Finally, a fundamental requirement that is orthogonal to the whole concept of decentralization is *componentization* of the system into independent, replaceable components. Componentization, among other things, is important to support architectural design autonomy, where SubEnvs can be built with different components (e.g., different OMS, or different transaction management). Preliminary work on componentization has been done by the author and others in the MARVEL project (see [7]). However, componentization (as well as architectural autonomy) is in general outside the scope of this thesis as a research topic, but is one of the main goals of the Oz project, and will be addressed as appropriate in the thesis.

The following is a list of specific requirements that a DEPCE should fulfill:

1. *Process Autonomy* - Each local environment should control autonomously its process and its data while allowing remote sites to share resources, under restrictions that are solely determined by the local environment. Thus, no centralized control/management should be imposed.

2. *Data Sharing and Visibility* - a DEPCE should provide non-transparent, but nevertheless efficient and highly available data access capabilities. In addition, it has to support broad data access mechanisms, to support various modes of access. In particular, sites should be able to access data residing in remote sites in varying degrees of granularity.

   In order to achieve this requirement, the system should provide *high visibility*. Ideally, a user should be able to browse and view remote objects (provided that there is no access-control restriction on those objects) regardless of their physical distance from the user's machine. Moreover, since PCEs often support complex and highly structured data models, it is especially desirable to be able to display graphically the types of objects and the relationships among them. This represents a challenge both in user interface design, and in the communication protocols that are responsible for updating the user's view of the objectbase(s). Notice that unlike conventional distributed systems, data transparency is intentionally not supported, which means that the user should have means to identify each piece of data with its origin. Moreover, he/she should be able to "turn off" and "turn on" visibility for parts of the remote data that are of interest to him/her. The rationale for non-transparency is given in section 3.1.

3. *Process Collaboration and Interoperability* - This is a major requirement. The main goal is to allow multiple groups of users, each group with its own process and schema, to collaborate with each other. Collaboration in this context means not only the ability to exchange information (*loose* collaboration), but also to perform operations that affect both the state of multiple process(es) and the state of the data "owned" by the processes (*tight* collaboration). Related problems are: how to ensure that inter-process collaboration does not violate the consistency of the data and the process both within and among sub-environments ? (this problem will be referred to as the *Inter-Process Consistency* problem). Another issue is what should be the association between the data and the process, given that they are not identical in different SubEnvs ?

   Interoperability is in general a hard problem, but is even harder in this case, due to the heavy semantics associated with a "process". In this thesis, interoperability is addressed only in a restricted form by assuming multiple processes but the same formalism to define the processes, and multiple schemas but the same data modeling capabilities. Nevertheless, as will be explained below, it can be extended to cover the more general case.

4. *Communication Modeling* - A DEPCE should support a range of geographical distances and different bandwidths between teams and within a team, ranging from local-area

networks, to intermediate ranges (i.e., city model) to long ranges (i.e., wide-area networks). This implies the necessity of flexible communication protocols between the various entities, according to a defined distance/bandwidth metric. In addition, it should support the notion of *logical distance* between sites that represents the relationships between communicating entities. For example, two sites that are in an "integration" phase may want to communicate closely for a period of time, although they may be physically distant. Thus, a metric that combines both logical and physical distance should be developed[3].

The notion of distance between sites adds a dimension to the general problem, and is a parameter in all the above requirements. It affects the data access mechanism since any method should take into account the cost of transferring data between sites as a parameter that affects the decision. Similarly, it affects process sharing and synchronization.

One special form of communication between SubEnvs is a "connectionless" operation, where the entities are usually disconnected from each other, but periodically connect to exchange information and update their state. Disconnected operation is mostly out of scope for this thesis as an normal operational option. It will only be discussed as an exception in the context of failure management. A restricted case, where a client is disconnected from its local server (as opposed to servers disconnected from each other), is being investigated by Skopp[4].

5. *Dynamic Reconfiguration* - In order to retain decentralization, a DEPCE should have the capability to dynamically add, delete, and move inactive SubEnvs from different sites in a global environment without disrupting the operation of the currently active SubEnvs, and notify inactive as well as active SubEnvs of the configuration changes made.

   A related issue is to enable a SubEnv with a pre-existing process to "join" a global environment with other pre-existing process(es), with minimal (re)configuration overhead. This requirement is important when two or more organizations with established processes need to collaborate for a limited time on a shared project.

6. *Decentralized query mechanism* - Query processing in a decentralized system must consider two aspects: (1) optimization of multi-site query evaluation that minimizes the inter-site communication overhead; and (2) expressiveness of the process modeling language and/or the process-engine to support "scoped" queries that apply to specified subsets of sites, as opposed to single global or local modes.

   The focus here is on query mechanism for Object Oriented Data Bases (OODB). While the first aspect is valid for distributed queries in general, the second is especially important in decentralized systems, particularly when the query language supports associative queries or cross-objectbase navigation. Query optimization per se is outside the scope of this thesis, but a practical mechanism will be part of the architecture.

[3]This topic will be mostly developed in the thesis itself and will not be addressed in the proposal.
[4]The exact notion of clients and servers is not important at this moment.

7. *Synchronization Mechanisms* - PCEs in general require flexibility of selection and application of concurrency control (CC) policies [7]. A DEPCE adds the dimension of remote vs. local access. This complicates the synchronization mechanism in that: (1)CC techniques that are adequate for centralized systems are not adequate here due to data distribution; (2) the fact that there is no centralized authority requires some form of "conversation" between involved sites in case of a conflict that spans multiple sites; (3) having different processes at different sites might impact CC, particularly when the process consistency differs, and when semantics-based CC policies are used; (4) if prefetching of data across sites is supported, this necessarily interferes with the synchronization mechanism.

8. *Coordination Modeling* - In order to support a wide range of synchronization mechanisms - including support for long and interactive operations, tight collaboration, and semantics-based concurrency control - there has to be a distinction between the synchronization mechanisms provided by the system and coordination modeling capabilities that set policies using those mechanisms. A coordination modeling language in the context of a PCE was proposed by Barghouti [2]. However, just as the synchronization mechanisms have to be extended for DEPCEs, the coordination modeling language should be able to express (the degree of) remoteness, replication, etc. Another interesting problem is to support multiple synchronization policies at different sites, while still providing consistency. This problem is also being investigated in the context of heterogeneous data bases [40].

9. *Failure Recovery* - Like synchronization, failure recovery in decentralized environments introduces new problems, such as handling of new cases of failures (e.g., network failures), and recovery of a decentralized process state.

   Synchronization and recovery in DEPCEs are mostly out of the scope of this thesis, and are being investigated by Heineman.

10. *Meta Process* - This is somewhat a tangential problem, but nonetheless an important one. The issue is, what is a good process that supports development of DEPCEs ? What should be provided in that process beyond a PCE development process ? Note that the problem here is external to the DEPCE[5]. It is about devising a process using an existing PCE (given the lack of an existing DEPCE) for developing a DEPCE. This can be generalized to process support for development of arbitrary distributed and decentralized systems. For example, an obvious problem is to provide a testing facility for communication between entities in the system being developed, using a centralized PCE. Another requirement is to support dynamic reconfiguration of environments for testing and development purposes. As a prototype of a DEPCE will be built (using the Marvel PCE), the author (together with Z. Tong) hopes to have some solutions to that problem as a by-product of the general problem of the DEPCE architecture.

The table below associates each requirement listed above with one or more of the core

[5]Unless the process supports *reflection*, in which case the solution may be part of the internal DEPCE architecture.

problems presented in section 2, where "strong, "weak, and "none" specify the degree of association.

	Modeling	*Enaction*	*Wide Decentralization*
AUTONOMY	**strong**	**strong**	**strong**
DATA SHARING	**strong**	**strong**	**strong**
INTEROPERABILITY	**strong**	**strong**	**weak**
COMMUNICATION	**weak**	**strong**	**strong**
RECONFIGURATION	**none**	**strong**	**weak**
QUERY	**strong**	**strong**	**none**
SYNCHRONIZATION	**none**	**strong**	**weak**
COORDINATION	**strong**	**weak**	**weak**
FAILURE	**none**	**strong**	**weak**
META-PROCESS	**strong**	**none**	**none**

The focus of this thesis is on fulfilling requirements 1 – 5 and addressing only to some extent requirements 6 – 10.

An underlying theme in the requirements is to provide both autonomy and collaboration, which are conflicting goals. A DEPCE should be carefully designed so that both aspects are adequately provided. Moreover, it should support a continuous range along the autonomy/collaboration spectrum, and provide the capability to control the degree of autonomy/collaboration on a per SubEnv as well as global environment basis, and not by a hard-wired policy.

2.2 A Motivating Example

The following example of a task in a decentralized PCE illustrates the problems that should be solved in order to properly support such tasks. The author is not aware of any existing PCE, either a product or a research prototype, that is capable of handling this example in a truly decentralized manner.

Assume there are three development teams in three SubEnvs **SE1**, **SE2**, and **SE3**, that are developing three disjoint modules of a system **S**, labeled as **M1**, **M2**, and **M3**, respectively (see figure 5). The teams reside in different geographical areas, and use their own process for developing their modules, along with their own set of tools (which might not be identical) used in the processes. Each module can be developed and unit-tested independently, but there is a shared library **L** that is accessible to all groups. The following procedure describes the operations that have to be made when **L** has to be modified: 1) the change has to be pre-approved (or reviewed) by all sites; 2) the change is actually made, producing **L'**; 3) a unit-test of all modules together with **L'** is performed; 4) an integration test of all modules combined is performed; 5) an acceptance (or alpha) test of the new system is done. For simplicity, only the "successful" path (i.e., assuming that all the tests passed successfully) is described.

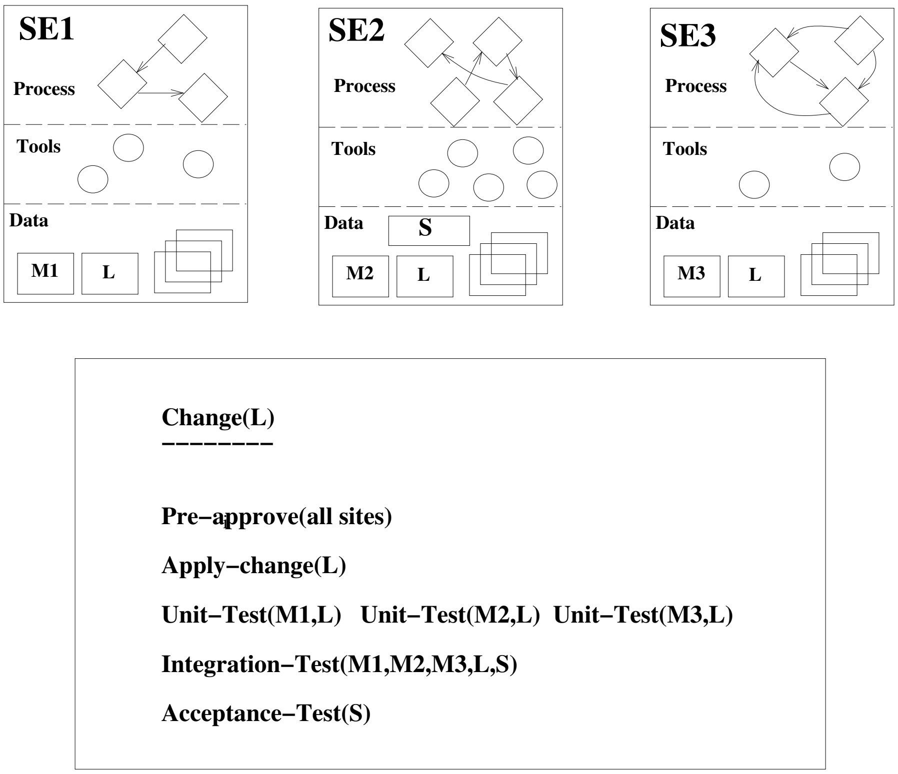

Figure 5: A Motivating Example

At the modeling level, there should be a conceptual framework that enables interoperability of processes in terms of interactions and information exchange among them, and means to specify them on a per-SubEnv basis. At the enaction level, a DEPCE should support the execution of a decentralized task in a decentralized manner. At the architectural level, there must exist a framework that is capable of providing mechanisms for consistent and reliable access to shared data, communication protocols and capabilities for accessing remote data, and a decentralized execution engine that *performs* well.

For this particular example, a *wrong* and in some cases impossible solution would be to collect all the necessary objects from the remote sites to the initiating site, and then perform all the process steps (and all the implications of them) as defined in the local process, on all modules. This approach, besides its obvious performance limitations, is a clear violation of autonomy, since each site has its own process of doing its "local" unit-testing of its module, which may not even be known to the invoking site. Another possible problem might be that some of the tools do not exist in all SubEnvs (e.g., due to licensing that binds a tool to specific site or host), and other tools can be executed only in specific SubEnvs (e.g., special-purpose machines, or architectures), which necessarily binds the execution of a process-step to a specific SubEnv.

Therefore, a more decentralized solution is required, that will enable handling of such tasks in a manner that retains the process autonomy while still provides for collaboration and interaction among the processes. In addition, the task should be optimized towards maximizing local execution of subtasks. The model developed in this thesis is intended to support such tasks in a manner that they are both feasible and effective.

3 Proposal

3.1 Focus of The Research

It is important to lay out at the outset what is the focus of this research, so as to limit the issues that are addressed, because the topic is broad and can easily diverge to many areas of research such as distributed and/or federated databases, concurrency control, process modeling, federated environments, and concurrent engineering, to name a few.

