Architectures for Federation of Process-Centered Environments

Israel Z. Ben-Shaul
Technion-Israel Institute of Technology
Department of Electrical Engineering
Technion City, Haifa 32000
ISRAEL
issy@ee.technion.ac.il
phone +972-4-8294680
fax +972-4-8323041

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
UNITED STATES
kaiser@cs.columbia.edu
212-939-7081
212-666-0140

CUCS-015-96
May 6, 1996

Abstract

We describe two models for federating process-centered environments, homogeneous federation where the interoperability is among distinct process models enacted by different copies of the same system and heterogeneous federation with interoperability among distinct process enactment systems. We identify the requirements and possible architectures for each model. The bulk of the paper presents the specific architecture and infrastructure for homogeneous federation we realized in the Oz *system. We briefly consider how* Oz *might be integrated into a heterogeneous federation to serve as one of its interoperating PCEs.*

Keywords: Collaborative work, Distributed system, Enterprise-wide process, Geographical distribution, Internet, Software process, Workflow management

This paper is based on work sponsored in part by Advanced Research Project Agency under ARPA Order B128 monitored by Air Force Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, and in part by the New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare NYSSTF-CAT-95013. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US or NYS government, ARPA, Air Force, NSF, or NYSSTF.

1 Introduction

Large-scale software engineering projects are not always confined to a single organization (e.g., group, department or lab), or even to a single institution (e.g., in a subcontracting or consortium relationship). A project may span multiple teams that are geographically dispersed across a wide area network such as a corporate intranet or the Internet. Collaborating teams may each have their own software development practices, favored tools, use different programming languages, etc.

There is a spectrum of approaches to software development environment (SDE) support for such projects. At one end, each team chooses its own SDE and there may be more or less concern with whether the different team's environments are compatible with each other. A little further along the spectrum, the teams might agree to use the same SDE, to minimize data conversion and supply a common vocabulary, but they still run entirely independent instances of the SDE. In both cases all sharing and collaboration between teams is done *outside* the environment — unless some special "glue" is added on top to bind them together.

Another intermediate range is covered when the teams share the same instance of what we call a *multi-site* SDE, which distinguishes among teams (who may reside at the same or different Internet sites) in some way, but provides facilities for sharing and collaboration between teams *inside* the environment. That is, the "glue" (or perhaps "cement" in this case) is conceptually part of the environment itself. The degree of independence afforded each team determines the point within the subrange. Finally, the far extreme is a geographically distributed SDE that does not distinguish among teams — all the users are treated as members of one very large team sharing everything. We choose the terms "multi-site" and "geographically distributed" here because many SDEs are said to be "distributed", meaning they have multiple internal components that may execute on different hosts.[1]

The geographically distributed SDE end of the spectrum is analogous to a distributed database system, where there is transparent access to distributed data, while the independent choice of SDE end is comparable to a set of independent databases. The database community has also delineated intermediate points, often termed federated databases [57, 54]. Federated databases generally permit a high degree of autonomy with respect to one or both of two criteria, schema and system: local components of a single database system may devise and administer their own schema independently (known as a *homogeneous federation*), and/or the local components may correspond to different database systems from among those supported by external federation "glue" (*heterogeneous federation*) — in which case even conceptually equivalent schemas may appear in different forms due to system-specific data definition languages.

There has been some work towards federation of SDEs. We are concerned in this paper with the subclass of SDEs known as *process-centered environments* (PCEs) [49, 23], and do not address SDE federations that do not specifically support process. In general, a PCE is a generic environment kernel intended to be parameterized by a *process model* that defines the software development process for a specific instance of the environment. The PCE's *process engine* interprets, executes or "enacts" the defined process, to assist the users in carrying out the process by guiding them from one step to another, enforcing the constraints and implications of process steps as well as any sequencing or synchronization requirements, and/or automating portions of the process. A federated PCE might coordinate users from multiple teams working on collaborative tasks, inform

[1] In previous papers we have used the terms "multi-site" and "geographically distributed" interchangeably, but here they refer to different concepts.

Figure 1: Multiple Team SDE Spectrum

one team when it should perform some task on behalf of another, notify one team on completion of some task it has been waiting for another to perform, and transfer process state and product artifacts (design documents, source code, executables, test cases, etc.) among local components of the federation as needed for this work.

A multi-site PCE is analogous to a homogeneous database federation. In particular, the PCE process model fills the role of the database schema with respect to homogeneous federation: the local components of the multi-site PCE are identical, except that they are tailored by and thus enact different process models (or possibly reflect different instantiations of the same process model). Separate PCEs bound together by external "glue" are analogous to database systems participating in a heterogeneous federation. These two notions of federated PCEs are superimposed on the SDE spectrum in Figure 1.

One approach to homogeneous PCE federation, where every team runs a component of the same multi-site PCE but enacts a different process, is taken by our Oz PCE [14, 9]. Oz introduces an *International Alliance* metaphor whereby each team retains its own local process, analogous to how each country has its own local customs and laws. A team may agree to extend its process to a small degree (and thus temporarily lose some autonomy) in order to participate in a *Treaty* with one or more other teams. The enactment of a *multi-site* task, defined as any task that involves interaction among the several "sites" of a multi-site PCE, is called a *Summit*. Only tasks specified in the Treaties may access data from other sites, and even then only in accordance with the privileges granted by the Treaty. For example, a site may agree to perform certain tasks requested by another site on its own local data; a site may agree to perform certain tasks on data from several sites; or a site may agree to allow another site to perform certain tasks on its local data. However, each site (or team) is responsible for any prerequisites or consequences of such tasks with respect to its own data, following its own process, just as in preparations for and follow-ups of meetings among country leaders.

One approach to heterogeneous PCE federation, where external "glue" binds together two or more PCEs, is taken by Heimbigner's ProcessWall [25]. Heimbigner refers to ProcessWall as a *process state server* because it enables interoperation between the PCEs through a (conceptually) centralized representation of global process state that the teams agree to share. The mechanism Heimbigner describes could alternatively be viewed as a *process task server*: it maintains the history of tasks that have already been completed, in aggregate representing the current process state, but more significantly from the viewpoint of federation the server posts those tasks that have been instantiated but not yet scheduled for enactment by one of the participating PCEs. A scheduler might be attached to ProcessWall to direct tasks to particular "sites" [48], or ProcessWall might be treated as a "blackboard" (using artificial intelligence terminology) from which the schedulers of the individual PCEs select the tasks they intend to perform or to which they delegate those pending tasks that cannot be performed locally. Any sharing of product artifacts, as opposed to process state, is implicit in the data information included with posted tasks. As in Oz, each site might autonomously devise its own process model.

Mentor [70] is similar to ProcessWall but divides the process state/task server into two components: a *worklist manager* acting as a pure task server and a *history manager* corresponding to a pure state server; data sharing is factored out as in ProcessWall. Note Mentor is a workflow management system intended for business applications, not a process-centered environment oriented towards software engineering; whether there is any fundamental difference between workflow and process is a matter of some debate,[2] but we blur the distinction in this paper.

Common *process interchange formats* [45, 43] support translation of a logically single process model into the different representations of distinct process systems, but do not provide any means for collaboration and interoperability during the process enactment by those systems. Thus there is no true federation in the sense addressed by this paper. However, some kind of translation facilities are needed as part of any heterogeneous federation: Mentor transforms the heterogeneous process modeling formalisms into StateMate charts [32], but in the case of ProcessWall only process state is translated (or the participating PCEs might be implemented to use a common task format).

We mentioned above that process enactment by a federated PCE might involve movement of product artifacts among teams that could potentially be geographically dispersed. Alternatively, all the sites might share a common centralized data repository, presumably located at one of the sites (or the teams might themselves reside on the same local area network together with the data), or even a distributed data repository dispersed among the sites (e.g., via a World Wide Web intranet). Globally shared data seems most appropriate for projects organized far in advance and involving only a single institution, perhaps with multiple campuses. In contrast, when different institutions work together, particularly when the federations are dynamically created and dissolved (i.e., the "glue" is relatively fluid), most likely the institutions would prefer to maintain locally at least those product artifacts produced by their local process.

Database access by components of a federated PCE is one of several architectural issues of interest here. The focus of this paper is on the *architectural aspects of PCE federation and associated infrastructures for multi-site process enactment.* That is, our investigation of architecture is strongly influenced by the fact that the main purpose of PCE federation is to enact multi-site processes. For example, multi-site processes devised using a top-down methodology, say intended for multiple campuses of a single institution [58], may require somewhat different architectural support than multi-site processes constructed in a bottom-up manner, e.g., for temporary multi-institution collaborations [15]. However, we do not propose or compare/contrast methodologies for developing multi-site processes in this paper.

First we present architectural requirements and alternative architectural models for homogeneous PCE federations. We then elaborate in great detail the design decisions and tradeoffs that were made in developing the Oz architecture and infrastructure for a multi-site PCE (which have not appeared in any previous papers on Oz). This is followed by requirements and architectural models for heterogeneous PCE federations. We do not go into detail regarding ProcessWall, Mentor, or any other such systems, since that is properly left to their developers. Instead we briefly discuss how Oz might be integrated into a heterogeneous federation based on the process state/task server model, to some extent synergizing the two federation mechanisms. We conclude with the contributions of this paper and outline some directions for future research.

[2] Both ARPA and NSF recently sponsored workshops on this topic.

2 Homogeneous Federation

2.1 Requirements

In the homogeneous model, each local site runs a component of a multi-site PCE. We refer to a site component as a sub-environment, or just SubEnv, and the "glue" that holds them together as the federation's foundation, or just Foundation. Recall that we are mainly concerned with multiple sites on a wide area network, generally with independent administrative domains — although of course nothing prevents multiple sites from running on the same LAN, that is, a multi-site PCE might operate entirely within a single organization or group and each "team" could conceivably consist of only one user. We take as given the requirements that each site must be able to support an autonomously devised process model, but also interoperate and collaborate with other sites in decentralized fashion, since this is the problem definition.

We have identified the following additional requirements:

- The most fundamental functional requirement for multi-site process enactment is that the Foundation include some infrastructure whereby the SubEnvs communicate with each other. This might be constructed directly on top of TCP/IP sockets, or employ some higher level mechanism such as RPC or CORBA [61]. Note the SubEnvs might communicate either directly with each other, or through some PCE-cognizant intermediary(ies) provided by the Foundation. Here we treat communication through a layer like CORBA as direct, since it is not PCE-cognizant.

- Another basic requirement is that as far as local work is concerned, a SubEnv should operate independently and provide the same capabilities and same support as would a single-site PCE. It should not in any way rely on communication with other SubEnvs, or with the Foundation, in order to perform its standard functions with regards to defining and executing the local process. The underlying assumption is that most of the work done by a site is local to that site, and therefore the multi-site PCE should still be optimized towards local work. A related issue is that the SubEnv should minimize the dependencies on uninvolved SubEnvs when executing part of a multi-site task. These two requirements are somewhat similar to control and execution autonomy, respectively, in multi-database transaction management [40]. The local site autonomy prized in the Oz approach to bottom-up process modeling has also been argued as necessary for top-down modeling: "A participant on a lower level [of the hierarchy] does not want his/her management to know how a task is performed" [56]. Thus we rationalize site autonomy as a critical requirement.

- The SubEnvs must somehow be aware a priori (statically), or become aware during the course of process enactment (dynamically), of each other's existence if they are intended to collaborate. In the case where a Foundation intermediary is the conduit for all communication and interactions among SubEnvs, the SubEnvs must at least be aware of that intermediary and vice versa. Furthermore, since the lifetime of enacted processes is often long, the federation must allow for SubEnvs to join or leave a federation during process enactment, that is, support configuration and reconfiguration. It is of course also necessary for SubEnvs to determine or negotiate what services each can expect from other (perhaps anonymous) SubEnvs in terms of process tasks and resources, and how to coordinate exploitation of those services, but that is the subject of another paper [15].

- The Foundation must provide some means by which the SubEnvs establish communication paths. Perhaps a given SubEnv always runs on the same machine and receives messages on the same port, but this seems overly restrictive — and unlikely when the SubEnvs run in independent administrative domains. However, a single-machine/single-port approach might not be unreasonable for a centralized intermediary, since that administrative domain has necessarily agreed to host the intermediary and the consequent PCE traffic.

- Either every SubEnv must be perpetually executing, which seems generally undesirable because of the draw on operating system resources, or the Foundation needs to support a facility for "bringing up" a SubEnv when its services are required by another one. Note this is distinct from "bringing up" an *internal* (sub)component of a SubEnv when it is needed by another internal (sub)component, although in practice the same mechanism might be used.

- While not strictly a requirement, the communication infrastructure of the Foundation should preferably support *asynchronous* communication since two SubEnvs may attempt to contact each other at the same time and *synchronous* communication in that case is likely to lead to deadlock (or, alternatively, a SubEnv and the centralized intermediary may attempt to send requests to each other at the same time, with the same result). Distributed deadlock detection and resolution is complicated, and best avoided. However, a purely synchronous Foundation can prevent deadlock from occurring by restricting a SubEnv from sending a message if it may soon be the recipient of another message, through token passing or some PCE-specific means.

- In cases where each PCE manages its own data repository, the Foundation must provide mechanisms for transferring product artifacts as well as process state among sites. For example, in a multi-site `build` activity one site may collect code modules from the other relevant sites and return to them copies of the resulting executables and/or libraries. A different example is a distributed groupware activity such as multi-user editing, in which files stored at one site may need to be (simultaneously) transferred to several other sites. In general, data may be temporarily cached, permanently copied, or migrated between sites.

- SDEs require sophisticated and flexible concurrency control and failure recovery mechanisms due to the long duration of tasks and task segments, interactive control by users, and collaboration among tasks and task segments while they are in progress [18]. The explicitness of the process in PCEs makes it possible to employ semantics-based transaction management [8, 6, 30]. Multi-site tasks that may modify remote data (i.e., data from sites other than the coordinating PCE) require some kind of global transactional support, such as two-phase commit. This topic is beyond the scope of this paper, see [11, 29, 27].

- There must be a (preferably graphical) user interface that allows users to browse and select objects from multiple SubEnvs while preserving each SubEnv's privacy with respect to which of its data may be viewed. An *administrator* interface should also be provided to define and interconnect SubEnvs.

- Last but not least, the multi-site PCE should provide means to enable modeling and tailoring of the infrastructure itself on a per-project basis, in analogy to single-site process modeling.