The following is a set of assumptions that try to keep the research within reasonable scope. Some of the topics which are "scoped out" will still need to be addressed to some extent in the implementation, but are not considered as part of this research.

3.1.1 Decentralization vs. Distribution vs. Federation

Broadly speaking, a distributed system is defined as one that provides a single logical view to its applications, but is physically distributed into multiple computing units, most often distributed across machines of a single site [12]. That is, a distributed system transparently shields the distribution from its applications. In contrast, a decentralized system is comprised of relatively *independent* subsystems with some degree of correlation between them. Thus, since each SubEnv in a DEPCE is logically, as well as physically, a separate entity, transparency is intentionally not supported because it would violate the autonomy of the SubEnv. Another important reason for non-transparency is that SubEnvs can be arbitrarily apart, and the difference between the cost of accessing a local object vs. the cost of accessing a possibly very "far" remote object can be arbitrarily large, in which case the accessor, whether a human user or a system module in any level, should be aware of the distance of the object to be accessed.

On the other hand, decentralization in this context differs from "federation" in that it does not support heterogeneity of formalisms and models. Instead, it supports interoperability only between multiple different instances defined in the same formalism. One approach to providing federation is by translating multiple formalisms and models into a low-level uniform formalism and providing constructs to analyze and execute the system using that formalism. This is further described in 3.1.2.

Observing the evolution and scale up of large systems, the natural order tends to be: (1) centralized control, (2) distributed control, (3) decentralized control, (4) federated control (the best representative of this kind of evolution is the database field). Therefore, one may ask, why skip over step 2 and jump directly into step 3, when the problems of step 2 are not resolved yet ? The answer is, that if transparency shields users and applications from knowing where the data is, and retains a uniform view of the data and the process, then the main problem becomes to provide this transparency. While the significance and relevance of this problem is not discounted (especially when dealing with distributed object-oriented databases), it is essentially a database research problem, perhaps with some process flavor. From the PCE research aspect, it is much more interesting to look at loosely coupled and autonomous systems that allow for different processes to coexist. Furthermore, environment distribution is a form of "vertical" scale-up, in that it allows for more users to work, but under the same process, and within some bounded physical distance (typically, but not necessarily, a local-area network). This thesis explores "horizontal" scale up, where the number of users per site may not grow much, but there is no bound on the number of groups or on their distance, and they are allowed to revise their processes independently (albeit with some restrictions that will enable shareability between sites/teams/processes). Note that PCEs add more constraints than pure databases constraints, because they contain more semantics that have to be transparent (similarly, object-oriented databases are harder to distribute than relational databases).

3.1.2 Rule-Based Paradigm

This research restricts itself, at least in its initial phases, to consider decentralization of PCEs that are defined and enacted by a rule-based formalism in the style of Marvel. Since the rule-based approach is in general the most dominant process formalism in PCE research [13], this assumption does not seem too restrictive. In addition, it is hoped that the results of this research will be applicable to a wider range of process formalisms. One approach to supporting interoperability among various formalisms (see [36]) is to use the rule-based language as an "assembly" process language, and translate into it other formalisms such as procedural [42] or petri-net [14]. (An experiment of this approach is described in [28], and is further studied by Popovich.) This approach resembles the approach taken in some heterogeneous databases [31], where it is assumed that there is a translation from the various schemas and query formalisms into a common formalism which enables interoperability. However, it is still not clear yet to what extent the common formalism has to be extended in order to support a wide range of process formalisms.

3.1.3 Shared sub-schema

As mentioned above, this research assumes a single data model, which distinguishes further this work from being related to federated systems and databases. Specifically, it assumes an object-oriented data model in the flavor of Marvel 3.1. As for the actual schema, some but not all of the schema must be identical. That is, a shared sub-schema is required for process interoperability. On the other hand, no single-schema system (like Orion-2 [30]) is assumed, and schemas of different sub-environments can be arbitrarily different from each other. The details are discussed in 3.2.1.

3.1.4 Multiple Processes

This is a key characteristic in the design of the architecture. I assume essentially different processes operating on the schemas, each sub-environment with its own process. However, like with data-modeling, the process formalism and translation mechanism are assumed to be the same.

3.1.5 Geographical Distance

Within a SubEnv, the model assumes that all services and interfaces to human users reside within a single site[6].

The general model places no restrictions on the distance and/or bandwidth between SubEnvs. Two SubEnvs can co-exist at the same site or be arbitrarily physically apart, and with arbitrary and variable bandwidth between them. Thus, two main sub-requirements are imposed here:

1. No sharing of resources such as processors or file system.
2. Addressing the "variable bandwidth" issue.

However, as mentioned above, the initial model (and implementation) will focus on logical decentralization. As such, the "no-sharing" aspect will be addressed but the "variable bandwidth" aspect will be deferred. SubEnvs would still be able to communicate with arbitrary geographical distance between them but with no special modeling and optimization with the "distance" metric as an affecting parameter. It is hoped that this issue will be fully explored and modeled in the second phase of the thesis.

3.2 The Decentralized Model

The abstract model is depicted in figure 6. Essentially, a global environment is viewed as a collection of one or more loosely coupled SubEnvs, each of which resides at a site, and maintains a disjoint persistent objectbase and a corresponding process that manipulates it. A SubEnv is accessed directly by "local" agents, and indirectly by remote agents, where agents can be either programs or human users. A Connection Server is available on each host to (re)establish connections (see section 3.3).

Collaboration and autonomy in a global environment are dealt with at three levels of abstraction: *process*, *synchronization*, and *data*. A fourth level, the *global process*, is realized by the support for each *local process* by the local environments. That is, to exploit decentralization, a global process is nothing more then a set of local processes, and there is no explicit support for modeling of a global process, synchronization, or data.

[6]This restriction may be removed in the future.

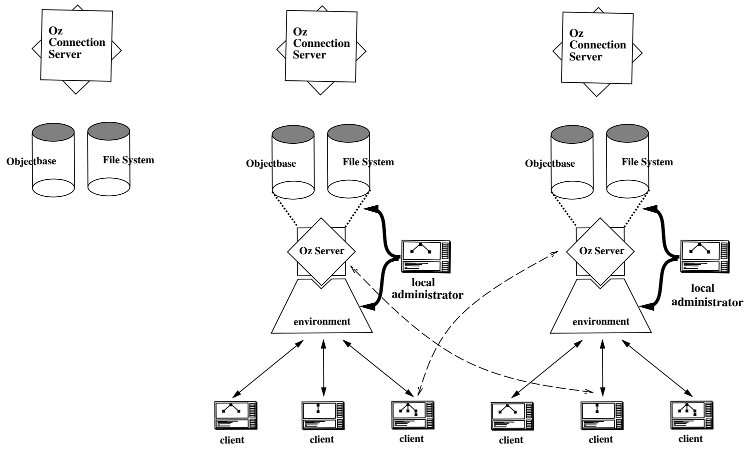

Figure 6: Oz Decentralized Environment

As already mentioned, synchronization is mostly outside the scope of this thesis, so only data and process modeling are discussed here.

3.2.1 Decentralized Data Modeling

Decentralized Schema

The basic data model is similar to the Marvel data model, as explained in section 1, namely, object-oriented data modeling supporting classification, multiple-inheritance, and four kinds of typed attributes: state, file, composite links, and reference links.

The main requirement on the data model is to provide support for private and shared subsets which are manipulated by multiple differing processes. There are several alternatives by which this requirement can be met to a certain degree. Before presenting them, an important distinction is made here between *product* and *process* data. The former represents the actual data elements under development (i.e., source files, object files, design documents, etc.), while the latter represents the data used by the PCE to manage the project. Examples of process data for a source file include its version, compilation status, reservation status, etc. An object can contain pure product data, pure process data, or a mixture of product and process data. The instantiated objectbase is used to maintain uniformly both the product and the process information as objects, either in separation or in mixed fashion.

One approach would be to require a single-schema system, i.e., all sub-environments must have identical schema. While this approach may still allow processes to be tailored, the degree of autonomy is greatly reduced. Since process state is maintained in the objectbase, requiring identical schema for all process-data across SubEnvs would seriously limit the allowed variance among processes. Moreover, this approach dictates "A" process that has to be followed, perhaps with minor allowed deviations.

The opposite approach would be to have only private schemas. This approach supports autonomy but eliminates collaboration (or at least tight collaboration as defined above), since a shared sub-schema is required for any two processes to tightly collaborate.

A third alternative would be to require identical product-data sub-schema, but allowing variation in the process-data sub-schema. This approach enables greater variance in the process models, while still retaining the capability for universal access to the product data from every process. (Note that "capability" means that it is technically possible to access the data, if desired. It does not imply a universally uncontrollable access mechanism.) In addition, it restricts the scope of a "template" process only to product sub-schema. While this approach supports a larger degree of autonomy, it is still prohibitively dictative, and unnecessarily restrictive. For example, a specific process may need to retain additional product data that is not required by other processes. A process might also want to share some of its "process" data with another process. And finally, data might not always be clearly classifiable in to one of the two categories. Thus, although the distinction between process and product data is important and can be utilized for other purposes (as will be seen in section 3.2.3), it seems like the wrong criteria for determining what parts of the schema to share or not to share.

The fourth and most general approach would be to have *shared* and *private* sub-schemas, regardless of the type of attributes/classes involved. Here again, a restricted approach would be to require the shared sub-schema be shared by all sub-environments, and a more general approach would eliminate this requirement, i.e., it would allow some but not necessarily all sub-environments to share a sub-schema.

This approach seems most unrestrictive, but it introduces certain difficulties: For example, if a class **C** is defined differently in two sub-environments **S1** and **S2** (e.g., the set of attributes differs) then instances of the class would have different structures in **S1** and **S2**, and a rule operating in **S1** on objects of class **C** from both **S1** and **S2** would have to be able to access objects from **S2** even though they are structurally different than the **C** objects from **S1**. An additional problem is a naming problem, i.e., how to "match" between multiple attributes, classes and class hierarchies which are isomorphic but are named differently. Finally, this approach requires static and dynamic mechanisms to analyze the differences between multiple data models in order to determine whether the definition is legal (static component) and in order to enable access to multiple items defined in different schemas (dynamic component). An example of two definitions of a class in two SubEnvs (taken from C/Marvel), is given in figure 7. For example, the `config` attribute which was represented in SubEnv **A** as a `string`, has been replaced in SubEnv **B** with a composite attribute that supports multiple configurations for the same source file.

Despite the difficulties in supporting multiple schemas, since the goal is to try to provide maximum flexibility in the degree of autonomy and collaboration, this unrestrictive approach seems most promising. Solutions to the aforementioned problems are partially addressed in section 3.2.2. However, the issue of schematic and data heterogeneity is in general a big problem in decentralized database research (see [31]), and is mostly out of scope for this thesis, so mainly practical and partial solutions to this problem will be given. An approach to solving this problem is given in section 3.2.2, and is based on avoiding any need for merging

```
----------
SubEnv A
----------

CFILE :: superclass FILE, PROTECTED_ENTITY;
# State Attributes
  analyze_status    : (NotAnalyzed, Analyzed) = NotAnalyzed;
  compile_status    : (NotCompiled, Compiled) = NotCompiled;
  object_time_stamp : time;
  compile_options   : string;
  config            : string

# File Attributes
  object_code       : binary = ".o";
  contents          : text = ".c";

# Composite Attributes

# Reference Attributes
  hfiles : set_of link HFILE;
end

----------
SubEnv B
----------

CFILE :: superclass FILE, PROTECTED_ENTITY;
   analyze_status : (NotAnalyzed, ErrorAnalyzed, Analyzed) = NotAnalyzed; ####
   contents      : text = ".c";
# State Attributes
  analyze_status    : (NotAnalyzed, Analyzed) = NotAnalyzed;
  compile_status    : (NotCompiled, Compiled) = NotCompiled;
  object_time_stamp : time;

# File Attributes
  contents          : text   = ".c";

# Composite Attributes
   configs  : set_of CONFIG_SRC_oz;

# Reference Attributes
  hfiles : set_of link HFILE;
end
```

Figure 7: Two Definitions of class CFILE

schemas, thus bypassing some of the general problems associated with "homogenization".

Decentralized Objectbase

An important characteristic of the model is that different local objectbases are *disjoint* from each other. This is an implication of the underlying decentralized model of separate, cooperating objectbases. That is, there is no logical objectbase that is transparently distributed. Moreover, composite objects cannot be partitioned into different objectbases, as this will violate the disjointness property, because a composite object contains its sub-objects.

Reference links can conceptually cross an objectbase boundary, since they do not impose a containment relationship. However, they imply resource sharing between sites (which is mutual when the links are bi-directional), which might possibly violate site autonomy . On the other hand, since derivation of parameters to rules during chaining is mostly navigation-based, cross-objectbase links are the only means by which rules can operate on objects from multiple objectbases when they are automatically invoked through chaining (as opposed to manual invocation of rules by end-users). Therefore they are an important feature and must be supported.

Access Control

Access control is a mechanism that allows to specify by who, and in what manner, an object can be accessed, regardless of any specific application (or in our case, the process) that tries to access it. (An example is the Unix premissions on the file system). In addition to support for access-control at the individual level, a DEPCE should provide a mechanism to control access at the team, or SubEnv level.