2.2 Architectural Issues

Although the focus of this paper is on *federation* architectures, it is useful to begin the discussion with an overview of SubEnv *internal* architectures, since they have a substantial impact on the design of the federation. Local PCEs can be roughly characterized as belonging to one of four classes with respect to process enactment:

1. Centralized process enactment and centralized task execution. An all-in-one single-user PCE such as Marvel 2.x [35] or a compiled process program such as those written in APPL/A [62] would presumably fit into this class. Even a client/server system might belong if the client supported only the user interface and all process enactment was performed in the server.

2. Centralized process enactment and decentralized task execution: A process server maintains the state of the process, controls its enactment, and synchronizes access to shared resources, but the tasks themselves execute at process clients. Marvel 3.x [17], ProcessWEAVER [22], and Mentor [70] fit this mold, albeit in different ways. Marvel (the single-site predecessor of Oz) relies on fixed user clients to execute tasks, whereas ProcessWEAVER spawns user "work contexts" as needed by the process. Oz sites are somewhere in between, with one server per site (i.e., per team), generally employing user clients inherited from Marvel but also supporting "proxy clients" that fork tools on behalf of one or more users under various circumstances [66]. Mentor is similar to Oz in that user clients can connect to multiple servers in the federation.

3. Decentralized process enactment and centralized task execution. At first glance this model does not seem very useful, but it would support sharing of special computational or database resources for task execution. For example, World Wide Web-based gateways from arbitrary Web browsers to numerous backend databases have already been developed — Microsoft recently bought out Aspect Software Engineering, acquiring their dbWeb gateway to databases meeting the Open Database Connectivity standard [46]. And workflow vendors are also rushing to the Web, e.g., Action Workflow Metro from Action Technologies [64]. One can easily imagine multiple workflow engines accessing the same task execution resource — particularly if only the task execution broker is centralized, directing tasks to distributed hosts as in WebMake [3]. We note that the Web might serve as a desirable infrastructure for any of the decentralized classes outlined here, but further elaboration is beyond the scope of this work.

4. Decentralized process enactment and decentralized task execution. Merlin [51] and several transactional workflow systems, such as Exotica [1], operate in a fully distributed manner — the former by divying up the process among user clients and the latter by expressing the workflow implicitly in a network of task managers that interact only with their predecessors and successors in the workflow routing.

The choice of local PCE architecture is influenced by several issues. For example, PCEs with data-integration facilities for the product artifacts (e.g., SPADE [4], EPOS [20]) might be likely to choose architecture 2 to minimize communication overhead between the data and process managers, whereas PCEs with no data-integration facilities might be fully distributed (architecture 4) in an easier manner — although note that full distribution of process enactment is not incompatible with sharing a centralized data repository; see [51].

Another characteristic that impacts the choice of architecture is whether the process modeling paradigm employed by the PCE is reactive or proactive. Reactive enactment may be realized

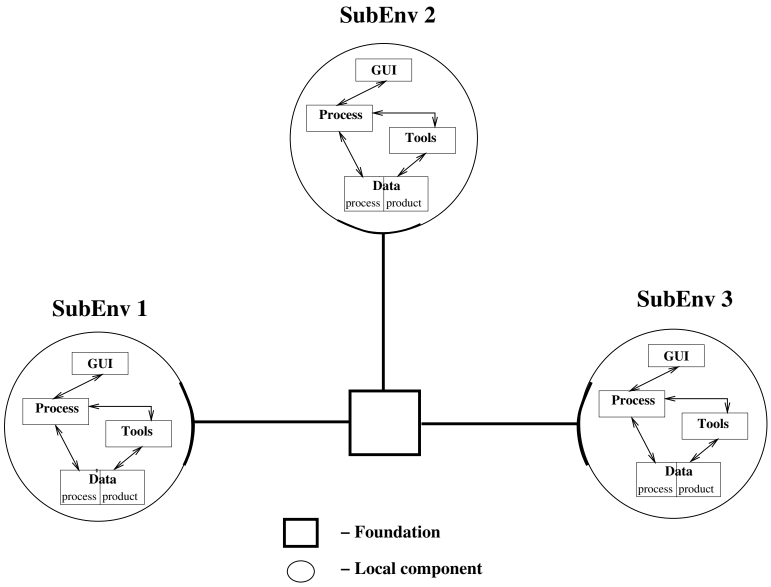

Figure 2: Centralized Glue Architecture

better in architecture 2 as requests for enactment are directed to a server that dispatches the service to a client (perhaps the requester itself), whereas proactive enactment may be distributed by assigning each task to a component, with the ordering and execution constraints inside each component.

Returning to federated PCEs, we identify four major categories of architectures:

- **Centralized glue:** The SubEnvs communicate through a single centralized component that implements the Foundation's major functions. Since we are concerned here with *homogeneous* federation, the federation "glue" is inherently part of the PCE rather than imposed externally. No translation services are needed, since all the SubEnvs speak the same "language" (including data formats and process modeling language). The Foundation may perform brokerage or routing among SubEnvs, and maintain the state of any "superworkflow" [56]. Figure 2 illustrates this architecture. For the sake of the example we show one plausible set of components that may comprise a local SubEnv (process, data, tool, and user interface components), but we do not intend to in any way specify or constrain a SubEnv to follow the given structure. In particular, process state and enactment may be managed by several distributed components, as opposed to the single centralized process server shown in the figure. We require only that the same SubEnv (and hence same structure, but not same process) resides in all collaborating sites.

- **Distributed glue:** The SubEnvs communicate through intermediaries, which in turn employ the Foundation's infrastructure to reach each other. There is one intermediary attached to each SubEnv. The intermediaries operate on behalf of, and are strongly affiliated with, the Foundation, and have relatively limited knowledge of their associated SubEnv's process-based concerns. See Figure 3. Such an intermediary would typically be implemented as a separate operating system process but reside on the same local area network where the local SubEnv is located.

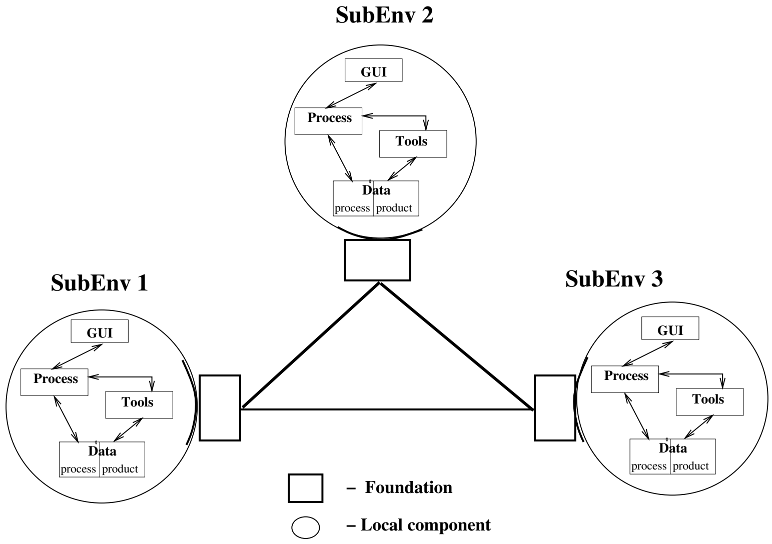

Figure 3: Distributed Glue Architecture

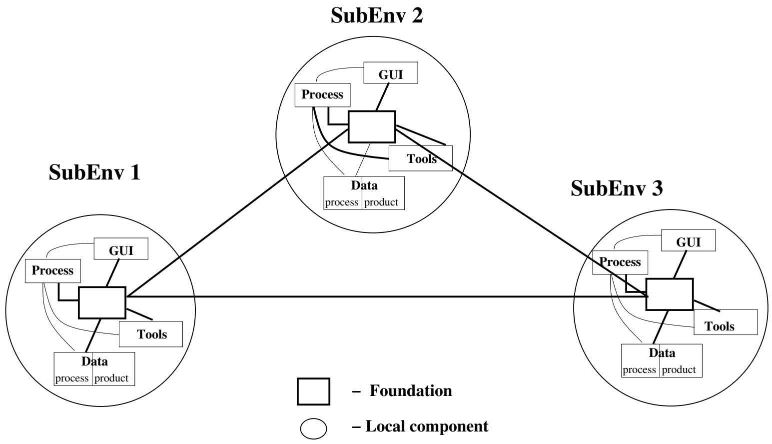

Figure 4: Moderated-peer-to-peer Architecture

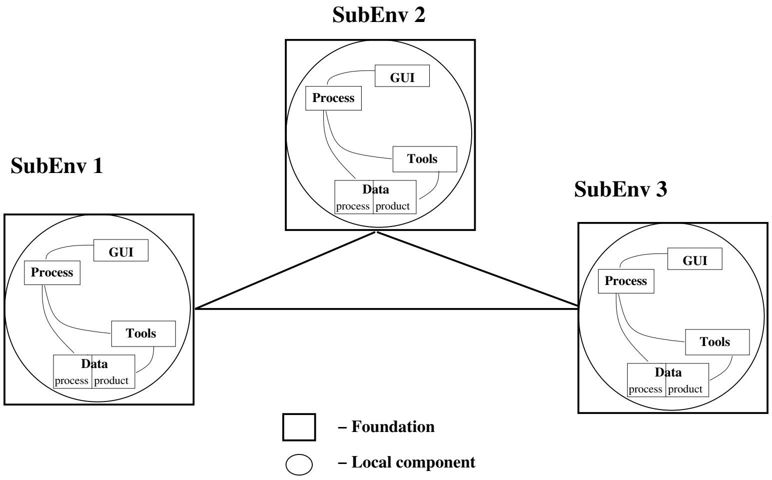

Figure 5: Direct Peer-to-peer Architecture

- **Moderated peer-to-peer:** The SubEnvs again communicate through intermediaries with one intermediary attached to each SubEnv, but here each intermediary is tightly coupled with its local SubEnv and has intimate knowledge of that SubEnv's process-based expectations regarding services from other SubEnvs. On the other hand, the intermediary is loosely coupled with respect to its peer intermediaries. The implication on the architecture is that it cannot assume any physical shared resources or capabilities (e.g., name servers) provided by the Foundation. In other words, it is a "shared nothing" architecture as far as the Foundation is concerned (note there may be sharing among internal components of the SubEnv). See Figure 4.

- **Direct peer-to-peer:** The SubEnvs communicate with each other directly via the Foundation's communication infrastructure. From the global perspective, the SubEnv is treated conceptually as one unit, although in practice there may be a single internal component that represents the SubEnv's process state and enactment functionality and manages the interaction with other SubEnvs. The main distinction from moderated-peer-to-peer is here the intermediary is *built into* one (or more) of the SubEnv's internal components; there is no distinguished component introduced *solely* to support interconnection with the Foundation. Note we are concerned here with the actual communication, not the establishment of communication paths, which may involve additional architectural components to assist in setting up the paths. See Figure 5.

The choice of federation architecture depends largely on two concerns:

1. The anticipated global process style.

2. The type of local SubEnv architecture as outlined above.

Regarding global process, we distinguish between two major styles, top-down and bottom-up, although of course hybrids are possible. *Top-down* refers to a process, probably hierarchical as in the Corporation metaphor [58], imposed on the local SubEnvs; this is analogous to a global transaction in federated databases. *Bottom-up* refers to interoperability among possibly pre-existing local processes, as in Oz's International Alliance metaphor, without a global overseer. We do not discuss here which of the two styles is more appropriate (see [15] for such a discussion), but rather which *architecture* best supports each of the styles.

In order to support top-down global processes, the underlying federation architecture must support maintenance of some global process state. This suggests a "glue" architecture, either centralized or decentralized, that is responsible for this state. In contrast, bottom-up federation can be naturally realized on top of a peer-to-peer architectural style, again in one of two possible ways, namely the moderated- or direct- peer-to-peer architectures we identify above. In other words, we make a primary distinction between top-down vs. bottom-up process interoperability style, and a secondary distinction between the physical realization of each style.

The association of top-down processes with "glue" added onto local PCE architectures and of bottom-up processes with peer-to-peer multi-site architectures is not exclusive, however. It is potentially feasible, for example, to realize a global process using a peer-to-peer architecture, but it is likely to be inefficient and harder to realize because of the needs to distribute the global process state among the loosely coupled intermediaries and to manage shared resources over a shared-nothing architecture. It is probably easier to realize a bottom-up process using a "glue" architecture, provided that administrative barriers (regarding access to private process state) can be relaxed or overridden. This can be done, for example, by letting the Foundation control all (and only) local sub-processes that are part of a multi-site task.

Let us now consider the impact of the structure of local SubEnvs on the choice of federated architecture. With respect to "glue" federations, the impact is relatively minor, because the Foundation is essentially external to the local SubEnvs. However, with respect to peer-to-peer there is a clear impact: SubEnvs with centralized process enactment naturally lend themselves to direct-peer-to-peer federation architecture, where the Foundation intermediary is built into the local process server that becomes the conduit to communicate with the Foundation and hence other SubEnvs. Decentralized local SubEnvs, in contrast, lend themselves to moderated-peer-to-peer architecture since there is no one component that stands out as the focal point. Instead, a new moderator component is attached to the SubEnv as a whole and communicates with each of the other local components as well as with the Foundation. However, a direct-peer-to-peer architecture is not inconceivable [69].

To summarize, the four categories represent different degrees of (de)centralization of the Foundation, ranging from a logically and physically centralized architecture, to a logically and physically decentralized architecture. Our key observation is that there is no one architectural style for federated PCEs that is superior to all others. Instead, we argue that the choice of a proper architecture depends on the requirements of the system, and more specifically on the architecture of the local PCE and on the multi-site process paradigm. The architecture should match these design requirements to enable the effective realization and specification of multi-site processes.

3 Oz Multi-Site PCE

The Oz system implements the only fully implemented[3] homogeneous PCE federation that we know of. It follows the direct-peer-to-peer architectural model, where the bulk of the process-oriented interoperability and collaboration support resides in the process engine (as elaborated in [15]) rather than the Foundation. The only part of the Foundation directly concerned with collaboration

[3]The Programming Systems Lab has used a multi-SubEnv Oz environment to support all our day-to-day software development since April 1995.

support, as opposed to communication infrastructure, is the registration/deregistration process for adding/removing SubEnvs from a multi-site environment (explained in Section 3.4). However, other aspects of the Foundation are strongly influenced by the need to support decentralized process interoperability, as detailed in this section.

The choice of architecture for Oz follows the rationale given in the previous section. First, one of the major requirements for the Oz system was to support interoperability among autonomous, geographically distributed, and possibly pre-existing processes. This requirement implies a bottom-up approach, which in turn suggests a peer-based architectural style. Second, Oz was developed (among other reasons) to interconnect Marvel instances [36].[4] Since the client/server architecture of Marvel [17] corresponds to the centralized-process-enactment, decentralized-task-execution local architecture, it was natural to adopt the direct-peer-to-peer approach.