In general, the idea is to support autonomy by means of *invisibility*, i.e., any part of a local environment that is not intended to be accessed by remote agents is invisible to those agents. Furthermore, in order to have better control over exposure of an environment, simple global and local modes are not sufficient. That is, a local environment should be able to expose some but not all of its objectbase to some but not all remote agents. With respect to data, this means that an "exposure" parameter must be associated with objects, which can be either local (private), global (public) or contain a refined list of SubEnvs allowed access, along with the type of access (e.g., read-only).

As for composite objects, some restrictions should be made on the exposure mechanism. A public object should not necessarily enforce all of its sub-objects to be public as well. It can contain some private "hidden" sub-objects. However, the reverse is not true, that is, if an object is public, all of its containing objects (or ancestors) are at least equally public. This seemingly arbitrary constraint is essential in order to retain both the syntax and semantics of the composition hierarchy - i.e., retain the basic forest of rooted trees.

There is an obvious tradeoff between granularity of access control and performance. It seems that being able to refine the set of accessible SubEnvs is an important feature because it enhances the degree of collaboration without sacrificing autonomy and security. However, maintaining arbitrary granularity might be prohibitively expensive. One reasonable compromise (which will be applied in the first phase of the implementation) is to restrict access control specifications (for the SubEnv level) to root objects, i.e., top most objects of a composite object hierarchy.

Back to the issue of user-level access-control, a questions that arises is whether to deal with remote users in addition to remote teams. This seems undesirable, since the number of possible users is not known at any time, and obtaining knowledge at each site about users from other sites contradicts decentralization. An elegant solution is the concept of a *friend* user, (like the friend function in C++) which, regardless of the origin SubEnv from which it operates, can access the remote data as a local user.

Objectbase Visibility

As mentioned earlier, high visibility is an important requirement, but at the same time it complicates the model, because it implies that some objectbase information has to be replicated in users' clients, and has to be maintained fairly up-to-date while the objectbase is dynamically being changed by multiple users. In a DEPCE the problem is exacerbated due to the fact that some sub-objectbases are arbitrarily remote to a user's workstation. The general solution to this problem, which was used in MARVEL, is to transfer only *structural* information (henceforth "image") to clients, namely, names of objects and their relationships, while the actual modification of objects occurs in the local servers (see 3.3 for details). Furthermore, in order to minimize the amount of transfer, only a "delta" of changes could be sent to each client, depending on the last update it received [7].

However, even the "image" approach is unrealistic when applied uniformly to all SubEnvs, since some images have to be transferred from remote SubEnvs. Therefore, a tailorable refresh policy mechanism is desired, that can control the refresh setting on each sub-objectbase separately, taking into account both the distance and the frequency of interactions as a parameter. Note that an image is not required to always be up-to-date (although the DEPCE should make an effort to keep it that way), because it does not contain "real" data. Its sole purpose is to serve as a browsing display for seeing and referencing objects in the objectbase.

3.2.2 Decentralized Process Modeling

As mentioned in section 2, there are two aspects to process decentralization: decentralized process modeling (or definition), and decentralized process enaction (or execution). The focus of this subsection is on modeling, and enaction is discussed in the subsequent section.

As far as local processes are concerned, OZ follows the MARVEL rule-based modeling paradigm, namely: a four-level hierarchy of contexts: the *activity*, *rule*, *task*, and *process*. That is, each activity is modeled within a rule, each task is modeled as a chain of rules, and a process is

[7] This feature was implemented recently by Y.S. Kim but was not part of MARVEL 3.1.

conceptually a set of tasks. Note that in a rule-based modeling language, there is no explicit support for the definition of tasks. Instead, they are implicitly defined by matchings between an effect of one rule and the condition of another rule. Nevertheless, since a process engineer often writes the rules in order to impose a specific workflow and execution ordering among them, the task is considered as a distinct level of modeling.

As with data modeling, there are four analogous alternatives in decentralizing process models. The simplest naive approach is to have a single (global) process, and to totally rely on data modeling, i.e., every object that is accessible to a remote SubEnv can be accessed by every activity of the remote process. This approach clearly violates autonomy at the process level, as all SubEnvs must follow the same process. In addition, it is desirable in DEPCEs to have control not only on general access to objects, but also on the semantics of the operations and the contexts in which they are performed.

Another approach at the other extreme would be to have private processes at the SubEnvs. While it might still allow an end-user to *view* remote objectbases (provided that such mechanisms exist at the data level), it cannot support any data access by remote *processes*. Thus, this approach is obviously too restrictive, in that it basically eliminates any possibility for collaboration at the process level beyond pure information exchange. Once again, the key issue here is keeping the balance between autonomy and collaboration.

A third alternative is to refine the first approach and to have a global "template" process with allowed deviations at the local SubEnvs. While this is a step towards autonomy, it is still restrictive.

The fourth and most generic approach is to allow for arbitrary shared and private subprocesses, determined by the process engineers. But it is harder since independently-modeled processes have to be somehow integrated in order to collaborate.

Before presenting the formal model, an important issue that is promoted here is that collaboration, by its nature, should be explicit. Explicit collaboration means that the process of devising a collaborative/autonomous model should involve negotiations between the process engineers in order to reach agreement on the degree of autonomy/collaboration. An opposite approach would be "implicit collaboration" or "collaboration by default", in which each process publishes to the external world what parts of it are accessible publicly, and hides the rest. This approach may suffice for data sharing, where there is no real collaboration, since the ways in which remote *operating-system* processes access the data and the kinds of operations done on them are outside that model. An example of this approach is NFS, in which data subsets are exported and thus accessible to remote hosts, without requiring the remote hosts to communicate with the exporting host in order to gain access.

Collaboration among *software* processes involves much more semantics, and those cannot be left out without explicitly stating how each process accesses data owned by other processes. Hence, There is no attempt to totally automate collaborative process modeling.

Collaborative Process Modeling (CPM)

The idea is to express collaboration between processes as overlapping sets of subtasks that are imported-exported between the collaborating processes.

Note that unlike a data import-export facility, such an approach for processes might be hard if not impossible in the general case. For example, it is far from clear how to export a procedure of an imperative process program (e.g., as written in APPL/A [42]) with possible persistent data of its own, between processes. For example, this requires addition of a calling interface, input-output specifications, data-structure compatibility, and so forth. On the other hand, exploiting both the declarative and the decompositional nature of the rule paradigm (i.e., the fact that a procedure, or task, is fragmented and consists of multiple independent rules), it is feasible to export a subset of a "program" and integrate it with another program, as will be seen below. There has been an on-going debate in the PCE community on what is the preferred process-modeling paradigm (e.g., rule-based, petri-nets, imperative) and comparative studies were made in [29] and [45]. This topic is in general outside the scope of this paper, but for the approach described here the rule-based paradigm is clearly favorable.

Each SubEnv *exports* a subset of its rules that can be used by remote processes, along with the list of SubEnvs that are allowed access to them. A SubEnv *imports* exportable rules into its own process, provided that the rules are *schema-compatible* with the importing SubEnv. Schema-compatibility means that the sub-schema addressed in the imported rule is compatible with the schema in the importing SubEnv. For example, if the imported rule accesses an attribute **att** of class **C**, then **C**, as defined in the importing SubEnv, must have an attribute **att** with the same type. (As a reminder, this check is necessary because it is possible to have **C** defined differently in different SubEnvs.) As a side note, schema-compatibility can be checked once when imported rules are integrated with the local process.

Imported rules are the only means by which a remote process can access data owned by the process that exported them, except for the objectbase browsing operations "display-structure" and "print-object-contents", which are system, non-process-specific operations that are implicitly exported-imported by any two processes. Those operations can be prevented on parts of the objectbase only by data access control restrictions. Once rules are imported, they are integrated with the set of native rules, and become part of the process. Therefore, the specification of the export-import rules cannot, and should not, be made without verbal negotiation between the process engineers. Note that if a process **PA** exports a set of rules, it does not mean that it can use them on any remote process - it only allows other processes to use those rules on data owned by **PA** and on their own data. Thus, it is incorrect to simply treat the intersection of the exported rule subsets of processes as a shared pool of rules in which every rule of one process can be invoked on any data of another process. Allowing such behavior provides a convenient way to construct an "invading" process. In order for **PA** to access data in **PB**, it has to use a rule explicitly exported by **PB**. However, a real intersection of two processes can be made if both processes export identical rules. In fact, the main application of this mechanism is to enable cross-SubEnv rules, i.e., rules that operate on objects from multiple SubEnvs. Hence, In order to simplify the definition of such decentralized rules, a *shared* rule can be defined (again, with the list of sharing SubEnvs), and it suffices that all SubEnvs sharing the rule will have it defined as a shared rule, eliminating the need to import-export it from each site. Figure 8 clarifies this issue: A directed edge from a process **P** to a rule **R** in another process means that **R** was imported into **P**'s rule set, and is thus part of **P**'s process. In the figure, process **P1**

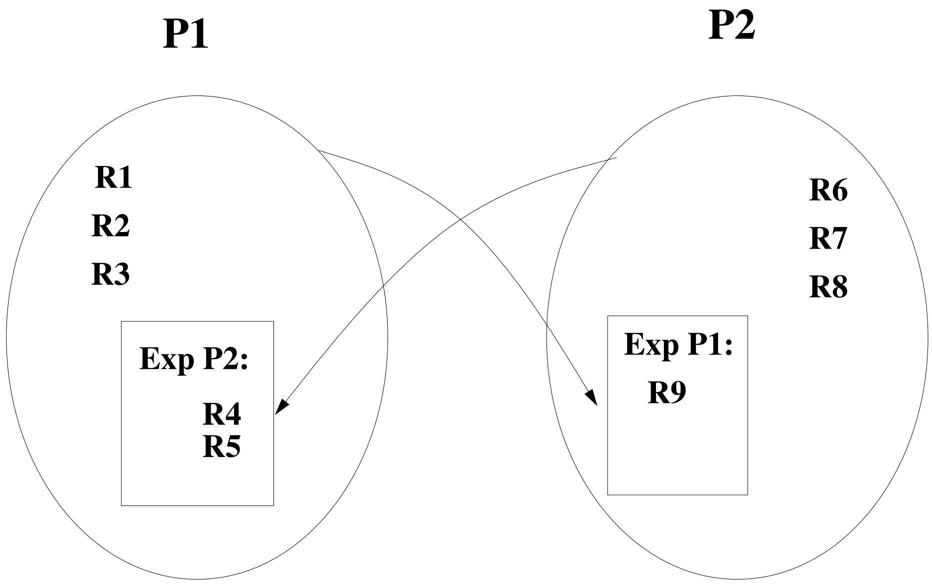

Figure 8: Rule import-export facility: First Example

consists of rules **R1-R5**, of which **R4** and **R5** are exportable to **P2**, and process **P2** consists of rules **R6-R9**, of which **R9** is exportable to **P1**. If **P1** imports from **P2**, it consists of rules **R1-R5** and **R9**, all of which can be invoked locally, but only **R9** can be used on **P2** and it cannot be used on any other remote process. Similarly, if **P2** imports from **P1**, it consists of rules **R6-R9** and **R4-R5**, but only **R4** and **R5** can be used when accessing **P1**, and they can be used only in **P1** and **P2** but not on other remote processes. Observe that if, for example, **R4** is identical to **R9**, then essentially the rule is shared between the processes. Thus, in order for a rule to be fired with parameters from multiple SubEnvs, the rule has to be appropriately imported-exported or be a shared rule.

Finally, a more complex example involving 3 processes and shared rules is given in figure 9, and the corresponding rule sets of the processes are: **P1**: { R1, R2, R3, R4, R5, R9(P2), R12(P3) R14(S)} **P2**: { R6, R7, R8, R9, R4(P1), R12(P3) R14(S) } and **P3**: { R10, R11, R12, R13, R5(P1) R14(S)}. Note that since **P2** did not import **R13** from **P3**, it is not part of **P2**.

The exported rules are essentially the means by which a process allows remote processes to access its local data. A remote SubEnv cannot access local data with an arbitrary rule. The collaboration between two SubEnvs (processes) is represented by the set of exported-imported rules. If the set is empty there cannot be any interaction between the processes, and if it contains the entire rule sets of both processes, it is highly collaborative. The implications of this model on chaining are given in 3.2.3.

Although the protocol is complex, its strength is in its flexibility with respect to the degree of autonomy and collaboration that can be expressed in and among processes. Explicit collaboration is exploited by the fact that both ends must explicitly agree and define the subset of tasks that can be used across SubEnvs, while with the absence of such definitions autonomy is enforced by default.

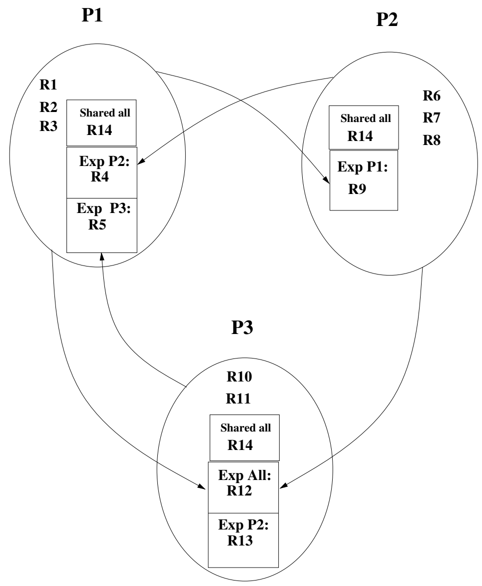

Figure 9: Rule import-export facility: Second Example

Inter Process Consistency

One more problem to be solved in this model, is how to ensure that a process that imported rules from another process, still retains its process-consistency, i.e., whether the *invariants* that were defined in the process are still valid in the objectbase with the extended rule-set. Process invariants are similar to program invariants, i.e., these are premises made on some parts of the data which are assumed to always hold on any instantiation of that process. For example, a simple invariant might be specified as: "A system is built if all of its subsystems are built." Then, it cannot be the case that at any time, in any instantiation of that process, a subsystem is not-built yet the system is marked as built. Adding new rules to an existing process may introduce new invariants, some of which might conflict with the pre-existing ones. As mentioned in section 1.2, MARVEL supports the modeling of consistency using "consistency predicates", with a proper enaction mechanism that preserves the defined consistency model.