3.1 Oz Overview

To define a project-specific local process or to parameterize a reusable process for an organization or project, Oz employs a *rule-based* process modeling language. A rule generally corresponds to an individual software development activity, and specifies the activity's name as it should appear in the user menu. A rule also defines typed parameters and bindings of variables to the results of queries on the local objectbase; a condition on the parameters and variables that must be satisfied before initiating the activity — generally an external tool invocation; the tool envelope and arguments for that activity; and a set of effects, one of which asserts the actual results of completing the activity on the objects referred to by the parameters and variables. There is generally more than one possible effect if the tool has more than one possible result (the simplest example is a compiler than generates either object code or syntax error messages).

The process engine enforces that rule conditions are satisfied, and automates the process via forward and backward chaining. When a user requests a rule whose condition is not currently satisfied, the system automatically backward chains to attempt to execute other rules whose effects may satisfy the condition; if all possibilities become exhausted, the user is informed that it is not possible to enact the chosen activity at this time. When a rule's activity completes, its asserted effect triggers automatic enactment of other rules whose conditions have now become satisfied. Both backward and forward chaining procedures operate recursively [31]. Users usually control process enactment by selecting a rule representing an entry point into a task consisting of one main rule and a small number of other auxiliary rules (reached via chaining) to propagate changes, but it is possible to define an entire process as a single goal-driven or event-driven chain — which is useful for simulation or training purposes. Built-in operations such as add an object, delete an object, etc., are modeled as rules for a uniform approach. Oz provides means for modeling and enacting synchronous and asynchronous "groupware" tools [67, 16], but the details are not relevant to this paper.

Oz supports object-oriented data definition and query languages. A class specifies primitive attributes (integers, strings, timestamps, etc.), file attributes (pathnames to files in an intentionally opaque "hidden file system" that should not be accessed except through Oz), composite attributes in an aggregation hierarchy, reference attributes allowing arbitrary 1-to-N relations among objects, and one or more superclasses from which it inherits attributes (and rules treated as multi-methods [7]). Ad hoc and embedded (in rules) queries may combine navigational and associative

[4] We mechanically upgraded our document authoring environment from Marvel to Oz, but devised a new software development process to better support componentization as explained in [37].

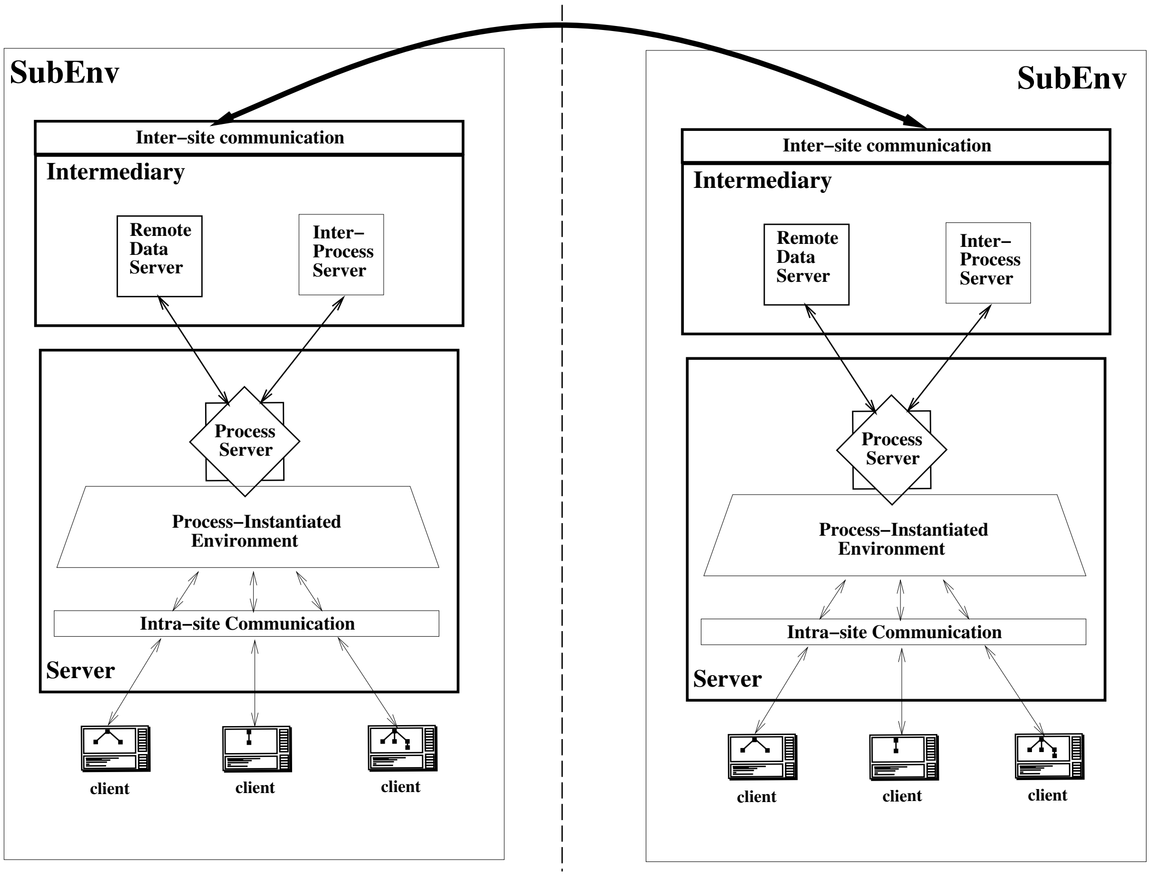

Figure 6: Oz External Architecture

clauses in a declarative style. Rules perform all data manipulation. Commercial off-the-shelf tools and other external application programs are interfaced to an environment instance through shell script envelopes, using augmented notation that hides from tool integrators the details of accessing the "hidden file system" and passing input and output parameters [24]. A return code from the envelope determines which of the several rule effects is asserted.

The external view of the multi-site direct-peer-to-peer Oz architecture is shown in Figure 6. It is a multi-client/multi-server architecture, within which each SubEnv follows a conventional client/server model and is self-sufficient for local work. Clients interface end-users to the SubEnv and allow them to issue requests for activities, and a centralized process server manages the enactment of each local process. Multiple SubEnvs are interconnected through the Foundation layer (represented by the intermediary boxes attached to the process servers), which enables multi-process interoperability and collaboration.

Servers communicate among themselves mainly to establish *Treaties* — agreed-upon shared subprocesses automatically and incrementally added on to each affected local process, and to coordinate *Summits* — enactment of Treaty-defined process segments that involve data and/or local clients from multiple sites. We stretch the International Alliance metaphor a bit, since Treaties among sites precede and specify Summits rather than vice versa.

3.1.1 Treaties and Summits

The essence of a Treaty is to establish common sub-processes that contain multi-site activities. The common sub-process is integrated within each local process, in the sense that its activities may be synchronized with other local activities, depend on the outcome of their execution, and so forth. Treaties are defined pairwise, which allows local environment administrators to form such agreements in a fully decentralized manner, without involving any global authority. Still, a Treaty among any number of sites can be created by forming all the relevant binary Treaties (and commands are provided to do this in one step, if the relevant administrator has appropriate privileges at each affected SubEnv). A Treaty between SubEnvs SE_1 and SE_2 over a sub-process SP_1 is established when:

1. SE_1 issues an *export* operation of SP_1 to SE_2. This operation assumes that SP_1 already exists in SE_1 (either as a native sub-process or imported from another SubEnv) and thus already integrated within it.
2. SE_2 issues an *import* command that fetches SP_1 from SE_1 and integrates it into its local process.

In order to control execution privileges, i.e., which site(s) can execute multi-site activities (e.g., due to platform restrictions, security, etc.), both *export* and *import* can be qualified with permissions to control on which site(s) the multi-site activities can be executed and from which site(s) the relevant data can be fetched. Finally, to further support decentralization, Treaties may be withdrawn unilaterally (except when the activities are actually being executed); note this requires a dynamic Treaty validation mechanism. Thus an interesting aspect of the Treaty mechanism is that it not only allows definition of decentralized multi-site processes, the (meta-)process for establishing Treaties is itself highly decentralized.

Summits are the enactment counterpart of Treaties. When a multi-site activity is issued for enactment in a given coordinator SubEnv, the Summit controller performs the following main steps:

1. Verify that the corresponding Treaty is valid.
2. Send to all participating SubEnvs a request to issue *local pre-Summit* activities, which involve local (hence private) process steps on local data with local tools. Wait for all sites to return before continuing to the next phase.
3. Execute the multi-site global activity, involving data from multiple sites, and possibly multi-site tools.
4. Send to the participating SubEnvs a request to issue *local post-Summit* activities, again, involving only local resources that are not exposed outside the local SubEnv. Wait for all sites to reply.
5. Enact further related Summits, if any.

Thus, Summits alternate between execution of shared, global, and multi-site activities, to execution of private, local and single-site activities, and effectively enact multi-site processes with minimal inter-process dependencies beyond the explicitly defined shared sub-processes and with minimal global architectural constraints. Full details of Summits and Treaties are given in [15, 9].

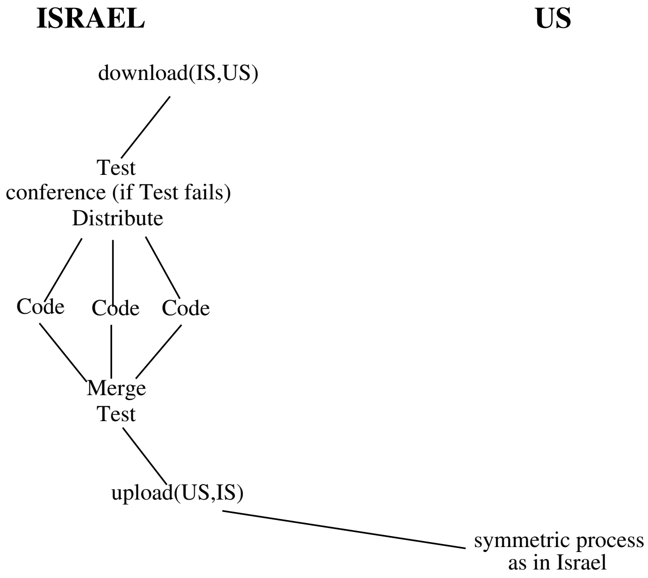

Figure 7: Example Multi-Site Process

3.1.2 Example

Imagine two collaborating development organizations, perhaps part of the same large institution, which are physically distant and temporally shifted, e.g., one in the western United States and one in Israel. As the deadline approaches, the tight schedule requires to maximize the parallel work of the teams, yet there are some crucial inter-module functional dependencies that might lead to inconsistencies, lost data, etc., if not handled properly. Fortunately, the time difference (10 hours) allows them to "take turns" and interleave work in different time intervals. More specifically, we define a symmetric process that is enacted at both sites 12 hours apart. (This is loosely based on a manual process employed by a company whose identity we cannot disclose.)

1. Download-module (across organizations)
2. Test-downloaded-module (local)
3. Distribute-module-to-local-developers (multi-SubEnv within an organization)
4. Code (local)
5. Merge (multi-SubEnv within an organization)
6. Test-new-module (local)
7. Upload-module (across organizations)

The `download-module` and `upload-module` activities are the main cross-organization Summits. Note that they are temporally constrained: the former should be done early to avoid "rush-hour" network traffic, and the latter has to finish in time for the other site to start its work for the day. (See [44] for discussion of temporal sensitively within Oz processes.) `Test-downloaded-module` and `Test-new-module` are shown here as local to a single SubEnv, but they might alternatively be distributed among multiple teams inside the campus, hence also defined as multi-site Summit rules. `Distribute-module-to-local-developers` and `merge` are multi-SubEnv but single-location activities; `merge` cannot start before all local `code` activities have completed (although a termination rule could be fired to ask the local developers to complete by submitting their work "as is"). Finally, `Code` is a purely local activity performed individually by each team. At times where both organizations are working, some groupware activities can be enacted to reconcile possible problems and conflicts, e.g., a desktop video conference might be set up if `test-downloaded-module` fails. Figure 7 illustrates this process.

3.2 Oz Architecture

The internal architecture of Oz is shown in Figure 8. We use the following graphical notations: squared boxes with the widest bold lines (e.g., the Server) represent operating system processes, or independent threads of control; squared boxes with lines with intermediate width (e.g., the Task component) represent top-level computational components that are part of the same operating system process as other components but are relatively independent from those components; squared boxes with narrow solid lines are computational sub-components; dashed-line separators within sub-components further modularize a (sub)component into its various functionalities; shaded ovals represent data repositories; and arrows represent data and/or control flow.

Oz consists of three main runtime computational entities: the Environment Server (or simply, the *Server*), the *Connection Server*, and the *Client*.[5] In addition, there are several entities that convert the various project-specific definitions into an internal format that is understood and loaded by the server, and some utilities for checking and repairing Oz objectbases.

There are three kinds of interconnections: client-to-local-server, client-to-remote-server, and server-to-server. The first connection is "permanent", in the sense that its existence is essential for the operation of the client. That is, a client is assumed to always be connected to its local server, and when such a connection becomes disconnected (either voluntarily on demand or involuntarily due to some failure) the client normally shuts down and is removed from the local server's state.[6] In contrast, the two other connections can be regarded as "temporary", since they are optional, and can be dynamically reconnected and disconnected over the course of a session, without disrupting the local operation of a SubEnv. This is a necessary feature to fulfill the independent-operation requirement, particularly when the servers are spread arbitrarily over multiple administrative domains. Implementation details of the connections are discussed in Section 3.3.

An Oz (multi-site) environment consists of a set of instantiated SubEnvs, and at any point in time none, some, or all SubEnvs may be *active*. A SubEnv is considered active if exactly one server is executing "on the environment", meaning that it has loaded the SubEnv's process, and the SubEnv's objectbase (containing persistent product data and process state) is under the control of the server's data management subsystem (described in [41]). Typically, an active environment also has at least one active (i.e., executing) client connected to its server, because the server automatically shuts itself down when there are no more active clients (and is automatically started up on demand by

[5] Oz actually supports several kinds of user and tool management clients, but the distinctions are not germane to this paper; see [59, 67, 21].

[6] An extension of this model, in which clients can be disconnected from their server and continue to operate independently to enact a process segment until reconnection, has been investigated separately to support mobile computing [60].

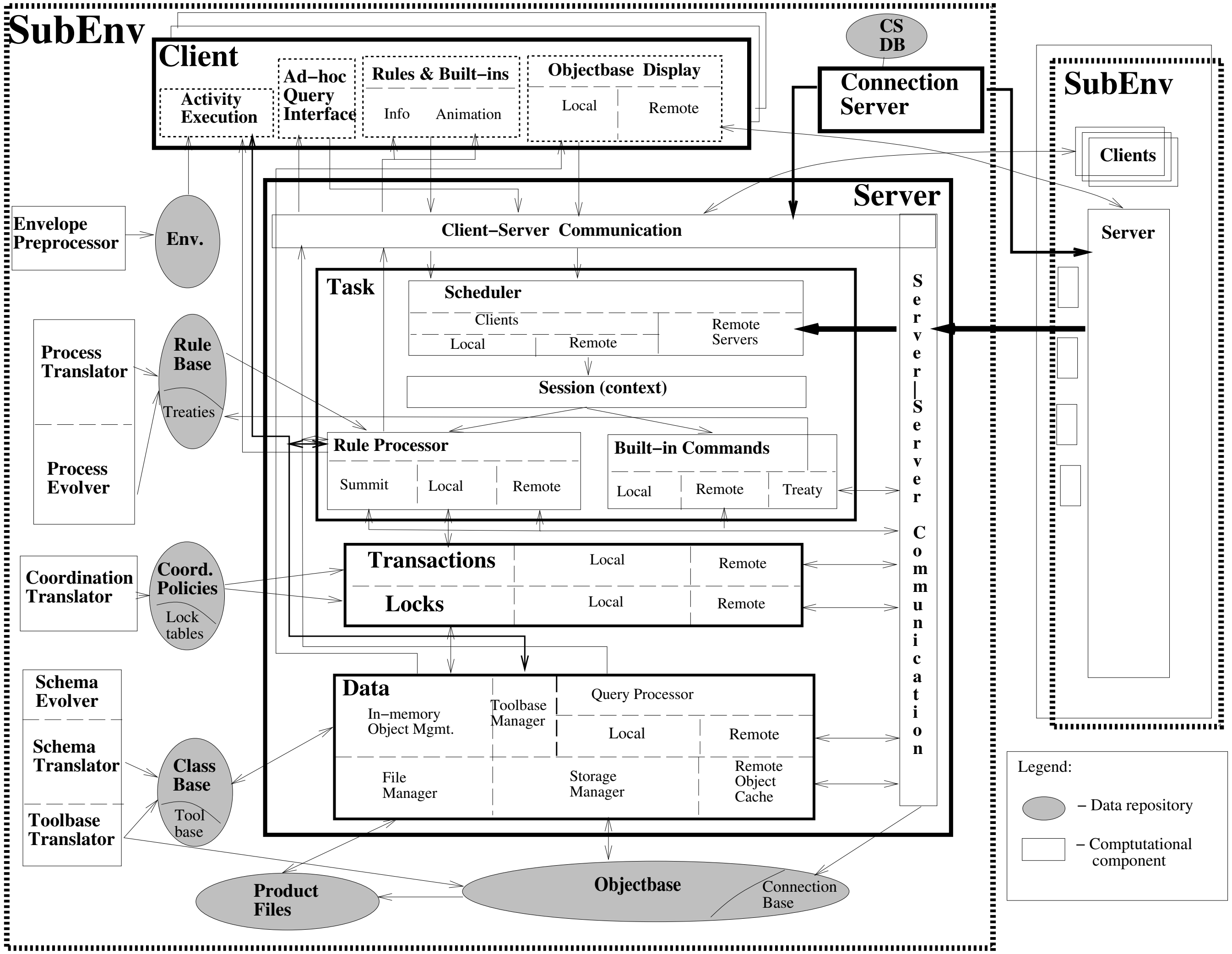

Figure 8: Oz Architecture

the Connection Server, as will be explained shortly).[7] In the rest of this section we will interpret the architecture figure, with emphasis on the Foundation components. Section 3.3 follows with discussion of the interactions among these components.

3.2.1 The Oz Environment Server

The server consists of three major components: task (or process), data, and transaction managers, each of which can be separately tailored externally. The task manager loads a parsed internal representation of the process model (including portions obtained through Treaty *import*), the data manager loads the schema for the product data and process state (currently imported rule sets must employ sub-schemas compatible with the local schema, although some conversion is supported), and the transaction manager is parameterized by lock tables and concurrency control policies, all stored in environment-specific files. See [42] for details.

Task Manager

The task manager is the main component of the server. Its front-end sub-component is the scheduler, which receives requests for service from three entities that correspond to the previously mentioned inter-connections, namely local clients, remote clients, and remote servers. With few exceptions, these requests are served on a first-come-first-served basis. The server is non-preemptive, i.e., it relinquishes control and context-switches to other tasks only voluntarily.

The session layer encloses each interaction with a server in a context containing information that enables it to switch between and restore contexts. The context of locally executing activities, including those that execute as part of a pre- or post- Summit, and the context of composite Summits, are represented in task data structures.

The rule processor consists of sub-components for processing local activities, Summit activities, and local activities spawned from either local or remote Summits. There are very few "system" built-in activities (notably parts of the configuration process, see Section 3.4.1), so the behavior of a particular instantiated SubEnv is mostly determined by the external rule set that defines the process.

The built-in command processor handles all the hardwired kernel services that are available to every SubEnv. These include the primitive structural operations on the objectbase (e.g., `add` and `copy` object), several display options and image refresh commands (the latter explained later on), access control, ad hoc queries, and the various dynamic process loading and Treaty operations.

Transaction Manager

All access to data is mediated in Oz by the transaction manager. Due to the required decentralization, each transaction manager is inherently *local*, i.e., it is responsible only for its local objectbase. However, transaction managers attached to each server communicate among themselves to support concurrency control and recovery involving remote objects.

Oz's transaction manager was developed as a separate component from the rest of Oz and is outside the scope of this work. This component is the subject of a related but semi-independent research line; see [17, 6, 5, 29, 30, 27].

[7] It is also possible to start up and shut down a server manually, often useful when testing a new version of the server.

Data Manager

This component consists of an in-memory object manager that provides uniform object-based access to data from any system component. Objects can be looked up in one of three ways: by structural navigation, by class membership, and by their object-id. Structural and by-class searches are requested by the query processor to service navigational and associative queries, respectively, and by-id lookup is used for several purposes, among them to support direct user selection of objects (mouse clicking in the objectbase display) as parameters to rules.

The second major sub-component is the query processor. It supports a declarative query language interface, and is called from both the rule processor for embedded queries and directly from the client for servicing ad hoc queries. Queries on remote objects are handled at this level, by invoking a server-to-server service.

The rest of data management consists of an untyped storage manager (implemented on top of the `gdbm` package) that stores the objectbase contents; a file manager that handles access to file attributes (recall that file attributes in objects are merely paths to files resident in the "hidden file system"); and an object cache (mentioned briefly in Section 3.5) that holds transient copies of remote objects when Summits take place.

As far as modeling facilities, the data manager is defined by the project-specific schema tied to the instantiated objectbase, including both class- and composition-hierarchies. As in the case of rules and the task manager, without a schema the data manager is useless since it cannot instantiate any objects.

3.2.2 The Oz Client

The client consists of four major sub-components: (1) interface to, and information about, rules and built-in commands, (2) objectbase display, (3) activity execution module, and (4) an ad hoc query interface. Oz clients are (conceptually) multi-threaded, i.e., a single client can support multiple concurrent interactions with local or remote servers. This enables a user to run in parallel several (possibly long) activities from the same client.

The command interface includes a process-specific menu and utilities for displaying rules and their (local) interconnections, all of which are stored at the client's address space and can be dynamically refreshed when a new process is (re)loaded. A dynamic rule chaining animator shows the control flow of enacted tasks, both local and Summits.

The objectbase display maintains an "image" of the *structural* information, i.e., parent/child and reference relationships, for browsing and for selecting arguments to activities. The contents of primitive and file attributes are transmitted only when needed. User can select the `open-remote` command to display the objectbase images from other sites and subsequently select objects from multiple sites, allowing invocation of a Summit activity. The client maintains multiple simultaneous connections to the remote servers, and is able to direct requests to appropriate servers.

Decentralization concerns imply that the policy for refreshing the various images — as the displayed objectbases change — should be determined on a per-SubEnv basis, and not be global, since the desired refresh policy for the objectbase image may vary depending on the degree of remoteness from (considering, e.g., low-bandwidth or highly congested connections), and frequency of interactions with, remote servers. Thus, Oz supports a per-SubEnv tailorable refresh policy. That is, a user client can determine for each connected SubEnv server the frequency for refreshing its local objectbase image, thereby controlling the communication overhead. The policy itself can be based on time, or on the number of structural changes made on the objectbase. The default policy, as with other aspects of communication in Oz, follows a "lazy" approach — the updates are deferred until users actively request services from the server, after which the updates (provided as deltas up to a size threshold after which the entire objectbase structure is retransmitted) are piggy-backed onto the reply.

3.2.3 Connection Server

The Connection Server's main responsibility is to (re)establish connections to a local server from local clients, remote clients, and remote servers. However, it does not participate in the actual interactions between those entities; it serves only as a mediator for "hand shaking" purposes. In some cases, the destination server to which a request for a connection is made may not be active, in which case the Connection Server is capable of automatically (re)activating a dormant server. In other cases the desired server may be active but its address (host IP address and port number) might be unknown to the requesting entity, in which case the Connection Server sends that information to the requesting entity for further communication.

Unlike the environment Server, the Connection Server is (conceptually) always active, since it is implemented as a daemon invokable from the Unix `inetd` mechanism. Thus, each configured host has its own (logical) Connection Server that supports all SubEnvs (of the same or different global environments) that reside on that host. The actual invocation and functionality of Connection Servers is discussed in Section 3.3.3.

The following subsections involve excruciating detail, and the reader might choose to skip directly to Section 4.

3.3 Communication Infrastructure

The Foundation's communication infrastructure is the cornerstone of the interconnectivity mechanism and is, therefore, important for the understanding of the decentralized architecture. We address here two main issues:

1. How to represent, store, identify and locate computational entities (i.e., clients and servers) across SubEnvs.
2. How to perform the actual transfer of data and control between those entities.

The design of the infrastructure is influenced by the following issues:

1. Decentralization and independent operation requirements (which in turn entail the "shared nothing" requirement) imply that no communication-related information can reside in a shared repository and must be therefore somehow replicated.
2. Independent operation, coupled with the fact that SubEnvs may or may not be active at any given point in time, implies that the architecture should be designed to tolerate temporary disconnections between SubEnvs as a normal scenario, not only as an exception. Moreover,

since the communication port of the entities might change dynamically (due to the "temporary" nature of these connections), the communication protocol should be able to *dynamically* (re)locate and (re)connect to remote sites, while carrying out other on-going tasks.

3. As an infrastructure for interoperating enable tailorability and modeling of the communication itself, on a per-project basis.

3.3.1 Approach

The key to addressing the two major issues, given the above requirements, is in the proper design of: (1) a decentralized *connection database*; and (2) a *communication protocol* that maintains this database.

The connection database is a persistent repository that contains the necessary information for cross-SubEnv communication. The shared-nothing requirement eliminates the possibility of a shared repository, so the obvious alternative is to replicate it in all sites. However, maintaining consistent replicas at all sites violates autonomy and independent operation, particularly due to the dynamic changes that occur frequently whenever sites are (de)activated. And with arbitrary geographical distribution of SubEnvs, this approach simply becomes impractical. On the other hand, despite the given lack of consistent replication, there must be a way to still ensure inter-SubEnv connectivity on demand.

A hybrid approach that addresses both concerns is to maintain a *semi-replicated* database, whereby the database consists of two kinds of data: A *static* component that contains connectivity information that changes rarely is fully replicated, and thus assumed to always be valid; and a *dynamic* component that contains information that changes frequently is not always replicated, and might sometimes be invalid. Corresponding to that division, there are two modes of communication: direct communication through the volatile dynamic information, and indirect communication through the always valid static information. The former mode is faster, but will not work if the dynamic information is invalid, and the latter is slower but the connectivity information is guaranteed to be accurate (this will be further clarified later in Section 3.3.3).

As for flexibility concerns, the obvious direction to follow is to exploit the process-centered approach and provide facilities and notations for (1) modeling communication on a per-project basis, and for (2) the corresponding enactment mechanisms. However, communication modeling imposes problems that do not exist in software 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 within a single SubEnv), which means that communication modeling is at least partially a global process. Second, communication involves low-level system calls and mechanisms that are hard to expose (without violating abstractions) to the high-level modeling language.

Our solution here is again a compromise: the connection database is modeled as a set of first-class instances of a class that is defined using the standard data definition language, but the class is built-in. Thus manipulation of the database is performed in part by low-level components of the Oz kernel, and in part by (built-in) rules. The idea is to define a hard-wired structure for the database, but expose it and its contents to all levels of the system (and to users), and in particular make it extensible via the process modeling language, as well as from the kernel. Thus, manipulation of parts of the connection database is performed through a built-in process, and has the benefits that come with process modeling and enactment in general, although with some limitations (see

```
SUB_ENV :: superclass ENTITY;
# Static Information
  env_name     : string;          # unique across global environments
  env_id       : integer;         # unique across global environments
  subenv_id    : integer;         # unique within a global environment
  subenv_name  : string;          # site:pathname or logical name
  site_name    : string;          # e.g.: cs.columbia.edu
  site_ip_addr : string;          # e.g.: 128.59.16.20
  has_nfs      : boolean = false; # true if shares NFS with local server
  state        : (New, Initialized, Defunct) = New; # configuration state
  local        : boolean;         # TRUE if local, FALSE if stub object
# Dynamic information
  active_host  : string;          # e.g.: bleecker.columbia.edu
  host_ip_addr : string;          # e.g. 128.59.24.34
  port         : integer = 0;     # port number, if active
  active       : boolean = false; # TRUE if active, not guaranteed
  subenv_ob    : set_of ENTITY;   # The local objectbase is connected here
# Process-Specific
end
```

Figure 9: The built-in class SUB_ENV

Section 3.4) while other parts can be added on as desired without modifying the Oz system: the class definition of the connection database can be augmented with additional attributes, and manipulating rules, so long as the default required attributes remain intact.

3.3.2 The Oz Connection Database

The implementation of the Oz connection database follows the rationale given above. Each SubEnv maintains a private connection database consisting of a set of objects that are instances of the built-in class `SUB_ENV`. Each of these objects represents a distinct SubEnv in the global multi-site environment. The SubEnv objects are represented as the root objects of their respective objectbases, and thus they are always part of the displayed image at all clients at all sites of the global environment.

The actual definition of the `SUB_ENV` class is given in figure 9.