The solution to this problem lies in the fact that augmenting the imported rules to an existing process can be regarded as *evolving* the process, and so a process evolution tool that checks syntactic and schematic (i.e., whether the rules operate on a compatible underlying schema) errors and analyzes the changes to detect possible inconsistencies in a single process is sufficient for retaining the consistency of the entire environment, because imported (and shared) rules are the only means for processes to interact with each other. Initial work on process evolution in PCEs has been done in MARVEL [27], and an Evolver tool was developed that is able to detect and update an objectbase to correspond to the newly defined process consistency. It is intended to be extended in the thesis for use by the import procedure.

3.2.3 Decentralized Process Enaction

Per SubEnv, Oz assumes a four-level hierarchy of process enaction: *activity*, *rule*, *task*, and *session*. At the global environment level, a fifth, *SubEnv process* level exists. The activity, rule, and task levels correspond to their definitions in process modeling, with the additional association of a rule with a "transaction", and a task (realized by a chain of rules) with a "nested-transaction". A transaction encapsulates a logical and atomic unit of work and a nested transaction encapsulates a composite unit of work. Transactions are deliberately only abstractly defined here, to avoid deviation from the main topic of this section. The session level does not have a counterpart in process modeling, and corresponds to the sequence of operations done by a single client from login to logout.

In the decentralized enaction model, the smallest unit of decentralization is the *rule*. This implies that a task containing multiple rules, and a session containing multiple tasks, are also decentralizable, and an activity is *not*. This does not mean that all activities of a given task must execute necessarily at the same SubEnv, but an activity by definition executes in its entirety in one SubEnv. A possible extension would be to redirect execution of an activity to different sites, for purposes of special-purpose resources, load-balancing, etc. Nevertheless, the activity would still be executed in its entirety in one host.

The main reason for this decision is that activities in Marvel (and subsequently in Oz) are considered as encapsulating external tools using a "black-box" approach and interfacing them to the environment [20]. As such, there is no environment control over the internals of activity execution. (Although an approach for "grey-box" integration has been contemplated, black-box integration would still be a required feature.)

At first glance, there are two ways in which a task **T** can be decentralized in execution: (1) **T** imports remote objects into its own process and executes locally, or (2) **T** spans a subtask which is executed remotely. This is similar to the two approaches in execution of a distributed program: fetch the data and execute locally, or send a request for remote function execution (a.k.a. remote procedure call). There are obvious trade-offs between the two approaches, and the superiority of one approach over the other largely depends on the nature of the activity and the volume and nature of the input and output data involved.

Viewing the Oz process model as described so far, several observations can be made: First, process autonomy and consistency restricts application of the first approach, since remote data may not be accessible to, or updateable by, the the local process. Second, activity execution restricts application of the second approach because tools invoked by an activity may not be available at the remote SubEnv. In addition, activity execution is not decentralized so there is no process control over its execution. But most importantly, a task in Oz is not a predefined set of operations. Rather, it is an initial activity, preceded (backward chaining) and proceeded (forward chaining) by implications of that activity. Moreover, conceptually, a task is not executed necessarily sequentially (although it was implemented that way in Marvel). The "program" to be executed consists of fragments of subtasks, which are not necessarily known in advance, since their activation may be triggered as a result of a change that was made in the objectbase (i.e., forward chaining). This paradigm may be well suited to decentralization, since a decentralized execution "falls through" – by modifying a remote object, the remote site's own process continues to execute by responding to the update. Thus, Oz employs a third approach, which differs from the two approaches mentioned above: At the activity level, fetch remote objects and modify them locally, but at the task level, fork and continue executions of subtasks in the remote sites. This execution model is explained succinctly below.

Communicating Software Processes (CSP)[8]

When a rule is fired (either manually or through chaining), the data set accessed by the rule being fired can consist of:

1. local objects only;
2. remote objects of a single remote SubEnv;
3. a mixture of local and (possibly multiple) remote SubEnvs;

The first case is handled locally (like in Marvel) and is straight forward.

In the second case, a check has to be made that the rule being fired can access the remote process (i.e., it has been exported by the remote process and imported by the local process). Once verified, the execution is done in a "remote-procedure-call" fashion, i.e., the remote SubEnv executes the rule. Essentially, this case is similar to invocation with only local objects, except the invocation point is remote. The rationale for choosing this approach is obvious: Since the entire operation executes in a remote site, there is no point in involving the local process in any way (except for possibly transferring data for activity execution purposes).

The difficult and interesting case is when a rule being fired involves a mixture of local and remote data. Assume process **P1** in SubEnv **SE1** invokes a rule **R1** with parameter objects from both **SE1** and from a remote SubEnv **SE2** maintained by process **P2**. (For simplicity, two SubEnvs are assumed, but this can be extended to multiple SubEnvs.) Call **SE1** the *coordinating* SubEnv. **R1** is effectively a shared rule, since it must be part of both **SE1** and **SE2** in order to be able to access remote objects. Recall that objects can be bound either as actual or derived parameters which are collected through the binding section of the rule.

The execution of rule **R1** is done as follows:

1. *binding* - The relevant attribute values of the remote objects (i.e., those considered in the rule's bindings) are transferred to **SE1** in order to evaluate the condition of the rule. Note that data autonomy is not a concern here, since by definition, **R1**'s objects are visible (otherwise they couldn't have been bound to **R1**).
2. *condition evaluation* - If the condition of the rule (a logical expression) is not satisfied, the list of unsatisfiable predicates along with the list of "failed" objects from **SE2** are sent to **SE2**. Note that in some cases, depending on the condition and the values of the attributes, an unsatisfied condition can be made satisfiable only by considering local objects. For example a condition of the form -

[8]There is no relation to Hoare's Communicating Sequential Processes.

```
(or
    (?a.status = 1)
    (?b.status = 1))
```

where `?a` is bound only to local objects and `?b` is bound only to remote objects, should be evaluated by trying to satisfy the first predicate, and only upon failure try to satisfy the second predicate, since satisfying either of them is sufficient. On the other hand, it is in general impossible to do decentralized condition evaluation, since the condition may be a nested logical expression involving predicates which are evaluated at different SubEnvs. Moreover, each symbol in a predicate may have bindings of objects from multiple SubEnvs, in which case even a single quantified predicate would have to be evaluated in the coordinating SubEnv. So, although each object's state with respect to a condition can be evaluated at the object's origin SubEnv, the combination of the evaluations must be coordinated at a central place. As mentioned earlier, query optimization per se is outside the scope of this thesis, but evaluation techniques will be discussed in the thesis.

In order to satisfy the condition, both **SE1** and **SE2** backward chain as necessary off the failed predicate(s), *each according to its local process*. If the predicates are satisfied, then the updated objects, along with the relevant portions required for the execution of the activity, are sent back to **SE1**, which in turn, invokes the activity.

3. *Activity Execution* - This is done in **SE1**, the coordinating SubEnv. Note that the activity is the only part of the rule that accesses files (represented by file attributes), which can be arbitrarily large in volume. Thus, in case of physical decentralization, the transfer of product data occurs only in this phase to avoid unnecessary overhead (see section 3.3 for more on remote data transfer).

4. *Assertions* - On completion of the activity, **P1** asserts the local effects on local objects, and sends the remote objects to **P2**, which in turn asserts the effects in **SE2**, and subsequently, *each process forward chains according to its own process*.

The interesting point here is that both backward and forward chaining occur in the local SubEnvs, according to the local processes, and process data is in sole control of the local process, while execution of the actual activity involves local and remote data. Since process enaction is recursive, both backward and forward chaining can lead to further cross-SubEnv execution.

A metaphor to this execution can be a "summit meeting": Before the meeting (the actual activity), each party (SubEnv) takes care of handling local constraints that are necessary for the meeting to take place, then the meeting (activity) is being held, where the various parties meet together and collaborate, and once the meeting is over and there were made conclusions (return code of the activity), all parties return home and carry out the implications of the meeting in their local sites.

An important principle employed is that an objectbase is "owned" by the local process that operates on it. Thus, while process data (which is specific to a SubEnv) is logically manipulated by multiple processes, it is physically manipulated only by the local process, while product data can be accessed and modified by remote processes. This is where the distinction between process and product data is significant.

The following example illustrates how two different processes can co-exist and produce useful results. This example involve collaboration at the "micro" level, involving fine-grained activities. The next section described a solution to the "motivation problem" given earlier.

In SubEnv **SE1**, an *edit* activity on a CFILE cannot be executed unless the file has been reserved with `RCS` [46]. After editing, and if changes were made, the file should be compiled, and if successful, notify dependent "caller" modules about the change.

In SubEnv **SE2**, an *edit* activity on a CFILE cannot be executed unless the file has been reserved with `SCCS` [38]. After editing, and if changes were made, the file should be analyzed using `lint`, and if successful proceed with compilation.

Rule **multi-edit** takes two CFILEs as parameters. Assume that multi-edit is fired from **SE1** with one local object, **a**, and one object from **SE2**, **b**. Further assume that multi-edit is a "shared" rule. Figure 10 illustrates the flow of control when executing the **multi-edit** rule (failures are ignored for now, these will be explored in the thesis):

Once the two objects are bound as parameters to the rule (following user selection of those objects) the condition is examined. If a predicate on **b** is not satisfied, then the failed predicate is sent to **SE2**, which in turn tries to satisfy it by backward chaining off the failed predicate. In this case, it will invoke the local `SCCS` activity. If a predicate on **a** fails, it will backward chain locally to execute similar rule, but will use `RCS`. Once conditions are satisfied, the updated **b** is shipped to **SE1** and the activity (in this case, editing) is executed in **SE1**. Once done, the proper assertions (depending on the result of the activity) that should be asserted on **b** are sent to **SE2**, which in turn starts a forward chain off those assertions. In this case the assertions trigger a `lint` and `compile`. On the other hand, **SE1** continues with compile followed by a notification subtask (which in turn, may trigger further forward chains to reach transitive closure of callers). This example shows how two processes with different tools (`SCCS` vs. `RCS`), and different process workflows, can collaborate.

3.2.4 Solution of Motivation Example Using CSP

Figure 11 describes one way of modeling the motivating example task described in section 2.2, with additional details about the local processes, and its actual enaction using CSP. Note that each box in the figure does not necessarily represent a single activity but rather sub-tasks which in reality would be broken down to a fine-grained set of rules.

A **change** task is initiated by the coordinating SubEnv **SE2**, which requires backward chaining to **pre-approve**. This takes place in a decentralized manner, and is preformed differently at different SubEnvs: Unlike **SE3**, **SE1** requires a separate analysis step before performing the pre-approval activity. Once done, the actual change is applied to **L**, followed by decentralized forward-chaining to the various testing sub-processes at the SubEnvs. Note that **SE3** has a manual-test procedure (e.g., for testing user-interface) which involves human users coming up with the input sequences for the testing suites and actually performing the

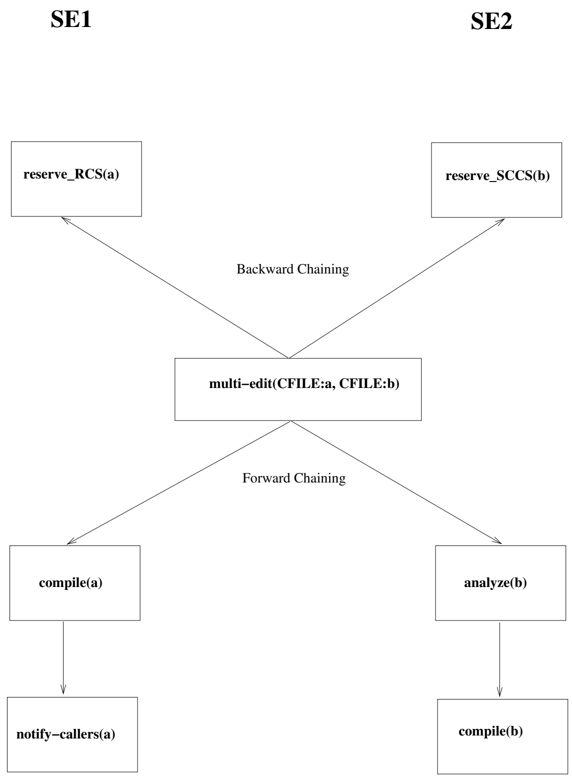

Figure 10: Example of the CSP protocol

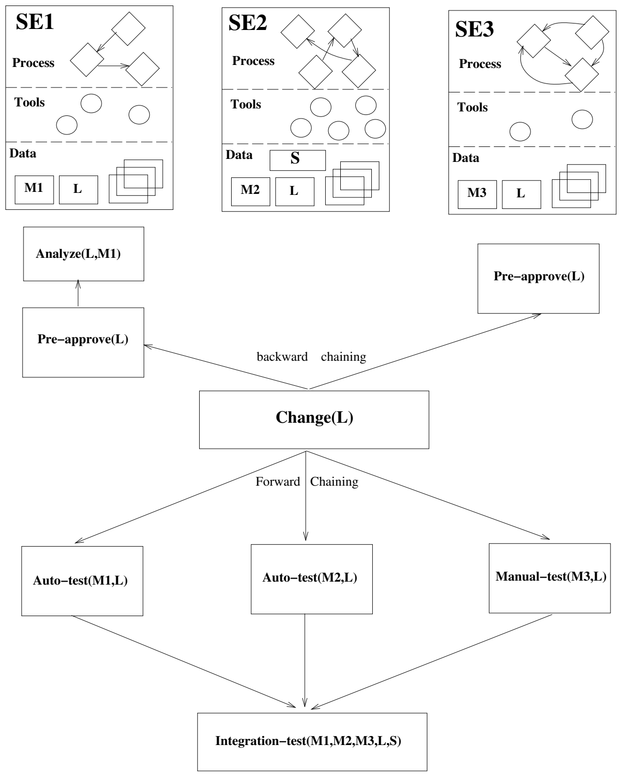

Figure 11: Solution to Motivating Example Using CSP

tests, whereas the other SubEnvs have an automatic testing facility. Once unit-testing is complete (and assuming no errors found), an integration test is performed in the coordinating SubEnv, involving product data from all SubEnvs, and thus performed in a centralized manner.