The static attributes contain information that is determined at site configuration time and is modified only by subsequent (re)configurations (see Section 3.4). A `SUB_ENV` object contains values that always enable location of the SubEnv and connection to it (through the Connection Server), such as the `subenv_name` and `subenv_id` fields for identifying the SubEnv, and the `site_name` and `site_ip_addr` that specify the location of the Connection Server. Note that the value of the `site_name` attribute need not be identical to the value of `active_host`, due to the fact that a Connection Server can activate other hosts within its domain. This point is discussed later.

Unlike the static attributes, the dynamic attributes are frequently modified by the Oz kernel during normal (inter-) process enactment, and contain dynamic bindings of values (e.g., current Internet address of the host that executes on the SubEnv, its listening port, etc.). In each local connection database there is exactly one local `SUB_ENV` object (denoted by having a `true` value in its boolean `local` attribute), to which the local objectbase is connected (through the `subenv_ob` compositional

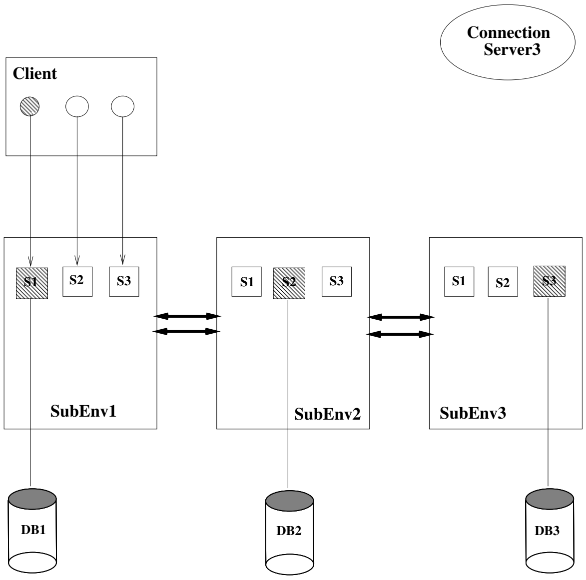

Figure 10: Connection Database

attribute). The rest of the SUB_ENV objects are "stubs" used to connect to other SubEnvs. For example, in an environment consisting of four SubEnvs, each SubEnv usually has four distinct SubEnv objects (i.e., the total number of SubEnv objects in an environment is normally the square of the number of SubEnvs), one of which is the local "real" object and the other three are stubs "pointing" to the other SubEnvs. By definition, all stubs that point to the same object (one in each SubEnv) must contain identical static information — this is guaranteed by the configuration process. In contrast, the dynamic information may vary in different stubs representing the same SubEnv object. The reason is that a stub in the server is updated only when the server (or one of its local clients) actively requests to communicate with other server represented by the stub. That is, the stub is not updated every time the corresponding real SubEnv object is modified (e.g., when it becomes inactive, or is reactivated on a different host). Thus, the dynamic information is always valid only in the real (i.e., non-stub) SubEnv object.

As for the client, the situation is as follows. Upon initialization, it receives from its local server an image of the local objectbase, and an image of the connection database. When the client issues the `open-remote` command with a remote SubEnv stub as its argument, the client replaces the image of the stub with the image of the (remote) real object, along with its connected objectbase, and the local server's stub is updated with the proper dynamic information. This switch of images at the client is illustrated in figures 10 and 11: In 10 the client has no open remote connections so its image of the connection database is directly mapped to the local connection database; 11 shows client's image after an `open-remote` was issued on `SubEnv3` (ignore for the moment the Connection Server in the figure). There, the image for `SubEnv3` has switched from the local stub to the image of the real object (along with its connected objectbase).

An alternative approach to maintaining the connection database at the client (which was in fact implemented in an earlier version of Oz) would not switch the image of the SubEnv objects upon opening a remote connection. Instead, a distinguished attribute of the stub would represent the sub-objectbase image stemming from the actual SubEnv object, and any requests to access remote objects would be directed to the local server — which would perform the request on behalf of its

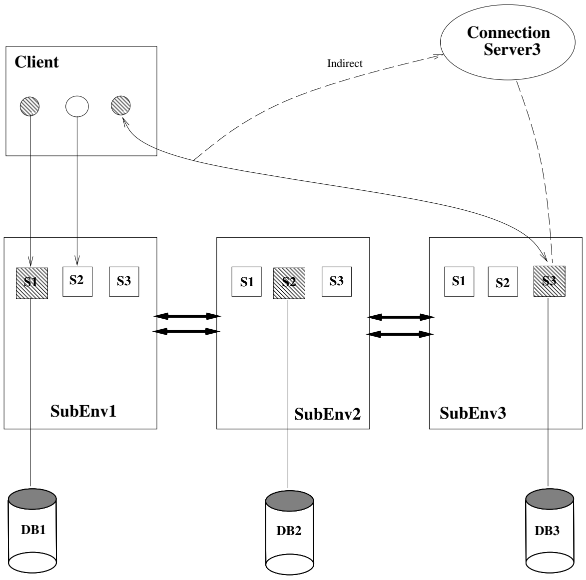

Figure 11: Connection Database with a Remote Connection

client, including possibly contacting the Connection Server. The main advantage of our current approach over the former one is that it simplifies the client's operation and the communication protocols in general since the client communicates only with its local server and all cross-site communication is done through the servers. However, this approach unnecessarily overloads the servers and overall significantly increases the performance overhead for remote communication, since every remote request must pass through the local server, including the built-in operations that do not require process-authorization such as parameter selection and remote browsing (conventional access control still protects sites from unauthorized remote access just as it does so for local clients, see [42]).

3.3.3 The Communication Protocol

As mentioned earlier, Oz supports two modes of communication: the *direct* communication mode, which attempts to use the (possibly invalid) dynamic information in the connection database to connect directly to the desired server, and the *indirect* mode, which uses the static and always valid information in the connection database to set up communication via the Connection Server. Indirect communication is used either to establish a new connection for which there is no dynamic information available at the requesting server, or when the dynamic information turns out to be outdated (e.g., due to the fact that the previous target server terminated its execution).

In either case, indirect communication is followed by updating of the corresponding dynamic information in the stub, so that subsequent interactions with the same server can occur in direct mode. In some cases, there is no running server on a given SubEnv, which means that indirect communication must take place. In this case, the activation capabilities of the Connection Server are used to start up a new server.

Figure 11 illustrates the two modes in client-to-remote-server interactions (it is similarly handled in server-server communication). As long as the direct channel is valid, all interaction between the client and the remote `SubEnv3` is done directly. If `SubEnv3`'s address is not known to the client,

```
if (remote-server is marked as active)
then
    try to connect directly using the dynamic information
    if (connection established)
    then
        communicate
    end if
end if
if ((remote-server is marked as not active) OR
    (dynamic information is invalid, communication failed))
    then
        contact the Connection Server through the static information
        if (connection established)
        then
            1. get the (dynamic) information from the Connection
            Server and update the local stub SubEnv object
            2. communicate
        else
            return error. Connection cannot be made at this point
        end if
end if
```

Figure 12: Server-to-server Communication

or has become invalid (e.g., the server has been deactivated), the indirect channel (shown as the dashed arrows) is used to establish the connection, after which the (new) direct channel is used again. Since the address of the Connection Server at `SubEnv3` is always known (maintained by the static information) and it is always available (through the daemon mechanism), the likelihood of successfully (re)connecting is very high (assuming network connectivity). Finally, the indirect communication has an important role for fault tolerance: it is essential for handling inter- and intra- site failures independent of Oz. For example, if a specific host that previously ran a SubEnv environment server crashes, subsequent communication with the Connection Server might lead to restarting a server on the SubEnv from the same or a different host in the domain.

The communication protocol is summarized in figure 12. Note how all the necessary information can be obtained from the SubEnv objects in the local connection database.

The design of the communication protocol meets the constraints imposed by independent operation and decentralization requirements, and is somewhat analogous to other aspects of the system that deal with interoperability. On the one hand, the "lazy" approach to updating dynamically changing information avoids the need to broadcast the updates made in the (real) SubEnv objects to all the stubs in the remote SubEnvs. This is particularly important since the sites might be physically dispersed and thus incur large communication overhead. Moreover, the fact that not all sites are necessarily active at all times simply makes the "eager" approach impossible. And most of all, such updates are not always necessary (e.g., when some remote SubEnvs are not interacting with the updated SubEnv). On the other hand, it is still always possible to reach remote SubEnvs (as long as they are reachable through the underlying network), with some overhead. The main point is that the freshness of the dynamic information is correlated with the frequency of communication, i.e., the more often a remote SubEnv is contacted, the more likely the dynamic information in the corresponding SubEnv stub will be accurate at the contacting SubEnv, thereby increasing the chances for successful direct communication.

3.4 Site Configuration Process

Recall that one of the goals in the design of the communication infrastructure was to enable some degree of modeling and tailorability. The first step towards achieving that goal was in the *definition* of the connection database as a set of first class objects instantiated from a designated class that could potentially be evolved on a per-project basis (Section 3.3.2). The second step towards achieving that goal is in the *manipulation* of the connection database. The idea is to exploit the concept of process modeling and apply it to configuration by defining a *registration process* specified in the normal process modeling language, and to exploit the concept of process enactment by executing the configuration process using the identical enactment engine normally used to enact a software process.

As with software processes, this approach grants the potential for tailorability of the configuration process. However, divergence from the standard process in this case is confined mostly to the global environment level, since configuration is inherently a global task. Nevertheless, some limited site-specific extensions to the global configuration process are also possible, in principle. In addition, since (re)configuration is performed using the normal process engine, it can be performed dynamically as it amounts to a normal process step invocation. This fulfills one of the base requirements set forth in Section 2.1, namely dynamic configuration. Further, the exploitation of process automation ensures that the addition/deletion of sites is carried out consistently across all of a global environment's sites, with minimal human (and error-prone) intervention. Finally, protection from accident is afforded through the objectbase's normal access control facilities. We now present the actual registration process. An earlier version of the configuration process was described in [13].

3.4.1 Configuration Facilities

The registration process presented here is fairly similar to any other Oz (sub)process. The main differences are that (1) it was written by the environment kernel implementors rather than by project-specific process engineers (although the latter might extend this process to some extent); and (2) it is a "global" process that requires the issuer to have administrator privileges for all affected SubEnvs, since it manipulates the connection database in those SubEnvs.

The process consists of a set of rules and envelopes that wrap "configuration" tools, and operates over the connection database, i.e., over all SubEnv objects in all SubEnvs. The details of the registration process, just like those of software development processes written by typical process engineers, can safely be ignored by most environment end-users. The process consists of three tasks:

1. Registering a new (perhaps pre-existing) SubEnv into an Environment.
2. Deregistering a SubEnv.
3. Migrating a SubEnv to a different location and/or host within the same global environment.

All tasks are modeled as rules that are invoked interactively inside any one of the existing SubEnvs, with the same user interface normally employed for regular process enactment.

The *registration* task is the means by which a multi-site environment becomes populated. This task can be invoked from any other existing active SubEnv. The only exception is the creation of the first SubEnv, which is "hand crafted" using a utility. The registration task consists of two steps: (1) adding a new stub object (representing the new SubEnv) to all existing SubEnvs and (2) physically creating and initializing a new SubEnv or joining a pre-existing one.

The first step may be executed from *any* SubEnv already participating in the relevant global environment. It binds the SubEnv objects of all existing SubEnvs (the "real" object for the local SubEnv and stub objects for remote SubEnvs), queries the administrator for the new site's static information (SubEnv name, IP address, etc.), assigns to the new SubEnv a unique id by simply selecting an id that is not used by any of the existing SubEnvs, and creates in all existing SubEnvs (including the local one) a replicated SubEnv stub object instantiated with the specified static information. If the activity detects the occurrence of any of a set of common problems (e.g., cannot contact a remote SubEnv), it returns an error code (which can in principle trigger the activation of an "exception handler" rule).

The second step (which may be automatically invoked when the first step completes) in the process creates and initializes the new SubEnv (or modifies the joining SubEnv if it pre-existed), by invoking a remote environment-initialization utility at the new (joining) location, creating (modifying) the local SubEnv object there and adding all the stub objects — one for each of the other SubEnvs in the environment.

Notice that both steps require contacting remote SubEnvs and updating their objectbases (adding SubEnv objects). This is possible due to a batch facility that enables recursive invocation of a new Oz client from within an envelope forked by an existing client. The new client performs the sequence of commands listed in a script and exits. This gives the ability for an envelope executing at a client in one SubEnv to generate a script of Oz commands and spawn another (batch) client that executes the generated script in a remote SubEnv. This technique provides for a simple registration mechanism that can be controlled from a single interactive client. For example, the registration activity generates a script of commands parameterized with the relevant SubEnv objects, which, when executed, adds the newly created SubEnv object in all remote SubEnvs by spawning a batch client that operates on the proper remote SubEnv with the generated script as the input command batch file.

The *deregistration* task removes a SubEnv from the global environment, by deleting the site's SubEnv objects from all other SubEnvs (again, using the batch facility), and by deleting the SubEnv objects representing these other SubEnvs in the deregistered SubEnv's own (sub)-objectbase. The SubEnv itself is only split off from the global environment, but it is not destroyed; the former SubEnv can continue operation on its own as a single-site environment, and may be rejoined into this or another multi-site environment later.

3.4.2 Discussion

The interesting aspect of configuration from the perspective of research in process modeling is that it is treated as a fully integrated process, benefiting from most of the advantages that come with process modeling. In particular, it can be enacted by the process-centered environment exactly like any other process that one undertakes during software development. Furthermore, the process can be partially modified and tailored for new and existing environment instances using the same process evolution capabilities (see [12]), provided that the required parts of the data and rules are protected from modifications. Finally, it was easy to implement, reusing largely pre-existing facilities. For example, maintaining the configuration database as part of the process and product database took advantage of Oz's persistent object management system. And since the uniform mechanism is part and parcel with the rest of the system, many aspects of the (re)configuration process come nearly "for free". For instance, transactional (re)configuration can be supported immediately as a private case of the general decentralized transaction manager, eliminating the need for a special purpose transaction facility for configuration.

3.5 Implementation issues

There are several other aspects of the Oz implementation that are not strictly part of the architecture, per se, but are necessary for pragmatic reasons. We discuss a few of them here.

3.5.1 Decentralized Naming Scheme

We already discussed in Section 3.4 how SubEnvs are identified uniquely. But there are several other entities that require global yet decentralized naming scheme, namely objects, multi-site activities (i.e., rules) and clients. Global object identification is needed because multi-site activities involve objects from multiple sites. The uniqueness of multi-site activities is necessary for avoiding name conflicts during the *import* operation, and client ids need to be uniquely generated to enable a server to accept connections from both local and remote clients (using the `open-remote` operation), as well as to be able to redirect rule chaining animation messages to remote clients.

The main goal in the design of the object-id management was to reconcile the conflict between allowing autonomy in id assignment and still providing uniqueness. The solution is to identify an object by the pair `(SubEnv_id, obj_id)` where the latter is determined by the owner SubEnv with no global constraints, and the former relies on the unique SubEnv scheme as explained earlier. a "long" id split into two fields, but rather two different ids. This reflects the decentralized nature of the architecture: each local object manager can employ its own id management without worrying about uniqueness across sites. Moving/copying objects permanently across SubEnvs is treated as adding a new object to the target SubEnv with the specified values (and in case of `move` also deleting the source object), thereby assigning to it a new id locally.

The implementation of activity- and client- id assignment is also based on the uniqueness of the SubEnv-id. Each local server maintains a private counter, to which it adds the SubEnv-id multiplied by a large constant (the same constant must be used, however, but this restriction is acceptable in homogeneous federation).

3.5.2 The Remote Object Cache

During the course of enacting a multi-site process, a remote object might be accessed several times by a SubEnv from various activities (i.e., rules) within the same task (i.e. rule chain). entirely different tasks. A naive implementation of multi-site tasks would request a new copy of the remote object for each access request, but this is obviously time consuming, particularly when the sites are geographically dispersed (and the communication bandwidth may be relatively low). Thus, an effective object cache may improve the performance significantly. Another motivation for the cache is to hold pre-fetched objects in case a pre-fetching mechanism is employed (e.g., based on the semantics of the process model as discussed in [60]). The interesting aspects of the cache implementation from the federated PCE perspective are to address its particular requirements, and to possibly use the semantics of the federated process and data models to direct the cache-invalidation policy.

A typical Oz activity involves a *set* of related objects bound using structural queries. To reflect this characteristic, each entry in the cache contains, in addition to a replica of the object, a relationship list with cached information regarding the ids of objects in its neighborhood, i.e., its parent and children in the composition hierarchy as well as referenced objects. Note, however, that only the relationships to other objects are cached, not the objects themselves (which may or may not reside in the cache). Thus, the following invariant is maintained: if a cache object is valid, then its relationship lists are also valid, i.e., it is indeed connected in the real (remote) objectbase to all objects whose ids are stored in those lists. The implication of this invariant is that an object needs to be invalidated not only when its content changes, but also when its relationships with other objects change (e.g., a referenced object is deleted or a new reference is added). However, since structural changes are less frequent than object changes, and since the relationship list is heavily used during process enactment, caching the structure outweighs its invalidation overhead.

The invalidation policy addresses the geographical distribution requirement, too. Instead of employing the conventional policy — whereby the original object, when modified, asynchronously broadcasts to all of its replicas invalidation messages — Oz employs a synchronous "short lived" cache. A remote object is cached in the coordinator SubEnv only for the duration of the multi-site task. Any modifications to a cached object (or to its relationship list) that occur at the "home" site through local activities, are propagated to the coordinator SubEnv at any of the synchronization points of the multi-site task, thus avoiding the need to send a separate update message. Finally, when the multi-site task completes, the remote objects that were accessed by this task are automatically cleared from the cache. This synchronous approach, while relatively limited in scope, ensures independence from race conditions that are likely to occur due to network delays (e.g., when an object update arrives after the access to an object), and overall decreases the communication overhead significantly.

To illustrate the improvement in performance, Table 1 compares the results of running a multi-site task consisting of two Summit (i.e., multi-site) activities with spawned local activities on cache-enabled server vs. regular server. In addition to the absolute execution times, we used the number of messages exchanged between the SubEnv as a measure of performance improvement, since the major delays are incurred by the communication overhead inversely proportional to the available bandwidth, — and since we used objects of similar size, all messages had about the same length.

As can be seen, the improvement is significant due to the reduction in the number of calls to retrieve the remote parents and children, and there is no additional communication overhead due to the operation of the cache. The reason for the large number of requests for `GET_REMOTE_PARENTS` and for `GET_REMOTE_CHILDREN` in the non-cache server is that when the rule processor evaluates which rules to chain to, there are many possibilities to instantiate each such rule with parameters. Only a small percentage of rule instances are actually executed because the rule's condition cannot be satisfied for most of these instantiations, but the evaluation with remote objects still has to take place.

Message Type	non-cache server	cache-server	purpose
GET_REMOTE_OBJECT	2	2	get remote obj
GET_REMOTE_PARENTS	36	2	get remote parents
GET_REMOTE_CHILDREN	18	2	get remote children
CHECK_REMOTE_EXEC	2	2	treaty validation
BC_REMOTE	2	2	remote backward chain
ASSERT_REMOTE	8	8	remote assertions
FC_REMOTE	6	6	remote forward chain
Totals	74 (29 sec)	24 (9 sec)	

Table 1: Performance comparison with and without cache

3.5.3 Context Switching During Server Interaction

In a conventional client/server architecture, clearly if some requests take a long time to service, and/or they consist of a series of interactions between the client and its server for which a context must be kept throughout the interaction, then a context switching mechanism is necessary to avoid starvation of other waiting clients. For example, in a single-server enactment of a (local) task, all activities execute at the client's address space. These activities might take arbitrarily long, and it is not reasonable to expect that the server will block while the activity executes at the client. Further, since a task consists of several activities, it is even more unacceptable to assume that a whole task (and its associated activities) will actually execute atomically, even if the "all or nothing" atomicity property is required for the execution (as in an atomic transaction). Thus, it is necessary for the server to keep a context for each chain and switch among the contexts to service multiple clients concurrently [10].

In the direct-peer-to-peer Oz architecture, there is an additional problem. In cases where the server has to communicate with other servers in order to service a client request, two problems might arise: (1) the server might wait arbitrarily long until the remote servers complete to service the request, thereby reintroducing the starvation problem; and (2) if a server has to wait for other servers, then the servers might deadlock. In other words, since a server acts as a client, it can block indefinitely waiting to be serviced by another server due to circular waiting. Moreover, since servers might wait arbitrarily long, the chances for getting into a deadlock situation in a naive implementation are pretty high, particularly if the servers communicate frequently.

One possible solution is to implement a fully context-switchable server, so that it never blocks. However, besides the difficulties with implementing arbitrary context-switching, this might introduce inconsistency in the server's state if arbitrary interleaving is allowed. In particular, some critical sections would need to be defined in order to protect the integrity of the data in the servers, reintroducing the deadlock problem.

The pragmatic solution in Ozconsists of three different methods that are applied at different cases. The first is full context switching, in which the server sends the request to the remote server(s), saves the context of the operating task (which then enters a "sleep" state), and is ready to accept new requests for service. Oz employs this approach in the three major waiting regions: when sending and activity to be executed by a client, and when the two fan-out phases for, pre- and post- Summit, occur.

The second method, which we refer to as the "busy-service-wait" loop, is applied to services that are characterized by being simple and consisting of a single, short step, but are called from deep within a complex context that makes it highly undesirable to switch contexts there. An example of such a case is Treaty validation at the beginning of a Summit. The gist of busy-service-wait is that the requesting server is not blocking but does not leave its context either. That is, it is primarily waiting for the reply to its original request, but while waiting it checks to see if new incoming requests for service arrived, and services them immediately. The key observation that makes this method feasible, is that unlike servicing a client, an Oz server that services a request from another <u>server</u>, never needs to communicate with other servers or clients. That is, servicing remote servers is done locally. Under that premise, it is guaranteed that there will be no circular waiting because the kind of services that the server handles while waiting for the reply do not depend on any other computational entity.

Finally, the third method, which we refer to as "extended-busy-wait-service", is a modification of the second method and is applied to steps that are themselves composite and require multiple service requests to complete yet they are still hard to fully context-switch. An example is the instantiation of a Summit rule with arguments and bound variables, which may involve multiple remote servers. In these situations, care must be taken so that the partial bindings are not altered while servicing incoming requests in the service loop — or in other words, there has to be a way to protect the integrity of the data while bindings take place because they are not performed atomically.

The solution here is to defer any service request that can potentially <u>update</u> objects and queue it for later execution. However, if not careful, the deadlock problem could reappear if two (or more) servers were in the same "extended-service-wait" phase, deferring each other's binding requests for later execution, indefinitely. Fortunately, since the binding phase is read-only, its requests can be serviced immediately, so a server in a midst of a binding phase can still service remote requests for binding from other servers. When the binding phase completes, the server can context switch to service any queued update requests. Figure 13 summarizes the "extended busy-wait-service" algorithm that is executed whenever a server requested a service from another server and is waiting for the reply.

3.5.4 Network Protocols

We implemented the interprocess communication layer of the Foundation directly on top of TCP/IP sockets. To improve performance, both client-server and server-server connections stay open until they are explicitly closed (through a close request), and the server multiplexes the requests using the `select` mechanism. This approach limits the number of possible connections to the maximum allowed by the operating system process, but in most advanced Unix implementations this is an adjustable and not a scarce resource — and the performance gains over the stateless method of establishing the connection for each interaction are significant.

The server has a front-end component that listens to both client and peer-server requests on the same port, parses the header and dispatches the request to either the local scheduler or to the federation's intermediary. Each request (including remote-server requests) is tagged with a client and thread (within the client) ids, content-length, and the body of the request. The server's port is determined dynamically. When started, it stores the port along with the host in a file that is accessible to clients. Remote servers and clients use the information in the connection database in order to locate the server's port, and the connection server listens on a well-known port.

```
while (waiting_for_reply)
do
    if (incoming_message)
    then
        if (this is a new service request message)
        then
            if (a non-update request)
            then
                service_request(incoming_message)
            else
                queue_request(incoming_message)
            end if
        else                                                    /* this is the reply */
            waiting_for_reply = FALSE
        end if
    end if
end while
```

Figure 13: The Extended Busy-wait-service Algorithm

3.6 Evaluation

The comprehensive infrastructure that was built for Oz to support interconnectivity seems to have fulfilled all required functionality while addressing the main "conflicting" requirements, namely decentralization, independent operation, and autonomy. The semi-replicated connection database ensures full decentralization of the global state yet full connectivity. The Connection Server as an auxiliary entity for (re)establishing connections across sites and for (re)activating servers enables sites to operate independently but acquire the necessary information when they needed to communicate with other sites. The configuration process, while global in nature, promotes site autonomy by allowing each site to augment its own site-specific rules and state as long as they preserve the global process. And the refresh policy enables process engineers to optimize inter-site communication based on the frequency of interaction and the available inter-site bandwidth.

In attempting to reconcile the inherent conflict between support for decentralization and autonomy on one hand, and facilities for global definition on the other hand, care must be taken, however, to avoid unacceptable compromises. For example, the global objectbase display utility in Ozprovides powerful browsing of whole remote objectbases but cannot be disabled, which violates the privacy principle. On the other hand, early experiments with Oz have shown the need to supply optional global facilities, particularly for tightly-coupled SubEnvs.

Finally, the necessity of a global context for SubEnvs as a whole is questionable. For example, it may have been possible to avoid a global connection database and only denote, for a given SubEnv, S, the set of remote SubEnvs with which S interacts. Then the connection database is not global in the sense that not every SubEnv maintains the same set of remote SubEnvs in its database. This approach has been adopted in the recent EmeraldCity Oz process for the continued development of Oz[37], but "hand crafting" of all the SubEnv stubs was required.

4 Heterogeneous Federation

4.1 Requirements

Recall that in the heterogeneous model, each site (or team) runs a separate PCE that works together with other PCEs via the external federation "glue". Each site may employ a *different* PCE, although a few may happen to use independent copies of the same system. Here we also refer to the local site's PCE as a SubEnv, and the "glue" as the Foundation. Again we are concerned primarily with multiple sites on a wide area network, with independent administrative domains, and again take as given the requirements that each site must be able to support an autonomously devised process model, but also interoperate and collaborate with other sites in decentralized fashion, from the problem definition.

We have identified the following requirements for heterogeneous federation, which are organized to parallel the homogeneous federation requirements.

- The Foundation must provide some infrastructure whereby it communicates with the SubEnvs. In general, the SubEnvs cannot communicate directly with each other since (by definition) they were designed as independent PCE systems, although it is conceivable that Foundation may consist of little more than a standard PCE-cognizant protocol layer between SubEnvs. Realization of heterogeneous federation requires that some conceptually homogeneous component is added to each PCE (with perhaps quite diverse per-PCE implementations), so that it can bind into federation, in the simplest case only to implement the common protocol. In any case, we assume this component is inherently quite limited, not taking over process modeling and enactment functions, since otherwise we could consider it to effectively convert the federation to the homogeneous case. (Figure 1 shows both homogeneous and heterogeneous federations as continuums, with a grey area in between, because the distinction is not clear-cut.)
- Another fundamental requirement, as in homogeneous federations, is that as far as local work is concerned, a SubEnv should operate independently and continue to provide the same capabilities and same support as it would operating as a single-site PCE outside any federation. It should not in any way rely on communication with the Foundation in order to perform its standard functions with regards to defining and executing its own local process.
- Usually, the SubEnvs must somehow be made aware of any federations in which they participate, possibly more than one at a time, in order to contribute to the global process enactment. In theory, it might be plausible for a SubEnv to perform work on behalf of a federation without ever noticing that the heterogeneous federation exists (which would not normally be the case for homogeneous federations). Thus an alternative is that only the Foundation is aware of the various SubEnvs, and picks up results through some non-intrusive manner, such as understanding file formats of what the PCE considers internal process state information. (Some possible interaction schemes are outlined in [28] for adding on external concurrency control to a PCE that does not know it is using concurrency control, and some similar mechanism could be applied here.) As in homogeneous federation, there must be some means for configuring federations and allowing individual SubEnvs to enter and leave a given federation over time.
- There must be some means by which the Foundation establishes communication paths with respect to each SubEnv, or vice versa. Such paths might be persistent, throughout the

operation of a SubEnv, or might be created temporarily as needed by the initiator. As in homogeneous federations, a single machine/single port assumption seems overly restrictive.

- The SubEnvs need not be directly aware of the other SubEnvs in the federation, per se, but there must be some means whereby the Foundation coordinates global process enactment, either by *notifying* a given SubEnv that it should or could perform specific tasks or by posting the request to some standard forum that each SubEnv *polls* to choose tasks it is able and willing to perform. Note this does not necessarily assume that SubEnvs have some means to inform the Foundation of pending tasks that they are unable or unwilling to do themselves. The Foundation could itself impose <u>all</u> tasks, perhaps through a special process modeling and enactment system intended to act as a "global hand" supporting some form of "super-workflow" [56], analogous to multi-part transactions submitted to heterogeneous database federations.