3.3 The Architecture

This section presents the physical architecture of Oz, intended to realize the decentralized model. The architecture is based on Marvel's architecture (as presented in section 1.2.5), and extends it in two main directions:

- Multiple instances Marvel's core entities – servers, clients, loaders and daemons – and added functionality and interfaces among them.
- Re-engineering of the architecture to accommodate true componentization, both for replacing components within the Oz framework and for exporting components to external frameworks.

As mentioned earlier, this proposal focuses on the first aspect, namely decentralization. Componentization is a group effort of the Oz project as a whole, and is not specifically addressed here, although it will be partially addressed in the thesis.

The following aspects of the architecture are discussed in this section:

1. Large-grained components and their inter-connections.
2. Specification of architectural extensions.
3. Communication among SubEnvs.
4. Decentralized naming schemes.
5. Dynamic configuration of SubEnvs.

3.3.1 Components and Inter Connections

The Oz architecture is illustrated in figure 12, showing two active SubEnvs. The system consists of three core runtime entities – Oz Connection Server (OCS), Oz Server (OSV), and Oz Client (OCL) – augmented by the process translator Oz Loader (OLD), and process evolution tool Oz Evolver (OEV).

Oz Connection Server (OCS) is an extension of the Marvel daemon. It is responsible for (re)establishing connections from local clients, remote clients and remote servers, into a local SubEnv, by serving as a mediator for "hand-shaking" between the various Oz entities. However, it does *not* participate in the actual communication between the entities. In addition,

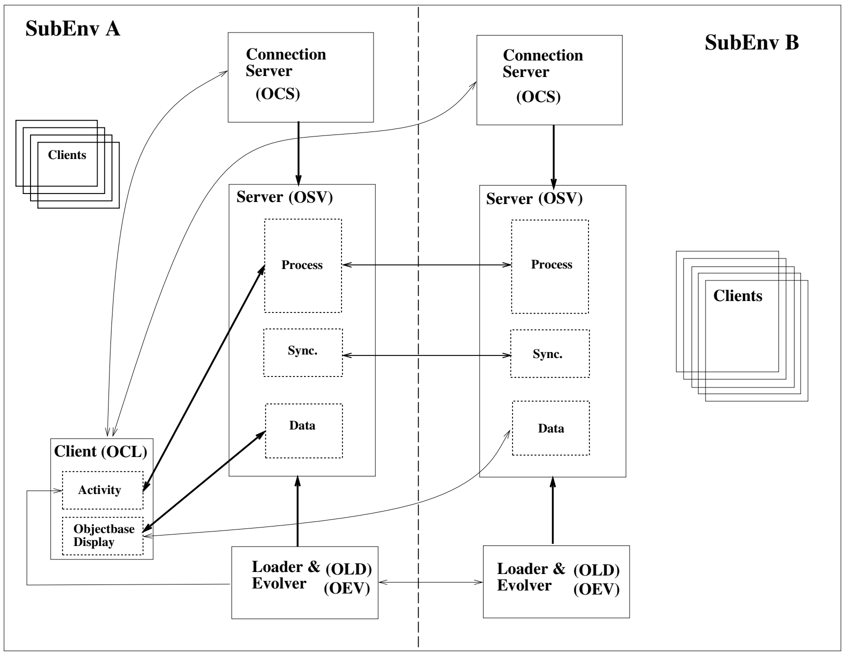

Figure 12: Oz Architecture

OCS provides a "wakeup" operation to activate servers on inactive SubEnvs upon (local and remote) requests for service.

Oz server (OSV) is the central entity in Oz. It enacts a project-specific process on a SubEnv, manages its data and serves local and remote clients. Like Marvel's server, it is composed of three main components: *process*, *synchronization*, and *data* managers.

Oz Client (OCL) is the front-end to the end-user, consisting of the user interface including an objectbase browser, an activity execution manager, and a message handler.

Oz Loader (OLD) is a collection of programs that translate and load the external project-specific process model, encapsulated tools, synchronization/coordination model, and the data model into the process, activity, synchronization, and data managers, respectively.

Oz Evolver (OEV) supports process and schema change and evolution, by analyzing the changes and evolving the objectbase to be consistent with the new data and process models.

Inter-component Connections

OCS is bi-directionally connected to local OCLs for activation requests, and to remote OCLs for activation as well as for SubEnv name resolution and for establishing initial connection and re-connections to the SubEnv (see section 3.3.3). In addition, OCS is uni-directionally connected to the local OSV that it either activates, or redirects to it indirect requests from remote OCLs.

OSV maintains connections at several conceptual layers. The process manager is connected to other process managers of related OSVs for execution of a decentralized process as part of the CSP protocol. Similarly, the synchronization manager is connected to other synchronization managers of related OSVs. However, the data manager is *not* connected to other data managers, since the data is owned and accessed by its local process[9]. But the data manager is connected to both local and remote OCLs for displaying the objectbase image. In addition, the link between the data manager and remote OCLs is used to transfer product data for activity execution when the file system is not shared. Finally, the process manager maintains connections to its local OCLs and possibly to "friend" remote clients (remote clients that have access to the local SubEnv) to service their requests (for simplicity, this kind of link is not shown in in the figure).

OCL maintains connections to process managers of local and remote OSVs, and to local and remote OCSs, as explained above.

OLD and OEV are both connected to their local OSV for loading and evolution of processes, and to peer OLDs and OEVs for implementation of the CPM model.

Note that some of the connections do not necessarily always exist, but they can potentially be formed if needed. The bold links (between a client and its local server) are assumed to always exist. Also, note that the links drawn in the figure are conceptual links, which are physically realized in lower levels of the architecture.

Operational Overview

Each global environment consists of **n** SubEnvs. At any point, none, some, or all SubEnvs may be active, meaning that exactly one OSV (and usually at least one OCL) is running on that SubEnv. A SubEnv is associated with a site, but not with a specific host, since the file system is shared within a site, so a SubEnv can run on any host in the site.

Each site can have zero, one, or multiple different SubEnvs, from the same or different global environments, spread out arbitrarily among hosts in the site. Their entry points are maintained in the OCS's *environment base* (see below) which is a site-specific table.

A user logs in to a specific SubEnv (of a specific global environment) by invoking an OCL, which in turn establishes a communication link with the local OSV, assumed to reside on the same site (i.e., share the file system). If the OCS is currently inactive, OCS is responsible for activating it, with preference to activate it on the host where the SubEnv's objectbase physically resides (termed the primary host). A user can have any number of active OCLs connected to the same or different OSVs.

Once the initialization phase is complete, the user receives (through a client) a set of process rules, an image of the local objectbase and its schema, and a list of SubEnv objects, each of which represents a remote SubEnv of the global environment (see section 3.3.3 for the description of SubEnv objects).

Using the SubEnv objects, the user can `expand` and `shrink` an image of a remote objectbase in order to access directly remote objects (subject to access control restrictions on the remote site). Recall that "image" means that only structural information for browsing is physically

[9] Such a link might be still needed in the future for geographical decentralization to support data prefetching.

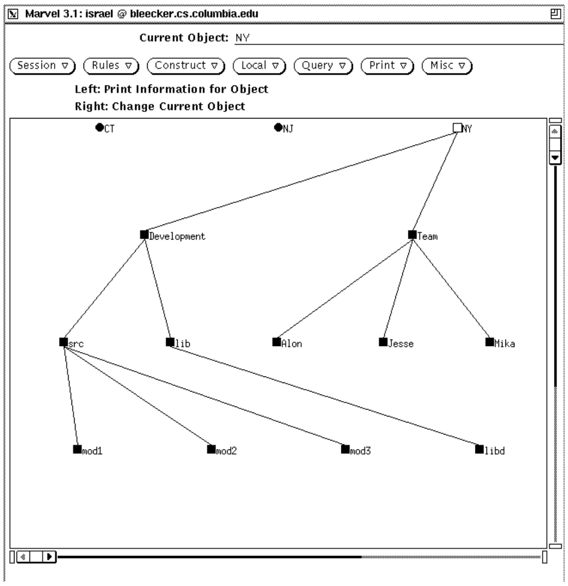

Figure 13: Oz objectbase

in the client, not the contents of the objectbase. The `expand` operation is established via a "remote login" request to a remote server, which entitles the client to browse the remote objectbase but not to use the process operating on it, as explained below. The "refresh" policy of remote objectbases may vary depending on the degree of remoteness and frequency of interactions. Remote objects can be accessed indirectly, as a result of a process (sub)task that involves remote access. In this case there is no need for an explicit expansion request.

A populated global objectbase is modeled as a forest of trees, each of which represents a local objectbase rooted with a special SUB_ENV object. For example, figure 13 shows an image of the local objectbase rooted at object **NY**, which is expanded by default, and two other remote SubEnv objects, **CT** and **NJ**, which are not expanded at this point; and figure 14 shows the objectbase with the two remote SubEnvs expanded. Note that although the objectbase contains also "horizontal" non-hierarchical links, there is always a tree structure imposed by the hierarchical composition links, so a tree structure always exists. Also, note that cross-objectbase links can be formed.

The architecture, as presented in the proposal, does not take into account all the implications of large geographical distribution yet. However, it does not preempt such operation, i.e., it does not assume that all involved entities run in a single site either. While an OSV and its local OCLs are assumed to reside on the same site and share the file system (but not other resources), no shared resources such as processors, memory or file system, are assumed between multiple OSVs and between OSVs and remote OCLs. The main issue to resolve for arbitrary geographical distribution is to incorporate the "variable-bandwidth" factor into the architecture, and to address the problems associated with transferring files across sites efficiently when the file system is not shared.

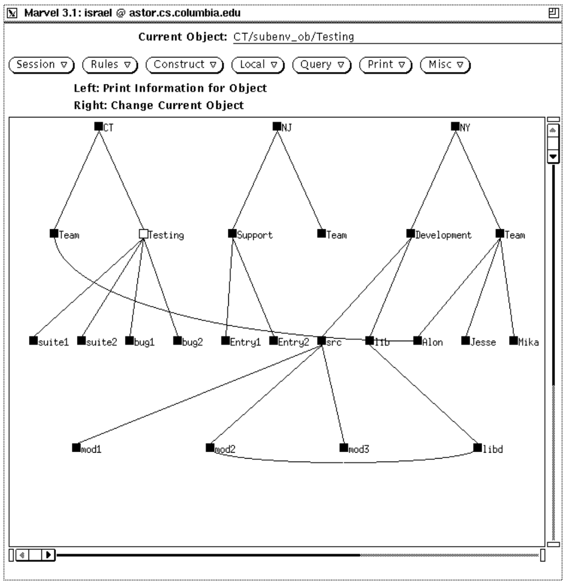

Figure 14: Expanded Oz objectbase

3.3.2 Architectural Extensions

The following is a specifications of the extensions to the Marvel architecture that are required in order to build the Oz architecture. These include listings of implementation problems to be addressed in the realization of Oz . The general communication and configuration model is described separately in subsequent sections.

OCS

Like Marvel's daemon, OCS has a "well-known" address (realized by the Unix network `services` data base) and is conceptually always active (realized by the Unix `inetd` mechanism). OCS extensions are:

- Remote activation of servers within a site (via `rsh` and `exec` mechanisms).
- Environment name resolution and maintenance of the Environment Base (EB) (explained in section 3.3.3) for establishing connections between remote OCLs and the local OSV.

OSV

Since OSV is the central component of Oz, extensions to this component are the main implementation effort of this thesis. The main extension is in supporting the CSP protocol for execution of decentralized processes in multiple SubEnvs by multiple OSVs. Specific extensions include:

Process Management

- Decentralized binding and condition evaluation.

- Chaining across SubEnvs - mechanisms for chaining from one SubEnv to another. This includes exchanging relevant information between OSVs and extensions to the current "inversion" algorithm for binding objects to parameters of rules [24].

- Scheduling - Marvel has a single context-switch point within a single rule (before sending an activity for execution at the client). This is adequate since both the condition evaluation and the assertion of an effect of a rule is performed atomically. However, a rule evaluation in Oz might involve multiple processors, some of which are remote, which requires to enable the coordinating OSV to switch contexts while waiting for binding or query evaluation response.