- Either every SubEnv must be perpetually executing, to perform the polling outlined above, or the Foundation must support a facility for "bringing up" a SubEnv each time it seeks to notify that SubEnv. If each single-site PCE participating in the federation supports a distinct start-up mechanism, then the Foundation needs some means to invoke new code (or scripts) on a per-component basis; actually, *all* communication could be handled this way, but would be relatively inefficient.

- If either the SubEnv or the Foundation may initiate interaction, then the communication infrastructure should preferably support *asynchronous* communications since both may attempt to contact each other at the same time (possibly resulting in deadlock as discussed with respect to homogeneous federation). If only one side can initiate the communication, as in pure notification or pure polling, the *synchronous* model is sufficient.

- When task enactment at one SubEnv involves access to data controlled by one or more other SubEnvs, the Foundation must provide mechanisms for transferring product artifacts and requisite process state among SubEnvs. As in homogeneous federations, data may be temporarily cached, permanently copied, or migrated between sites. Note enactment of such tasks may not be frequent in a heterogeneous federation, e.g., data exchange may be limited to scheduled milestones, whereas such multi-site tasks are expected to be more commonplace in a homogeneous federation. Recall a homogeneous federation can assume a standard data repository, although with differing local schemas, whereas heterogeneous federations also incur the problems of incompatible data formats; this is basically a distributed computing issue attacked by OMG and others through CORBA and similar layers, and not addressed further in this paper.

- If data is indeed shared or transferred among the participating SubEnvs, then distributed concurrency control and failure recovery mechanisms able to deal with heterogeneous data sources are needed. This is related to the transactional workflow concern in the database community; e.g., Meteor [68] incorporates a variety of repositories with and without their own transaction management. Again, this topic is outside the scope of this paper.

- It is unlikely that a heterogeneous collection of single-site PCEs would conveniently provide a common user interface, although the local SubEnvs of course include their own user interaction facilities. If the Foundation provides its own "global hand" notion of process, then presumably it must also supply a corresponding user interface.

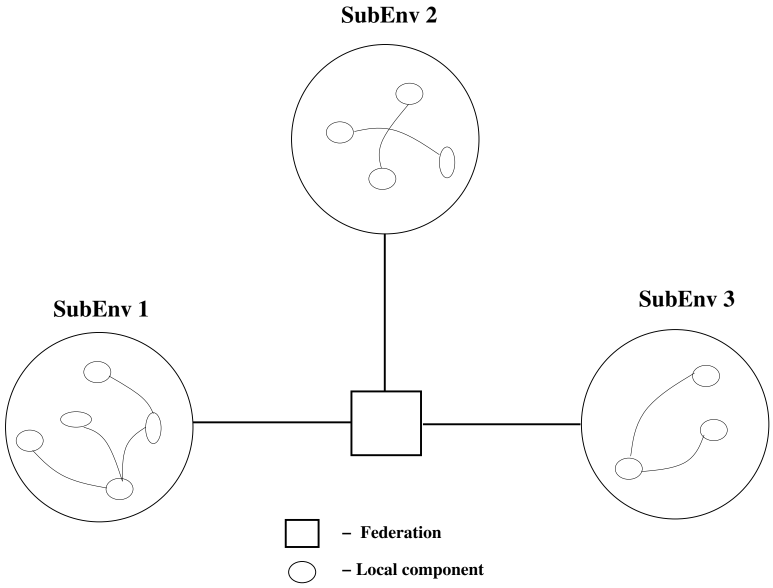

Figure 14: Centralized Glue Architecture for Heterogeneous Federation

- Finally, there should be some means for configuring a federation on a per-project basis. We anticipate this is considerably more difficult for heterogeneous than for homogeneous federations, and in the former case may involve substantial design and implementation to introduce a new PCE (i.e., if it was not previously integrated with the Foundation) rather than just invoking a pre-defined (re)configuration process.

4.2 Architectural Issues

Here there are two major categories of architectures:

- **Centralized glue:** The SubEnvs communicate through a single centralized component that implements the major functions of the Foundation, as shown in Figure 14. The external view of the architecture is the same as in Figure 2, except that the SubEnvs may be different systems rather than components of the same multi-site system. This approach is exemplified by ProcessWall. Mentor takes a similar tack, except that there may be *multiple* state servers and task servers, not just one (of each). Note that the Foundation is partially distributed, even in the conceptually centralized case, since a small interface piece must be attached to every local PCE. This is similar to the moderator of the moderated peer-to-peer architecture shown in Figure 4.

- **Distributed glue:** The Foundation is divided into multiple distributed components, one attached to each SubEnv; see Figure 15. This is the same as in Figure 3, except each of the SubEnvs may be a different system.

When a wide range of internal architectures is exhibited among the PCEs of interest, there is usually no obvious preference exhibited for centralized vs. decentralized "glue"; if the range is limited, the analysis from Section 2.2 would apply. As discussed there, top-down global processes would generally be more amenable to a centralized Foundation, and bottom-up to a decentralized

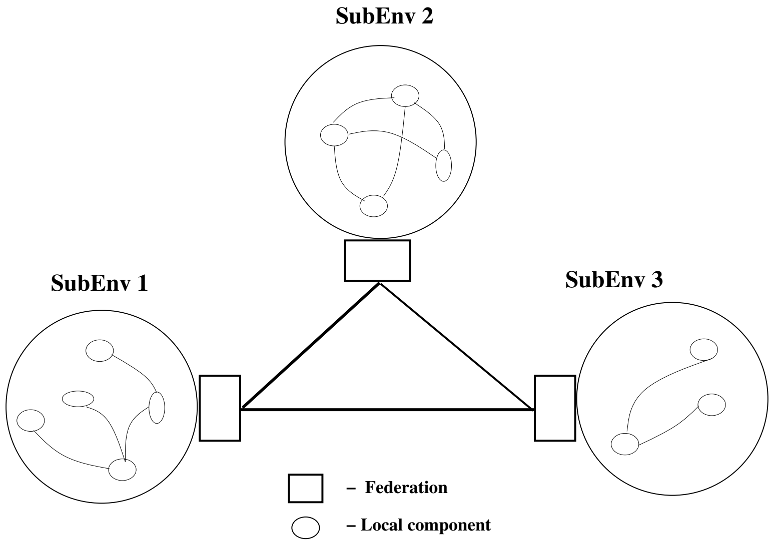

Figure 15: Distributed Glue Architecture for Heterogeneous Federation

Foundation. But the case is not so compelling for heterogeneous as for homogeneous federation, since construction of a global process using diverse process modeling languages and paradigms is so complex as to overwhelm all other concerns. This may be why Heimbigner proposes a third model for his process task server, neither top-down nor bottom-up, where *constructor* PCEs post pending tasks to the shared process repository and *constrainer* PCEs remove disallowed tasks from among those posted.

The following discussion considers both centralized and distributed federation architectures, where Oz plays the role of a constructor (and employs its own constraints prior to instantiating a task to post); other PCEs integrated into the same heterogeneous federation could, of course, act as constrainers on the posted tasks.

4.3 Integrating Oz into a Heterogeneous Federation

A heterogeneous federation is inherently more general than a homogeneous federation. Thus it is desirable to consider how a multi-site PCE like Oz might "fit" together with a process state (or task) server, the only proposed model we know of for heterogeneous federation of PCEs, where the federation also includes various non-Oz SubEnvs. One approach is to drop the homogeneous Foundation entirely and employ only the heterogeneous Foundation for multi-site tasks. Then the homogeneous SubEnvs — i.e., homogeneous with respect to system but heterogeneous with respect to process model — would be treated as if they were separate PCEs. Assuming that some component is added to interface with the federation "glue", this should work trivially if they fulfill the requirement of independent operation — that is, that they do not depend on each other in any way to perform totally local work. But then the main advantage of a homogeneous federation is lost, namely the relative ease with which SubEnvs can call on each other to perform specific agreed-upon services within the identical (and thus mutually understood) process modeling and enactment paradigm.

An alternative approach is to allow individual (or all) SubEnvs of a homogeneous federation to participate in one or more heterogeneous federations, while retaining the higher level of intimacy afforded by the system-level homogeneity when (intentionally) interacting with other local components of the same system. Note that SubEnvs that happen to participate in the same homogeneous federation may happen to employ each other's services indirectly through the heterogeneous federation, without necessarily any knowledge that they have more direct means of interaction. In fact, through this "backdoor" one might, in an unusual circumstance, inadvertently arrive in a situation where a SubEnv indirectly requests services from itself.

The main questions to answer are:

1. How would a SubEnv post to the state server those tasks it has instantiated but not initiated, and (generally speaking) would like some other PCE to perform. There are complications regarding representation of data arguments as part of the task specification, and later regarding data transfer when the task is enacted by some PCE participating in the federation; we ignore those here, except to note that something like World Wide Web URLs (uniform resource locators [63]) would probably suffice. Even though the task may eventually be picked up by another Oz SubEnv, this cannot be assumed a priori (if it could, direct interaction through the homogeneous Foundation would almost certainly be more efficient).

2. How would a state server ask a particular Oz SubEnv to perform specific posted tasks. This assumes some kind of scheduler or other entity to select among enabled tasks for enactment and choose the recipient SubEnv. The latter function might be achieved in the style of a broadcast message server like Field [55], where the SubEnv's *register* their interests in or abilities to perform certain kinds of tasks, perhaps by supplying a pattern that is matched by the task server against the enabled task specifications; the application of event subscription to workflow management system interoperability is suggested in [56]. Note this is a remarkably efficient form of "polling", differing from the conventional busy-wait or blocking receive primarily in that notifications may come in at times when the SubEnv is not immediately equipped to handle them. The only true polling currently supported by Oz is buried deep in one of several possible tool managers, and there looks only for files that have been created or modified by a "wrapped" tool invoked as part of some process task. A completely new interface would be needed to fit into a blackboard architecture.

3. How would an Oz SubEnv notify the state server of a task it had just completed. There are two cases: the task was previously posted to the state server by the same or a different SubEnv, or it arose entirely inside the given SubEnv (and thus is supplied only as historic information). In the former case, the state server might have requested that this SubEnv perform the task, or alternatively the SubEnv might have selected the task from among those enabled.

There are three different "levels" of task-like units supported by Oz. The lowest level is individual rule activities. Oz already defines a client/server protocol whereby user interface clients tell the server to apply a rule to a list of objectbase arguments, once the condition is deemed satisfied; the server supplies the client with file and primitive arguments and directs it to invoke the tool envelope specified in the rule activity; finally, the client returns to the server with status code and (optional) outputs, and the encapsulating rule (and its pending chain) continues.

It would not be very difficult to insert a new kind of client that receives such messages from the server and does something different than the typical user client; in fact, we've already introduced numerous special-purpose clients where the server tells the client for one reason or another about the activity to be carried out (only a few of these have been written up, see [44, 59, 67]). The process state server could be that new client, to implement point 1. Similarly, we could implement point 2 by treating the task server as the "client" in this same protocol. If the task server cannot be parameterized by new code to implement this protocol, then some mediation agent would be needed — in both directions, since presumably the process task server would expect some reply from the PCE indicating acceptance or refusal of the request. Finally, point 3 could be implemented by replacing the Oz server's role with the process state/task server in the above protocol, and either the Oz server or the Oz client (probably the former to limit communication connections) would operate as the client.

The intermediate level corresponds to full Oz rules, with condition and effect(s) as well as activity. Oz servers already transmit rule definitions between themselves as part of Treaty negotiation, and transmit the parameters and bound variables of instantiated rules as part of Summit enactment. The newest version of Oz, which replaces the native process engine with a process server component [39], among other things introduces a protocol for transferring instantiated rules (after condition evaluation but before activity initiation) between client and server to support delegation to and selection from user and group agendas ("to do" lists). These facilities might be combined and extended a bit to support all three points above regarding the process state/task server.

A complication: In the lowest level case the condition is already satisfied, by definition, prior to posting the activity, but this would not in general be true in the intermediate case. The ProcessWall and Mentor task representations allow for predecessors and successors, but not all the constraints embodied in Oz conditions are concerned with checking simple predecessor relationships (e.g., the bindings clause might find all objects that match a complex associative query and then the condition checks that at least one of those objects satisfies a complex logical clause), nor are all assertions made in Oz effects concerned with triggering successors (e.g., objects can be created and deleted, references formed and removed, etc.).

One could argue for a simplification, whereby Oz's postings to the Foundation are limited to those tasks whose conditions and effects are solely concerned with predecessor/successor relations that can be directly represented in the process state/task server. Although Oz's process modeling language tends to obscure such relationships from a human-readability standpoint, they are visible in the internal rule network compiled from the process model [34]. A less restrictive option would be to only post tasks with already satisfied conditions, or prerequisites in some other non-rule-based process modeling paradigm, but this prevents posting of tasks obligated for eventual completion [50], but not currently enabled.

A better approach might be to extend the process task server's task representation, or develop some additional control channel, for transmitting the conditions and/or effects from the Oz SubEnv to the (potentially) foreign SubEnv for evaluation within its paradigm, and vice versa regarding communicating any prerequisites and consequences that might be supported by the non-Oz paradigm to an Oz SubEnv (and of course both issues come up between pairs of non-Oz SubEnvs as well). If Oz were configured as a multi-site homogeneous federation where some (or all) sites happened to also belong to a heterogeneous federation, pending tasks posted through the Foundation to another Oz SubEnv (in the same federation) could include their conditions and effects in some *opaque* data stream understood only by Oz servers. So difficulties arise only when pending tasks posted through the Foundation involve non-Oz SubEnvs. Fortuitously, we have already shown fairly straightforward mappings from most of the major PCE paradigms, including Petri nets [53], task graphs [26], and grammars [38], into Oz rules, and reverse mappings are not inconceivable. And as previously noted, Mentor involves translation from one notation into another, as does the standard process interchange format work.

However, in the general case we also require substantial translation capabilities regarding both data formats and predicates and operations over those formats. The "universal data model" problem is a well-known unresolved, probably unresolvable [47], issue in database research. It may be possible to address a special case of this problem with respect to PCEs, e.g., if we assume the main data arguments are files and all attributes that might be referred to regarding task prerequisites and consequences are standard file appendages supported by most operating systems, such as owner, read/write timestamps, access permissions, etc.

The third task "level" in Oz is its notion of task — a full rule chain, i.e., all the rules emanating from some user-selected (or Foundation-requested, here the process task server) rule through backward and/or forward chaining. Oz's new "guidance chaining" already supports delegation of rules, and all subsequent chaining from those rules, to agendas [65]. The Oz process engine could alternatively *simulate* a full rule chain, with all alternatives, to unfold all predecessor/successor possibilities for the process task server (note the availability of backward chaining implies that not all predecessors have been exercised at the time a task is considered for execution). The simulation of branchings and iterations, with pruning to most probable paths, has been considered (for unrelated purposes) in [60]; some similar approach could be taken here to limit the cluttering of the server's task representation.