Synchronization and Failure Recovery

As mentioned earlier, synchronization is in general out of scope and is investigated in parallel by Heineman. The following is an outline of the problems to be addressed in the implementation of the synchronization component.

- Decentralized transaction management.

- Extensions due to non-atomic evaluation of a rule.

- Extensions to the programmable interface to support inter-SubEnv. coordination

- Recovery of partial (site) failures.

- Implications of Prefetching (for physical decentralization).

Object Management

- Support multiple differing schemas.

- Cross-SubEnv links, which are important to enable semantic relationships between objects in different SubEnvs, and to enable cross-SubEnv navigational queries, which are necessary for cross-SubEnv chaining.

- Split and merge operations on objectbases which are necessary to support mobility and dynamic reconfiguration.

- Built-in operations on objects across SubEnvs, such as **move**, **copy**, **link**.

- Distributed query evaluation and optimization, in the presence of multiple schemas in a global environment.

- Prefetching of remote objects for local processing, if the file system is not shared.

- Access control - extensions to Marvel's access control to support partial visibility to remote SubEnvs.

OCL

Like MARVEL's client, OCL is a "process-less" entity, i.e., it has no knowledge of the process it operates on, and can be viewed as a front-end interface to the environment. The fundamental difference between OCL and MARVEL's client is in the support for multiple connections to servers. OCL can view multiple objectbases from multiple SubEnvs (i.e., multiple OSVs) simultaneously, both for viewing purposes and for selecting objects from multiple objectbases as arguments to multi-SubEnv rules. However, an OCL is always attached to a single OSV (hence a single process), typically its local OSV. A possible extension may be to allow an OCL to switch among processes, and add the notion of a "current" process, where the access to a process is manifested by having the rule set of the current process accessible to the client.

Another required extension is in enabling to specify an objectbase image refresh policy on a per objectbase basis. This is crucial for physical distribution, where different SubEnvs may have totally different bandwidths between them and the OCL.

As for activity management, since an activity is modeled in OZ as an atomic non-decentralizable unit, decentralization should have minimal effect on its design. The main difference is that executing an activity on a remote objectbase requires physical transfer of product data, since the file system is not necessarily shared among SubEnvs. Thus, under the "variable bandwidth" assumption, prefetching techniques such as in [43] would be employed to enhance the performance of remote activity execution.

OLD

- Language extensions to support CPM (import-export).
- Distributed process translation and loading mechanism to support CPM.

OEV

- Addressing fully the inter-process consistency problem introduced by CPM.

3.3.3 Communication Model

The following section is organized as follows: First, high-level design goals are presented; second, the notion of "SubEnv-based" addressing is presented; third, a description of the model for storing the communication information is given; and fourth, the general protocol for communication across SubEnvs is given.

Design Goals

Tailorability - Ideally, all the information necessary for providing the various communication services should be tailorable and specified with a modeling language, which could either be an extension of the process modeling language or in separate language, while the runtime entities should provide mechanisms to implement those specifications. However, communication modeling poses some problems that do not exist in process modeling: First, since communication is primarily concerned with inter-SubEnv interactions, tailoring can be made only on a global environment basis (as opposed to per SubEnv), which means that communication modeling is at least partially a global modeling procedure. Second, communication involves low-level protocols and mechanisms that are usually built-in internally in the system and are not exposed to higher level applications. Nevertheless, it is clear that for arbitrary geographical distribution, some form of parameterization or programmability will be required to determine inter-site communication.

The first step taken here towards providing communication modeling capabilities, as will be seen shortly, is that all the information regarding communication services is modeled with classes and is kept internal to the objectbases of the SubEnvs, and a set of rules operate on them. However, the classes and rules are built-in, and known to the system. The second step would be to explore possibilities to generalize this approach and allow for each global environment to define its own communication classes and rules, and evolve them when necessary. As mentioned earlier, another specific aspect of communication modeling that should be parameterized is the client-server "refresh" policy.

Naming - In order to exploit decentralization, no central entity is assumed. This applies to naming schemes, data repositories, and control procedures. In particular, the configuration of a global environment is replicated in all SubEnvs to avoid the need for a central gatewaying entity. Naming is discussed separately in section 3.3.4.

Performance - The main requirement is to keep the communication overhead of the possibly overloaded multi-user servers as low as possible. Therefore, whenever possible, clients (which are single-user) should communicate with remote servers directly, without going through their local servers. In particular, operations that do not involve "process" interactions (e.g., browsing remote objectbases) should be done with minimal or no intervention of the local server.

SubEnv-based Addressing

A fundamental aspect of the communication model is how to identify and locate remote SubEnvs of a global environment.

One possible approach, termed here *site-centered*, is to bind each SubEnv with a site (and a corresponding site object). The advantage of this approach is that it relies on the Internet addressing scheme for SubEnvs, thus eliminating the need for an Oz-specific name space for SubEnvs. However, site-based addressing has several limitations: First, it restricts to have in each site only one SubEnv per global environment in order to provide uniqueness. But there can be cases where it is desirable to maintain several SubEnvs of a global environment in the same site, each SubEnv with possibly different responsibilities (i.e., testing team and development team), and thus with different processes. Second, it implies hard binding of a SubEnv to a specific site. This may not be desirable, especially for reconfiguration purposes.

Thus, an alternative approach chosen here is a *SubEnv-based* addressing. In this approach, each SubEnv is identified and represented by a special SubEnv object with a unique identity (see below). The SubEnv object contains all the necessary contact information, including

```
SUB_ENV :: superclass ENTITY;

# Static Information
   env_name     : string;
   env_id       : integer;
   subenv_id    : integer;
   subenv_name  : string;
   site_name    : string;          # hostname()
   site_ip_addr : string;          # dotted format

# Dynamic information
   active_host  : string;          # hostname()
   host_ip_addr : string;          # dotted format
   port         : integer = 0;     # port number, if active
#  protocol     : (TCP, UDP, OTHER) # Not supported yet
   local        : boolean;         # TRUE if local, FALSE if remote
   state        : (New, Initialized, Defunct ) = New;
   active       : boolean = FALSE; # TRUE if active, not guaranteed
   subenv_ob    : set_of ENTITY;   # The actual objectbase
```

Figure 15: SUB_ENV built-in class

its network address. The important point here is that unlike in site-based addressing, the network address information is changeable and is *not* part of the identity of the object.

Communication Database

OZ maintains two repositories of information for storing location information: the SubEnv map, and the Environment Base (EB).

The SubEnv Map is a repository that contains all the necessary information for locating and communicating with remote as well as local SubEnvs. The SubEnv map is a semi-replicated resource (the notion of semi-replicated will be clarified shortly), and all SubEnvs of a given global environment contain it. Note that unlike distributed non-decentralized systems, OZ's SubEnv map is not hidden; it is available in all levels of abstraction including OSVs, OCLs, and users who can see the map and know where each object physically resides.

As explained above, in order to integrate the communication protocols within the process framework, the SubEnv map is maintained within the objectbase of each participating server, and is manipulated by special built-in rules. Each SubEnv maintains its connectivity information in an object instantiated from a built-in class named `SUB_ENV`, as defined in Figure 15. Figure 16 shows an example of a `SUB_ENV` instance corresponding to the **NY** object in figure 13.

The SUB_ENV class consists of *static* and *dynamic* components. The static component contains general *site* information (e.g., environment and SubEnv ids). The dynamic component contains frequently-changed information regarding the currently active *host*, such as ip address, port number, etc. The distinction between the dynamic and static components is used in the communication protocol (see below).

The Environment Base (EB) - Every site has a single EB table maintained by its OCS, with entries for all SubEnvs registered in the site, each SubEnv listed within its global environment name, to identify them uniquely (see section 3.3.4). Each entry contains the global

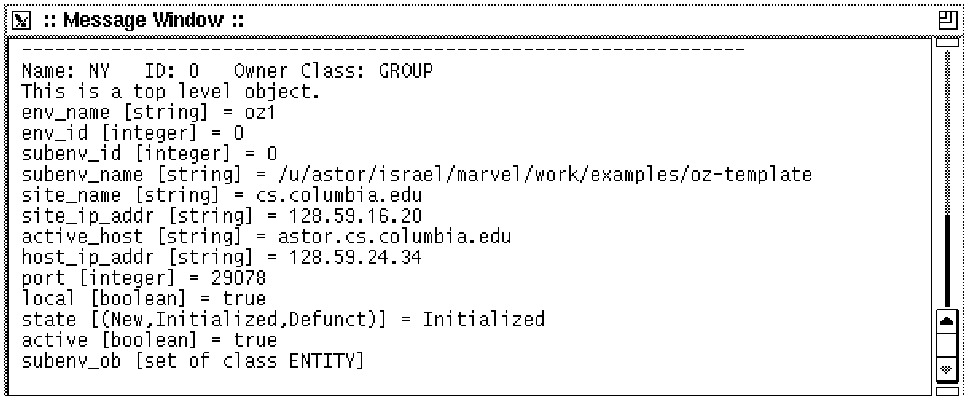

Figure 16: A SUB_ENV instance

environment name and id, the SubEnv name and id, and the local environment directory (i.e., directory in the file system where the local environment resides, including the objectbase, the process, and the file system containing product data). Note that this is the only place where there is a mapping between a SubEnv and its associated directory. Thus, remote servers or clients are not aware of the location in the file system of the environment, since it is not assumed that they have access to this file system. Since an EB is site-specific (as opposed to SubEnv-specific), it need not be replicated. It is updated when the configuration changes (see 3.3.5).

The Communication Protocol: A Lazy Update Approach

There is an inherent trade-off between providing good average performance to all communicating entities regardless of their degree or kind of interaction – and between providing high performance as a function of the degree of interaction between the entities. In decentralized environments, the price of maintaining average performance may be very high, because there is no bound to the distance/bandwidth between entities, and the communication overhead for updating the entities about changes made is likely to be very high, especially when the environments are not always active, as in Oz.

Thus, the latter approach is adopted here, where the system as a whole makes some effort to keep the SubEnv map up to date with respect to the *dynamic* information (maintained in the SubEnv objects). However, this information is not assumed to always be valid. In contrast, the *static* information is assumed to be valid invariably. The idea is to keep the "freshness" of the dynamic information as a function of the frequency of communication. That is, the more a remote SubEnv is contacted, the more its dynamic SubEnv object information will be accurate at the contacting SubEnv.

There are essentially two possible channels of communication. The *indirect* channel involves going through the remote OCS in order to locate the remote server and establish the connection. This is done using the static site information of the SubEnv object representing the remote server. The other, *direct* channel, is using the dynamic host information in order to contact directly the remote server. A remote server can always be located using the indirect channel which is always valid, but with higher cost. (The only time when such a connection would fail is when the site itself is not reachable.) On the other hand, the direct channel provides fast connection, but may be out of date. In some cases, there isn't even a running server on a given SubEnv, which means that indirect communication must take place. The main point is that if the communication between the entities is frequent, the dynamic information is kept up to date by the remote server. That is, *a server updates its representative SubEnv objects in remote SubEnvs only on demand*. This is why the SubEnv map is only semi-replicated: while the static information is truly replicated, the dynamic information is not, and a SubEnv object may vary in its contents in different SubEnvs. Note that indirect communication is essential to handle host and intra-site failures, in which case it might be able to restart a remote server on a different host in the site. In addition, since a SubEnv may not be active, the OCS must invoke a server when a request is made to contact it.

The main benefit of the double mode approach with "lazy" update is that there is no need to maintain frequently changing information up to date in all SubEnv servers, which are arbitrarily distributed and vary in their frequency of communication between them. Such a requirement would be prohibitively expensive. On the other hand, the system can still provide direct and faster access between entities that communicate frequently.

The actual protocol for communication between remote entities (either client-server or server-server) is described in figure 17. For example, in case of client (**CL**) to remote-server (**RS**) communication the protocol is as follows: if **RS** is marked in the local SubEnv map as active, then the client uses the (dynamic) host information to connect to the server. If the connection fails, or if the SubEnv object representing **RS** is marked as inactive, it uses the (static) site information of the **RS** object to establish connection. If successful, it gets the active host information from the currently active host, updates the local SubEnv map, and communicates. If connection is refused then there is a system failure (e.g., network breakdown, primary host of the SubEnv is down etc.). Note that all the information for both direct and indirect communication is obtainable from the local SubEnv objects (as defined in Figure 15) and the environment base.

Finally, the addressing scheme as presented so far allows only for a flat address space for SubEnvs. This may be limited and inefficient as the number of SubEnvs in a global environment grows. Extending the SubEnv address space to support a hierarchy of SubEnvs may be explored in the thesis.

3.3.4 Decentralized Naming Schemes

As mentioned above, the goal is to avoid a centralized naming scheme or authority. In particular, it is desired for each SubEnv to be able to name objects, rules, and new SubEnvs autonomously, yet still guarantee uniqueness. This is the subject of this section.

Object Ids

Distributed object naming schemes have been thoroughly investigated over the years, especially in the area of distributed databases (e.g., [30]). This is in general outside the scope of this thesis. However, an approach which fits the requirements of a DEPCE is presented below. A short survey about object naming schemes will be given in the thesis.

```
# Connection between a client, CL, and a remote server RS

if RS is local
then
   communicate
end

if RS is marked as ACTIVE
then
    try to connect directly using the dynamic host information
    if connection is successful
    then
        communicate
    end
end

# if we're here, either the dynamic information is out of date, or RS
# is marked as non-active.

connect indirectly using the site information.
if connection is successful
then
    get the active host information and update the local SubEnv object
    communicate
else
   return  FAILURE
end
```

Figure 17: Oz communication protocol

A desired property of an object id scheme should be the ability to extract an object's SubEnv from its id, so that a remote client can instantly identify and possibly contact the remote SubEnv. This feature is important to support high visibility yet reasonable performance. Experience with using Marvel has shown that objects are accessed frequently for browsing purposes. In addition, for autonomy purposes object id assignment should be performed solely by the objectbase manager of a single SubEnv.

In order to determine uniqueness and support fast addressing, each object is identified by the pair `(SubEnv_id, obj_id)` where the latter is determined locally, and the former relies on the unique SubEnv id (see below). An alternative approach would be to rely directly on the Internet naming scheme. In this case, the Internet address of the SubEnv's primary host could be chosen, since a site address does not guarantee uniqueness. However, it is possible (although not very likely) that multiple SubEnvs of the same global environment will reside in a given host, which prevents uniqueness. In addition this approach would violate the general approach of SubEnv-based addressing by binding fixed host information to an id.

SubEnv Naming scheme

The problem here is how to identify SubEnvs uniquely. SubEnv naming scheme is important since SubEnv objects are the vehicle for communicating across SubEnvs. As will be seen in 3.3.5, it should be possible for the registering server at a given SubEnv to autonomously assign a name and id for a new SubEnv. The approach taken here is simple: Relying on the premise that the SubEnv map is a replicated resource, adding a SubEnv is not much more than adding a normal object to a local objectbase, and is subject to the same constraints, namely: its name cannot be the same as any of its sibling objects (in this case, other SubEnvs), and its id, which is generated by the local objectbase, has to be unique within the local objectbase. Thus, only local considerations are taken. The SubEnv's user-level name is the SubEnv name given to it by the creator of the SUB_ENV object and its internal env_id is generated by the objectbase at the registering SubEnv exactly like any other object, except in this case the generated id is used for the `subenv_id` component of the object id.

One problem with this approach is that this scheme assumes only one SubEnv registration in an environment at a time. If two or more SubEnvs are added at a time (from multiple registering SubEnvs), a possible name clash might occur. However, if the registration process is implemented as an atomic operation, then since registration implies replication, multiple simultaneous registrations will result in a conflict by which only one registration will get through and all others will be aborted and rolled back[10]

Global Environment Naming Scheme

In order to uniquely name global environments without limiting (geographically or by other criteria) the possibility of any site to join an environment, a global assignment is in general unavoidable. On the other hand, it is reasonable to assume that certain sites which are not related to each other in any way, will never work together. In this case, the existence of multiple environments with same name does not present a problem. In addition, unlike SubEnvs, the global environment name space is used only at the system level, typically to enable the Connection Server to distinguish between multiple SubEnvs in a site, which can overlap in their names across environments.

In order to provide system-level uniqueness without imposing a central addressing scheme, the existing global Internet addressing scheme is used. (Note that this does not contradict the SubEnv-based addressing approach as described earlier, since the Internet names are used only for identifying *global* environments and not for identifying SubEnvs within an environment.) An environment id is a (global_id, local_id) pair, where global_id is the Internet site address of the creator site of an environment, and local_id is an id generated by the creating site, to distinguish between multiple global environments created in the same site. This way, although there maybe multiple environments with the same user-level name, they will be distinguished in their system-level environment id. Since each SUB_ENV object contains its global environment id, uniqueness is guaranteed at the system level.

3.3.5 Environment Configuration

The configuration of a global environment is dynamic, i.e., SubEnvs can be added or deleted dynamically. Environment configuration is supported by a *registration* process. Note that unlike other aspects of communication, where the "lazy" approach is preferred, the registration process must be preformed promptly and consistently. Registration of a new SubEnv is a two-step process, performed at an active server of an existing SubEnv. (As for creating the

[10]Ensuring atomicity of a distributed operation by two phase commit is assumed to be supported in Oz, and is in general outside the scope of this thesis.

first SubEnv of an environment a simple procedure can be applied which does not involve any remote interaction. It is omitted here.)

1. Using an active server on a SubEnv, the new SubEnv information is registered in a newly created SUB_ENV object, which is replicated in all other (active and inactive) SubEnvs.

2. The registering SubEnv initializes the newly established SubEnv by sending to it the SubEnv map.

A similar approach can be defined for removing a SubEnv from a global environment.

Note that step one implies that inactive servers get also notified and updated. This is done using the normal connection protocol as described in figure 17, since an inactive server can be activated by its connection server.

Registration is performed using the Oz process and data models, i.e., at the process level, in order to enable future tailoring on a per-environment basis. The relative detail in which the registration procedure is given below is intended to show the feasibility of the Oz process formalism to support configuration tasks. However, it is not a crucial part of the proposal, and can be skipped.

SubEnv registration is done using the `register_subenv` and `send_subenv_map` rules (shown in figure 18) which correspond to steps 1 and 2 above, respectively. Each of these rules invokes an envelope (not shown here) which internally invokes servers in batch mode to update their objectbases, by invoking the "hidden" rules (i.e., rules not accessible to end users) `init_remote_subenv` and `init_subenv_map` respectively, shown in figure 19.

4 Related Work

This section is divided into work that has been done directly in PCEs, and relevant work done in related fields, namely Heterogeneous Data Bases, Concurrent Engineering, and Computer Support for Collaborative Work.

4.1 Distributed and Decentralized PCEs

As noted in the introduction, to date there has not been much work on decentralization of PCEs. Thus, mainly theoretical work is presented. The work surveyed here can be categorized into two kinds: (1) "motivational" papers that realize the need for decentralization of PCEs and specify requirements for DEPCEs, and (2) few attempts to build systems which support some form of decentralization, distribution, and/or process inter-operation.

Shy, Taylor, and Osterweil [39] draw an analogy between software development and the corporation model, and advocate that the "federated decentralization" model is the most

```
#
# collect subenv_information about the new subenv,
# and replicate it in all current subenvs using a batch file
# consisting of adding the object and calling init_remote_subenv
#
register_subenv [?new_name:LITERAL]:

    # collect all remote SubEnv objects
   (and
    (forall GROUP ?se suchthat (?se.local = false))
    (exists GROUP ?lse suchthat (?lse.local = true)))
  :

# this envelope actually does the replication in remote subenvs
    { REGISTER register_subenv ?new_name ?se.subenv_name
              ?lse.subenv_name}
    no_assertion;

#
# initialize the newly created subenv, by sending to it the
# environment map.
#

send_subenv_map [?nse:GROUP]:

    # collect all SUB_ENV objects except the new one
    (forall GROUP ?s suchthat (?s.subenv_name <> ?nse.subenv_name))
    :

    # this envelope actually copies the EnvMap to the new SubEnv
    {
     REGISTER send_subenv_map
        # new subenv's identification and location
        ?nse.Name ?nse.env_name ?nse.env_id
        ?nse.subenv_id ?nse.subenv_name
        ?nse.site_name ?nse.site_ip_addr
        # all information of all subenvs.
        ?s.Name ?s.env_name ?s.env_id  ?s.subenv_id ?s.subenv_name
        ?s.site_name ?s.site_ip_addr
    }

    no_assertion;
```

Figure 18: Registration Rules

```
#
# called from within an envelope (in batch mode), from all
# remote SubEnvs, to assign the proper values to the objects which
# was just added by register_subenv
#

hide init_remote_subenv [?env_name:LITERAL, ?env_id:LITERAL,
            ?subenv_id:LITERAL, ?subenv_name:LITERAL,
    ?site_name:LITERAL, ?site_ip_addr:LITERAL]:

   (exists GROUP ?new suchthat (?new.state = New)):

    {}

    (and (?new.env_name = ?env_name)
         (?new.env_id  =  ?env_id)
         (?new.subenv_id  = ?subenv_id)
         (?new.subenv_name  = ?subenv_name)
         (?new.site_name = ?site_name)
         (?new.site_ip_addr  = ?site_ip_addr)
         (?new.local = false)
         (?new.state = Initialized));

#
# called from within an envelope (by invoking a batch client)
#  of send_subenv_map, this assigns to all objects of the SubEnv
#  map the proper values.
#

hide init_subenv_map [?new:GROUP, ?env_name:LITERAL, ?env_id:LITERAL,
            ?subenv_id:LITERAL, ?subenv_name:LITERAL,
    ?site_name:LITERAL, ?site_ip_addr:LITERAL, ?local:LITERAL]:
    :
    {}

    (and (?new.env_name = ?env_name)
         (?new.env_id  =  ?env_id)
         (?new.subenv_id  = ?subenv_id)
         (?new.subenv_name  = ?subenv_name)
         (?new.site_name = ?site_name)
         (?new.site_ip_addr  = ?site_ip_addr)
         (?new.local = ?local)
         (?new.state = Initialized));
```

Figure 19: Hidden Rules Invoked from Enveloeps

appropriate for software development environments – with global support for environment infrastructure capabilities, and local management with means to mediate relations between local processes. Among the justifications made in the paper for this model (as opposed to "corporate autocracy" or "radical decentralization") are:

- Level of global support is not rigid.
- While the communication is established under guidelines determined by the global process, the actual communication is provided and maintained under the control of the agents themselves.
- Extensibility - integration of processes and services can be implemented gradually.

Although it does not map directly to the Oz model (and it is impossible to draw an exact mapping because the model described in that paper is highly abstract), it is close enough to view it as a "philosophical" justification for the decentralized model presented in this proposal.

Heimbigner argues in [21] that just like databases, "environments will move to looser, federated, architectures ... address inter-operability between partial-environments of varying degrees of openness". He also notes that part of the reason for not adopting this approach until recently was due to the inadequacy of existing software process technology. However, his focus is on support for multiple formalisms, and in retro-fitting a process onto a process-ignorant environment. The *ProcessWall* [22] proposed by Heimbigner is an attempt to address one particular aspect of federation, namely: process formalism inter-operability. The main idea in the ProcessWall is the separation of process *state* from the *programs* to construct the state, so in theory, multiple process formalisms (e.g., procedural and rule-based) can co-exist and be used for writing fragments of a process. However, decentralization per se is not addressed, and in particular, the process state server is inherently centralized. A process server as a distinguished component of Oz that supports multiple formalisms is being investigated by Popovich [36] and complements the homogeneous process interoperability described in the proposal.

Peuschel and Wolf explain in [35] why current client-server architectures are not adequate to support distributed software processes, analyze the requirements for supporting such processes, and discuss possible architectural approaches to supporting them, with the intent to investigate a future distributed architecture for the Merlin PCE [16]. This work can also be regarded as motivational work.

Kernel/2r [25], from the Eureka Software Factory project, is an actual system that supports process inter-operation. The approach taken here is to identify and divide the global process into three distinguished kinds of process fragments, and support each with a separate process engine (and formalism). The process fragments and their associated engines are: (1) *interworking* process engine for cooperation between teams or within a team (using MELMAC [14]); (2) *interaction* process engine for a single-user workspace; and (3) a software bus for tool *interoperation*. Thus, unlike the ProcessWall approach, the inter-operation is not only at the logical level but at the physical, architectural level. But decentralization per se is not addressed in the architecture.

Oikos [1] is a rule-based PCE that supports distribution using a hierarchy of blackboards that resemble the tuple spaces in Linda [11]. Oikos enables to specify a wide range of services as part of process enactment, including database schemas and transactions. In addition, each blackboard can have a different sub-process, and in that sense, process decentralization is supported, although in a different way than in Oz .

4.2 Heterogeneous and Distributed DataBases (HDDB)

The relevance of work in HDDB to DEPCEs is as relevant as work in DBMS to PCEs. As most PCEs are essentially data-centered, they require special-purpose databases for management of persistent storage of the artifacts being developed [8] (also known as software engineering databases). Similarly, data-centered DEPCEs (including Oz) require mechanisms to support distributed and decentralized data, and therefore HDDB technology is of major importance.

Orion-2 [30] is the latest in the series of Orion databases developed at Microelectronics and Computer Technology Corp. (Itasca is a later commercial version based on the Orion systems). It is a distributed (but not heterogeneous) Object-Oriented Data Base (OODB) supporting private and shared databases. The main motivation for this design is to support design applications (e.g., CAD, SE) that involve long-duration activities. The idea is to provide private workspaces that are isolated from the shared area, and thus can be treated privately (e.g., to reduce concurrency conflicts), while supporting consistent references and migration from and to the shared area. However, Orion-2 is homogenous, in that it supports a single schema, which, as evidenced, limits the degree of freedom provided in the private area. Such an approach, if adopted by Oz, would preempt decentralization. Furthermore, it allows only for two modes, namely private and shared, and presupposes a check-out model for concurrency. This may not be suitable for more advanced cooperation models within and among teams.

Pegasus [17] from HP laboratories is an HDDB that supports integration of various database models (e.g., relational, hierarchical) with an object-oriented language called Heterogeneous Object SQL (HOSQL) that allows to define mappings from other schemas (local data sources) into a unified Pegasus data model by supporting "attachments" of actual databases to a Pegasus database using the schematic mappings. The approach for integration is based on the notion of "upward-inheritance", where types can be superimposed to generalize on local types in different schemas (and different databases). Pegasus integrates function and object-oriented programming techniques to enable mapping of any data model and schema into the Pegasus model by attaching optional functions that implement the mapping.