Or Oz could send additional activities to the process task server dynamically as alternatives are discarded or pursued. A variant would be for the task server to treat the entire rule chain as a unit, analogous to our implementation of TeamWare's [19] coarse-grained tasks as entry points into Oz rule chains as described in [39]. In either case, the intermediate model posed above requires relatively little extension.

Now the question arises as to how exactly Oz performs the mechanics of posting or notification (points 1 and 3). Unlike most PCEs, Oz's process engine is conveniently organized as a component (known as Amber when used independently) with a series of "callbacks" before and after every phase of rule execution. These callbacks are to subroutines in application-specific *mediator* code, which could be written for many purposes — among them for a SubEnv to interface to a heterogeneous Foundation, although the approach was not developed for that reason. To post a task after instantiation (with parameters) but before enactment, the *after* callback following the process engine's "begin" stage could be employed; to notify after a task has completed, the *before* callback preceding the "end" stage would be used (it is too late after the "end" stage, because the data structures containing all the relevant information would have been deallocated). Thus it would not be necessary to modify the process engine itself. Details on the mediator callback model can be found in [39, 52]; this is not a subject of this paper.

Now let's consider how Oz might receive requests from a heterogeneous federation controller component, or poll its worklist manager for opportunities to perform tasks on behalf of other SubEnvs, point 2. The client/server protocol messages mentioned above, now sent via TCP/IP sockets in Oz, could be converted by an intermediary attached to the federation "glue" (or added to each participating PCE implementation, probably more expensive) to whatever format is needed. It does not seem to matter much whether the task was previously posted, or the notification only informs the history manager, except that if it was previously posted it is reasonable to assume some kind of "checkout" model [33] whereby the SubEnv indicates that it intends to perform a given task and then acquires the data and other resources it needs to perform the work; later a "checkin" is achieved via the notification. An alternative to "checkout" is that the first (or last) notification wins, with respect to what the systems deems the result of the task, which seems a dubious proposition since unnecessary effort is required — often on the part of human users or computation-intensive tools.

5 Contributions and Future Directions

The main contributions of this work are:

- The elaboration of requirements and architectures for homogeneous and heterogeneous federations of process-centered environments. Both the homogeneous and heterogeneous federation architectures we present are in line with a proposed distributed workflow reference model [69].
- The design of a specific homogeneous federation architecture in Oz, described in sufficient detail to permit adaptation by another party to extend their own single-site PCE architecture to multi-site, as we did going from Marvel to Oz.
- A presentation of the issues that must be addressed to integrate an existing PCE into a heterogeneous federation based on the process state/task server (or worklist/history manager) approach.

The obvious next step is to complete an experimental integration between Oz and a process state/task server, assuming one has been fully implemented and is available. Evaluation against the heterogeneous federation requirements and lessons learned should prove interesting.

One area of current research by ourselves and others is federation of process-centered environments and computer-supported cooperative work (or "groupware") systems; see, e.g., [16, 2]. This should be followed by hybrid federations mixing and matching multiple PCE and CSCW components, perhaps based on a decentralized hypermedia infrastructure such as the World Wide Web; see [71] for a possible substrate for this direction.

Another major avenue of future research is to consider federation of PCEs in their role as tool integrators, i.e., the tools invoked during task execution directed by process enactment. This would extend the work we conducted in [67, 66] to facilitate remote execution of tools that "belong" to a remote SubEnv (in either the homogeneous or heterogeneous case), while for now we assume all tools are executed by local users and/or are available at all participating sites. Pragmatic concerns include architecture and operating system platform requirements, host and site licensing restrictions, running a tool within network file system services or at least relatively "near" the data in terms of network transmission costs, data security and "tunneling" through corporate firewalls, etc., introducing numerous technical challenges orthogonal to the problems solved here.

Acknowledgments

We would like to thank George Heineman, Peter Skopp, Steve Popovich, Jack Jingshuang Yang and Wenyu Jiang for their various work on Oz, Amber and the replacement of Oz's native process engine by Amber. We also thank Dennis Heimbigner, Alex Wolf and George Heineman for their first cut towards a Oz/ProcessWall integration as part of a funding agency presentation. We also

appreciate George Heineman's stimulating conversations on general interoperability and federation issues for workflow management systems. Connect to http://www.psl.cs.columbia.edu/ for current information about Oz licensing.

References

[1] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Gunthor, and Mohan U. Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *IFIP WG 8.1 Workgroup Conference on Information Systems Development for Decentralized Organizations*, Trondheim, Norway, August 1995.

[2] John E. Arnold and Steven S. Popovich. Integrating, customizing and extending environments with a message-based architecture. Technical Report CUCS-008-95, Columbia University, Department of Computer Science, September 1994. The research described in this report was conducted at Bull HN Information Systems, Inc.

[3] Michael Baentsch, Georg Molter, and Peter Sturm. WebMake: Integrating distributed software development in a structure-enhanced Web. In *3rd International World-Wide Web Conference*, Darmstadt, Germany, April 1995. Elsevier Science B.V. http://www.igd.fhg.de/www/www95/proceedings/papers/51/WebMake/WebMake.html.

[4] Sergio Bandinelli and Alfonso Fuggetta. Computational reflection in software process modeling: the SLANG approach. In *15th International Conference on Software Engineering*, pages 144–154, Baltimore MD, May 1993. IEEE Computer Society Press.

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

[6] 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.

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

[8] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[9] Israel Ben-Shaul and Gail E. Kaiser. *A Paradigm for Decentralized Process Modeling.* Kluwer Academic Publishers, Boston, 1995.

[10] Israel Z. Ben-Shaul. An object management system for multi-user programming environments. Master's thesis, Columbia University, Department of Computer Science, April 1991. CUCS-010-91.

[11] Israel Z. Ben-Shaul and George T. Heineman. A 3-level atomicity model for decentralized workflow management systems. Technical Report 1013, Technion, Israel Institute of Technology, Department of Electrical Engineering, January 1996. Submitted for publication.

[12] Israel Z. Ben-Shaul and Gail E. Kaiser. Process evolution in the Marvel environment. In Wilhelm Schäfer, editor, *8th International Software Process Workshop: State of the Practice in Process Technology*, pages 104–106, Wadern, Germany, March 1993. Position paper.

[13] Israel Z. Ben-Shaul and Gail E. Kaiser. A configuration process for a distributed software development environment. In *2nd International Workshop on Configurable Distributed Systems*, pages 123–134, Pittsburgh PA, March 1994.

[14] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the oz environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[15] Israel Z. Ben-Shaul and Gail E. Kaiser. An interoperability model for process-centered software engineering environments and its implementation in Oz. Technical Report CUCS-034-95, Columbia University Department of Computer Science, December 1995. Submitted for publication.

[16] Israel Z. Ben-Shaul and Gail E. Kaiser. Integrating groupware activities into workflow management systems. In *7th Israeli Conference on Computer Based Systems and Software Engineering*, Tel Aviv, Israel, June 1996. In press.

[17] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.

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

[19] Gregory Alan Bolcer and Richard N. Taylor. Endeavors: A process system integration infrastructure. Technical Report UCI-96-17, University of California at Irvine Department of Information and Computer Science, April 1996. Submitted for publication.

[20] Reidar Conradi, Espen Osjord, Per H. Westby, and Chunnian Liu. Initial software process management in EPOS. *Software Engineering Journal*, 6(5):275–284, September 1991.

[21] Stephen E. Dossick and Gail E. Kaiser. WWW access to legacy client/server applications. In *5th International World Wide Web Conference*, Paris, France, May 1996. In press.

[22] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.

[23] Carlo Ghezzi, editor. *9th International Software Process Workshop: The Role of Humans in the Process*, Airlie VA, October 1994. IEEE Computer Society Press.

[24] 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.

[25] 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.

[26] George T. Heineman. Automatic translation of process modeling formalisms. In *1994 Centre for Advanced Studies Conference (CASCON)*, pages 110–120, Toronto ON, Canada, November 1994. IBM Canada Ltd. Laboratory.

[27] George T. Heineman. *A Transaction Manager Component for Cooperative Transaction Models*. PhD thesis, Columbia University Department of Computer Science, 1996. CUCS-010-96. Forthcoming.

[28] George T. Heineman and Gail E. Kaiser. Integrating a transaction manager component with Process-WEAVER. Technical Report CUCS-012-94, Columbia University Department of Computer Science, May 1994.

[29] George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle WA, April 1995. ACM Press.

[30] George T. Heineman and Gail E. Kaiser. The CORD approach to extensible concurrency control. Technical Report CUCS-024-95 (revised), Columbia University Department of Computer Science, February 1996. Submitted for publication.

[31] 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.

[32] Watts Humphrey and Marc I. Kellner. Software process modeling: Principles of entity process models. In *11th Internation Conference on Software Engineering*, pages 331–342, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[33] Gail E. Kaiser. Cooperative transactions for multi-user environments. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 20, pages 409–433. ACM Press, New York NY, 1994.

[34] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[35] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[36] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[37] Gail E. Kaiser, George T. Heineman, Peter D. Skopp, and Jack J. Yang. Incremental process support for code reengineering: An update (experience report). Technical Report CUCS-007-96, Columbia University Department of Computer Science, February 1996. Submitted for publication.

[38] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In Walter F. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, chapter 2, pages 39–72. John Wiley & Sons, 1994.

[39] Gail E. Kaiser, Steven S. Popovich, and Stephen E. Dossick. A process server component supporting integration of heterogeneous process-centered environments. Technical report, Columbia University Department of Computer Science, May 1996.

[40] Henry F. Korth. Extending the scope of relational languages. *IEEE Software*, 3(1):19–28, January 1986.

[41] Programming Systems Lab. Darkover 1.0 manual. Technical Report CUCS-023-95e, Columbia University, Department of Computer Science, March 1995.

[42] Programming Systems Laboratory. Oz 1.1 Manual set, January 1996. ftp://ftp.cs.columbia.edu/pub/marvel/oz.1.1.manuals.

[43] Jintae Lee, Gregg Yost, and the PIF Working Group. The PIF process interchange format and framework, December 1994. http://www-sloan.mit.edu/ccs/pifmain.html.

[44] Wenke Lee, Gail E. Kaiser, Paul D. Clayton, and Eric H. Sherman. OzCare: A workflow automation system for care plans. Technical Report CUCS-012-96, Columbia University Department of Computer Science, March 1996. Submitted for publication.

[45] Workflow Management Coalition Members. Coalition overview, September 1995. http://www.aiai.ed.ac.uk/WfMC/overview.html.

[46] Microsoft announcements agreement with aspect software engineering, inc. to enhance active internet vision, March 12 1996. Press release. http://www.aspectse.com/Announce.html.

[47] Erich Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record*, 18(1):17–26, March 1989.

[48] Leon J. Osterweil. Presentation at Software Process Architectures Workshop, March 1995.

[49] Dewayne E. Perry, editor. *3nd International Conference on the Software Process: Applying Software Process*, Reston VA, October 1994. IEEE Computer Society Press.

[50] Dewayne E. Perry. Enactment control in Interact/Intermediate. In Brian Warboys, editor, *3rd European Workshop on Software Process Technology*, volume 772 of *Lecture Notes in Computer Science*, pages 107–113, Villard de Lans (Grenoble), France, February 1994. Springer-Verlag.

[51] Burkhard Peuschel and Stefan Wolf. Architectural support for distributed process centered software development environments. In Wilhelm Schäfer, editor, *8th International Software Process Workshop*, Wadern, Germany, March 1993. Position paper.

[52] Steven S. Popovich. *Rule-Based Process Servers for Software Development Environments.* PhD thesis, Columbia University Department of Computer Science, 1996. CUCS-014-96. Forthcoming.

[53] Steven S. Popovich and Gail E. Kaiser. Integrating an existing environment with a rule-based process server. Technical Report CUCS-004-95, Columbia University Department of Computer Science, August 1995.

[54] Sudha Ram, editor. *Special Issue on Heterogeneous Distributed Database Systems*, volume 24:12 of *Computer*. IEEE Computer Society Press, December 1991.

[55] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[56] Friedemann Schwenkreis. Workflow for the German federal government. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, Athens GA, May 1996. Position paper. In press.

[57] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

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

[59] Peter D. Skopp. Low bandwidth operation in a multi-user software development environment. Master's thesis, Columbia University Department of Computer Science, December 1995. CUCS-035-95.

[60] Peter D. Skopp and Gail E. Kaiser. Disconnected operation in a multi-user software development environment. In Bharat Bhargava, editor, *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 146–151, Princeton NJ, October 1993.

[61] Richard Mark Soley and William Kent. The OMG object model. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 2, pages 18–41. ACM Press, New York NY, 1994.

[62] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A language for software process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.

[63] L. Masinter T. Berners-Lee and M. McCahill. Uniform resource locators (url), December 1994. http://www.w3.org/hypertext/WWW/Addressoing/rfc1738.txt.

[64] Action Technologies, Inc. Action workflow metro, November 1995. http://www.actiontech.com/metrotour/resources/Metwp.htm.

[65] Andrew Z. Tong, Gail E. Kaiser, and Steven S. Popovich. A flexible rule-chaining engine for process-based software engineering. In *9th Knowledge-Based Software Engineering Conference*, pages 79–88, Monterey CA, September 1994. IEEE Computer Society Press.

[66] Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into process-centered environments. *Automated Software Engineering*. In press. Also available as Columbia University Department of Computer Science, CUCS-022-95, revised March 1996.

[67] Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into computer-aided software engineering environments. In *IEEE 7th International Workshop on Computer-Aided Software Engineering*, pages 40–48, Toronto Ontario, Canada, July 1995.

[68] M. Rusinkiewicz W. Jin, L. Ness and A. Sheth. Concurrency control and recovery of multi-database work flows in telecommunication applications. In *ACM SIGMOD Conference on Management of Data*, May 1993.

[69] Kurt Wallnau, Fred Long, and Anthony Earl. Toward a distributed, mediated architecture for workflow management. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, Athens GA, May 1996. Position paper. In press.

[70] Jeanine Weissenfels, Dirk Wodtke, Gerhard Weikum, and Angelika Kotz-Dittrich. The Mentor architecture for enterprise-wide workflow management. In *NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, Athens GA, May 1996. Position paper. In press.

[71] Jack Jingshuang Yang and Gail E. Kaiser. An architecture for integrating oodbs with www. In *5th International World Wide Web Conference*, Paris, France, May 1996. In press.