UniSQL\M [31] is an HDDB that assumes a common relational data model to which all component database systems must convert their schemas. The paper provides a complete framework for classifying schematic and data heterogeneity as a basis for a later "homogenization" of the databases. The general approach to homogeneity with an underlying common formalism is the one intended to be used in Oz for process as well as data integration. However, it is important to verify that the common model is powerful enough to support a wide variety of schemas, or ways to extend the common model. For example, it is not clear how to map an object-oriented data model with recursive definition of composite objects into a relational data model. Similarly, the common *process* model has to be sufficiently expressible to support various process modeling formalisms, and ways to extend the common model when needed. As mentioned earlier, this problem is currently studied by Popovich.

4.3 Concurrent Engineering (CE)

CE is concerned with computer support for engineering tasks that require collaboration of different groups which are possibly geographically-dispersed and often with different perspectives on the product. The main focus is on providing proper support for concurrency, in particular with respect to access to data by tools. CE is relevant to PCEs and (particularly) DEPCEs in that DEPCE requires cooperative tools and mechanisms for implementation of the functionality as specified in a multi-user decentralized process.

Distributed & Integrated environment for Computer-aided Engineering (DICE) [41] is used to integrate multiple engineering disciplines and applications. It is based on the idea of a logical blackboard as a shared workspace. As such, it does not support the issue of autonomy. It supports collaboration and heterogeneity at the application level, but not at the data level.

FLexible Environment for Collaborative Software Engineering (Flecse) [15] is an environment intended to support product development of a group of geographically-dispersed engineers, focusing on multi-user tools for editing, debugging, and versioning. The idea is to provide a local user interface front-end and cached state of the tool, and maintain the actual tool's data in a central location.

4.4 Computer Support for Collaborative Work (CSCW)

CSCW is focusing on support for human-to-human interaction (as opposed to human-tools, and human-computer interaction, which is the main focus in CE), including social and psychological aspects of collaboration and tools to enhance it.

Conversation Builder (CB) [18] is a CSCW system that supports collaborative software development. The main concept in CB is that of a "conversation", which is a context in which a user performs its actions ("utterances"), and can potentially affect other users participating in the same conversation through a shared conversation space yet still protect their private conversation space. However, the CB architecture is centralized, and all client processes are communicating with a shared conversation engine. The underlying bus architecture of CB allows for componentized flavor, and can be potentially integrated within the framework of Oz . This is currently under investigation both in the Oz and the CB groups.

Media Spaces [9] - A multi-media project at Xerox PARC, that has been in use for several years and connected two sites. It was intended to support "virtual-reality" (i.e., provide an environment whereby people that are physically separated can feel and operate as if they were co-located). Their specific approach to reaching virtual reality was on providing non-activity-specific oriented environment (such as "chance-encounters"), in addition to the standard activity-oriented support (e.g., video conferencing). The relevance of this work to the proposal is mainly in that at some point, DEPCEs will have to consider integration of such technology.

5 Contribution and Evaluation

The main contributions of this work are in:

- Identifying the requirements for decentralization of PCEs;
- Designing a model for *process*, as well as data decentralization, both for process modeling and for process enaction, with emphasis on autonomy and collaboration and on flexible control of those properties;
- Realizing the model with an architecture that supports the model;
- Building an actual working system as a proof of the concepts mentioned above, and for evaluation of them.

As PCE decentralization is in its first steps, the innovation in this work is primarily with the pioneering work in building such a system, and with devising a model and an architecture for a DEPCE.

The evaluation of the system will be mainly done against the requirements and its intended goals. In addition, with lack of benchmarks for software processes in general (except the "ISPW" problem [23]) and for decentralized processes in particular, metrics for evaluating the system would have to be defined. As part of the evaluation, the ISPW problem will be extended to cover cooperation among teams (as opposed to cooperation within team).

It is expected that the following will be achieved in the thesis:

- A complete theoretical framework and design for Oz, covering logical and physical decentralization.
- A working implementation of Oz supporting logical decentralization, and some aspects of physical decentralization (see below).
- A sample decentralized collaborative process based on extensions to the ISPW example will be:
  - defined (as explained above)
  - modeled and enacted
  - evaluated

6 Evidence of Feasibility and Research Plan

6.1 Evidence

First and foremost, Marvel 3.1, the latest version in a series of releases, is a complete working PCE that has been reliably used for over a year by the Marvel group (supporting the development of Marvel itself) and by other external groups, and is well recognized in the PCE community. Marvel's multi-user architecture (which was designed by the author in his MS thesis [6]) is perhaps the strongest evidence that Oz is likely to succeed, since the Oz architecture is founded on Marvel's, although it extends it significantly.

Another past work that is directly applicable to Oz is process evolution, which, as described in section 3.2.2, is instrumental in providing inter-process consistency, although some extensions are likely to be made in it.

Work that has been already done directly in the Oz project so far, includes:

- Preliminary work on the communication infra-structure for server to server and server to remote-client communication as described in section 3.3.3
- Client connections to multiple servers for viewing purposes, and a per-objectbase refresh policy[11].
- The registration sub-process, as described in section 3.3.5, has been implemented using MSL rules and batch invocations of clients, including support for "re-registration" of an existing process, and for de-registration from an environment. It was also an exercise in testing the feasibility of the Oz process formalism to support such operations (and revealed some necessary extensions which are not dealt with here).
- Preliminary work towards componentization was initiated by the Oz project as a whole including writing a special process for development of componentized systems and re-engineering of the Marvel code. In addition, preliminary work on a process for support of DEPCE construction is underway.

6.2 Plan

This is a proposal for a work-plan for Oz. There will be three stages of development:

Oz 0.1 - a preliminary version that will include infra-structure for inter-SubEnv communication, initial decentralized object management (but still with a single schema), and a client user interface for browsing multiple SubEnvs. In addition there will be support for decentralized process modeling (i.e., implementing CPM) but without decentralized process execution and requiring single-schema and single-process. Note that some of the above have already been implemented.

[11] This work is being implemented by Y.S.Kim and is near completion.

Oz 0.5 - will include heterogeneous data and process models, decentralized rule processing implementing CSP, including chaining across SubEnvs, simple decentralized query evaluation, and integration with transaction support (Heineman).

Oz 1.0 - will include architectural support for physical decentralization.

In addition the thesis will include a full *design* of "distance" metrics (as defined in section 2.1) and will address in general the "Variable Bandwidth" problem, and optimization of decentralized query evaluation.

Version 1.0 is intended to be the first operational version of Oz, and the final version associated with this thesis.

6.3 Schedule

The research and its associated implementation are expected to take roughly one year, collaborating with other members of the Oz project and with help from project students.

The following table summarizes the research plan and times.

Date	*Phase*	*Description*
August '93	Oz **0.1**	Communication, Decentralized Data
January '94	Oz **0.5**	Logical Decentralization
May '94	Oz **1.0**	Limited Physical Decentralization
August '94	**Defense**	Written Dissertation

The milestones in the schedule represent, in addition to implementation phases, the stages in the development of the theoretical framework, evaluations of the model and the architecture, and writing of the dissertation.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 183–192, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[2] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments.* PhD thesis, Columbia University, February 1992. CUCS-001-92.

[3] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[4] Naser S. Barghouti and Gail E. Kaiser. Modeling concurrency in rule-based development environments. *IEEE Expert*, 5(6):15–27, December 1990.

[5] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. In A. van Lamsweerde and A. Fugetta, editors, *3rd European Software Engineering Conference*, volume 550 of *Lecture Notes in Computer Science*, pages 380–395, Milano, Italy, October 1991. Springer-Verlag.

[6] Israel Z. Ben-Shaul. An object management system for multi-user programming environments. Master's thesis, Columbia University, Department of Computer Science, April 1991.

[7] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149–158, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[8] Philip A. Bernstein. Database system support for software engineering. In *9th International Conference on Software Engineering*, pages 166–178, Monterey CA, March 1987.

[9] Sara A. Bly, Steve R. Harrison, and Susan Irwin. Media spaces: Bringing people together in a video, audio, and computing environment. *Communications of the ACM*, 36(1):28–47, January 1993.

[10] Maryse Bourdon. Building process models using PROCESS WEAVER: a progressive approach. In Wilhelm Schafer, editor, *Reprints of the 8th International Software Process Workshop*, Schloss Dagstuhl, Wadern, Germany, March 1993.

[11] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[12] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases*. McGraw Hill, 1985.

[13] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[14] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment MELMAC. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 193–205, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[15] Prasun Dewan and John Riedl. Toward computer-supported concurrent software engineering. *Computer*, 26(1):17–36, January 1993.

[16] W. Emmerich, G. Junkermann, B. Peuschel, W. Schafer, and S. Wolf. Merlin: Knowledge-based process modeling. In *1st European Workshop on Software Process Modeling*, pages 181–187, Milan, Italy, May 1991. AICA.

[17] R. Ahmed et al. The Pegasus heterogenous multidatabase system. *Computer*, 24(12):19–27, December 1991.

[18] Simon M. Kaplan et al. Supporting collaborative software developement with conversation builder. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 11–20, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[19] G. Forte and R.J. Norman. A self assessment by the software engineering community. *Communications of the ACM*, 35(4):29–32, April 1992.

[20] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[21] Dennis Heimbigner. A federated architecture for envrionments: Take II. In *Preprints of the Process Sensitive SEE Architectures Workshop*, Boulder CO, September 1992.

[22] Dennis Heimbigner. The ProcessWall: A process state server approach to process programming. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 159–168, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[23] Dennis Heimbigner and Marc Kellner. Software process example for ISPW-7, August 1991. /pub/cs/techreports/ISPW7/ispw7.ex.ps.Z available by anonymous ftp from ftp.cs.colorado.edu.

[24] 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.

[25] Bernhard Holtkamp. Process engine interoperation in PSEEs. In *Preprints of the Process Sensitive SEE Architectures Workshop*, Boulder CO, September 1992.

[26] R. Kadia. Issues encountered in building a flexible software development environment. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[27] Gail E. Kaiser, Israel Z. Ben-Shaul, George T. Heineman, and Wilfredo Marrero. Process evolution for the MARVEL environment. Technical Report CUCS-047-92, Columbia University Department of Computer Science, April 1993. Submitted for publication.

[28] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In *15th International Conference on Software Engineering*, Baltimore MD, May 1993. IEEE Computer Society Press. In press. Available as Columbia University Department of Computer Science, CUCS-016-91, September 1992.

[29] Takuya Katayama, editor. *6th International Software Process Workshop: Support for the Software Process*, Hakodate, Japan, October 1990. IEEE Computer Society Press.

[30] Won Kim, Nat Ballou, Jorge F. Garza, and Darrel Woelk. A distributed object-oriented database system suporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.

[31] Won Kim and Jungyun Seo. Classifying schematic and data heterogeneity in multidatabase systems. *Computer*, 24(12):12–18, December 1991.

[32] Programming Systems Laboratory. Marvel 3.1 Administrator's manual. Technical Report CUCS-009-93, Columbia University Department of Computer Science, March 1993.

[33] Jacky Estublier Noureddine Belkhatir and Walcelio L. Melo. Adele 2: A support to large software development process. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 159–170, Redondo Beach CA, October 1991. IEEE Computer Society.

[34] Maria H. Penedo and William Riddle. Process-sensitive SEE architecture (PSEA) workshop summary. In *ACM SIGSOFT Software Engineering Notes*, Boulder CO, April 1993.

[35] Burkhard Peuschel and Stefan Wolf. Architectural support for distributed process centered software development environments. In *Eighth International Software Process Workshop*, Schloss Dagstuhl, Wadern, Germany, March 1993. Position paper.

[36] Steven S. Popovich. Rule-based process servers for software development environments. In *1992 Centre for Advanced Studies Conference*, volume I, pages 477–497, Toronto ON, Canada, November 1992. IBM Canada Ltd. Laboratory.

[37] CLF Project. *CLF Manual.* USC Information Sciences Institute, January 1988.

[38] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1:364–370, 1975.

[39] Izhar Shy, Richard Taylor, and Leon Osterweil. A metaphor and a conceptual architecture for software development enviornments. In *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 77–97, Chinon, France, September 1989. Springer-Verlag.

[40] Nandit Soparkar, Henry F. Korth, and Abraham Silberschatz. Failure-resilient transaction management in multidatabases. *Computer*, 24(12):28–36, December 1991.

[41] D. Sriram. Computer-aided collaborative product development. Technical Report IESL-91-05 IESL-92-02, Massachusetts Institute of Technology, September 1992.

[42] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, 1990.

[43] Carl Tait and Dan Duchamp. Service interface and replica management algorithm for mobile file system clients. In *IEEE First International Conference on Parallel and Distributed Information Systems*, pages 190–197, December 1991.

[44] Richard N. Taylor, Richard W. Selby, Michal Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf. Foundations for the Arcadia environment architecture. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston MA, November 1988. Special issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.

[45] Ian Thomas, editor. *7th International Software Process Workshop*, Yountville CA, October 1991. IEEE Computer Society.

[46] Walter F. Tichy. RCS — a system for version control. *Software — Practice & Experience*, 15(7):637–654, July 1985.

[47] Wilhelm Schäfer, Burkhard Peuschel and Stefan Wolf. A knowledge-based software development environment supporting cooperative work. *International Journal on Software Engineering & Knowledge Engineering*, 2(1):79–106, March 1992.